Note
Before using this information and the product it supports, read the information in "Notices" on page 781.

Fifth Edition (January 2014)
This edition applies to the Common Cryptographic Architecture (CCA) API, Release 4.2.10, 4.2.0, 4.1.0, 4.0.0, and 3.60 for Linux on IBM System z, and to all subsequent releases and modifications until otherwise indicated in new editions.
This edition replaces SC33-8294-03.
This book is for planning and programming purposes only.
IBM welcomes your comments. A form for readers' comments may be provided at the back of this document, or you may address your comments to the following address:

IBM Deutschland Research & Development GmbH
Information Development
Department 3248
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Internet e-mail: eservdoc@de.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
• Title and order number of this document
• Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corporation 2007, 2014.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z .......................... 3

Chapter 10. Financial services .................. 393

Chapter 13. TR-31 symmetric key management verbs .................. 523

Part 3. Appendixes ................................. 591

Appendix A. Return codes and reason codes .................. 593

Appendix B. Key token formats ........................ 609

Chapter 11. Using digital signatures .............. 467

Chapter 12. Managing PKA cryptographic keys .............. 477

AES Key Record Create (CSNBAKRC) ............... 352
AES Key Record Delete (CSNBAKRD) ............... 355
AES Key Record List (CSNBAKRL) ................. 357
AES Key Record Read (CSNBAKRR) ............... 360
AES Key Record Write (CSNBAKRW) ............... 362
DES Key Record Create (CSNBKRC) ............... 365
DES Key Record Delete (CSNBKRD) ............... 367
DES Key Record List (CSNBKRL) ................. 369
DES Key Record Read (CSNBKRR) ............... 372
DES Key Record Write (CSNBKRW) ............... 374
PKA Key Record Create (CSNDKRC) ............... 376
PKA Key Record Delete (CSNDKRD) ............... 378
PKA Key Record List (CSNDKRL) ................. 380
PKA Key Record Read (CSNDKRR) ............... 383
PKA Key Record Write (CSNDKRW) ............... 385
Retained Key Delete (CSNDRKD) ................. 387
Retained Key List (CSNDRKL) .................... 390

Chapter 10. Financial services .................. 393

How personal identification numbers (PINs) are used ......................................................... 393
How VISA card verification values are used ............................................................................ 393
Translating data and PINs in networks .................................................................................... 394
Working with Europay-Mastercard-Visa Smart cards ......................................................... 394
PIN verbs ............................................................................................................................... 395
ANSI X9.8 PIN restrictions .................................................................................................... 397
The PIN profile ....................................................................................................................... 399
Clear PIN Encrypt (CSNBCPE) ................. 405
Clear PIN Generate (CSNBPGN) ............... 408
Clear PIN Generate Alternate (CSNBCPA) .... 412
CVV Generate (CSNBCCG) ...................... 417
CVV Key Combine (CSNBCKC) ................. 421
CVV Verify (CSNBCSV) ......................... 426
Encrypted PIN Generate (CSNBEPG) ............ 430
Encrypted PIN Translate (CSNBETR) ............ 435
Encrypted PIN Verify (CSNBEPVR) .............. 441
PIN Change/Unblock (CSNBPCU) ............... 446
Secure Messaging for Keys (CSNBSKY) .......... 453
Secure Messaging for PINs (CSNBSPPN) ...... 457
Transaction Validation (CSNBTRV) .......... 462

Chapter 11. Using digital signatures .............. 467

Digital Signature Generate (CSNDDSG) ........ 468
Digital Signature Verify (CSNDDSV) .......... 473

Chapter 12. Managing PKA cryptographic keys .............. 477

PKA Key Generate (CSNDPKG) ............... 478
PKA Key Import (CSNDPKI) .................... 484
PKA Key Token Build (CSNDPKB) ............ 488
PKA Key Token Change (CSNDKTC) .......... 497
PKA Key Translate (CSNDPKT) ............... 500
PKA Public Key Extract (CSNDPKX) ............ 504
Remote Key Export (CSNDRKX) ............... 506
Trusted Block Create (CSNDTBC) .......... 517

Chapter 13. TR-31 symmetric key management verbs .............. 523

Key Export to TR31 (CSNBT31X) .............. 524
TR31 Key Import (CSNBT31I) ............... 553
TR31 Key Token Parse (CSNBT31P) .......... 578
TR31 Optional Data Build (CSNBT31O) .... 583
TR31 Optional Data Read (CSNBT31R) .... 587

Appendix A. Return codes and reason codes ......... 593

Appendix B. Key token formats ..................... 609

AES internal key token ......................... 609
Token Validation Value ......................... 610
DES internal key token ......................... 612
DES external key token ......................... 613
External RKX DES key tokens ................. 613
DES null key token ......................... 615
RSA public key token ......................... 615
RSA private external key key .................. 616
RSA private key token, 4096-bit
Modulus-Exponent external form ............ 618
RSA private key, 4096-bit Modulus-Exponent
format with AES encrypted OPK section
external form ...................................... 619
RSA private key, 4096-bit Chinese Remainder
Theorem format with AES encrypted OPK
section external form ......................... 621
RSA Private Key Token, 4096-bit Chinese
Remainder Theorem External Form ........ 623
RSA private internal key token .......... 624
RSA private key token, 4096-bit
Modulus-Exponent internal format ............ 628
RSA private key, 4096-bit Modulus-Exponent
format with AES encrypted OPK section
internal form ...................................... 630
RSA private key, 4096-bit Chinese Remainder
Theorem format with AES encrypted OPK
section internal form ......................... 632
RSA Private Key Token, 4096-bit Chinese
Remainder Theorem Internal Form ........ 634
ECC key token .................................... 636
PKA null key token ......................... 640
HMAC key token .................................. 640
HMAC variable-length symmetric key token .. 640
HMAC symmetric null key token ............. 646
TR-31 optional block data ..................... 646
Trusted blocks ..................................... 647
Trusted block organization .................... 648
Figures

1. CCA security API, access layer, and cryptographic engine .......................... 4
2. CPACF ........................................................................................................ 15
3. Control Vector Generate and Key Token Build
   CV keyword combinations for fixed-length DES key tokens .......................... 43
4. PKA key management .............................................................................. 66
5. Control vector base bit map (common bits and key-encrypting keys) ............ 670
6. Control vector base bit map (data operation keys) ........................................ 671
7. Control vector base bit map (PIN processing keys and cryptographic variable-encrypting keys) ................................................................. 672
8. Control vector base bit map (key generating keys) ....................................... 673
9. Control Vector Translate verb mask_array processing .................................. 680
10. Control Vector Translate verb ................................................................. 681
11. ISO-3 PIN-block format ........................................................................... 684
12. 3624 PIN generation algorithm ............................................................... 688
13. GBP PIN generation algorithm ................................................................. 689
14. PIN-Offset generation algorithm .............................................................. 690
15. PIN verification algorithm ........................................................................ 691
16. GBP PIN verification algorithm ............................................................... 693
17. PVV generation algorithm ....................................................................... 694
18. Triple-DES data encryption and decryption ................................................ 703
19. Enciphering using the ANSI X3.106 CBC method ...................................... 704
20. Deciphering using the CBC method .......................................................... 704
21. Enciphering using the ANSI X9.23 method ............................................... 705
22. Deciphering using the ANSI X9.23 method ............................................... 706
23. Triple-DES CBC encryption process ........................................................ 707
24. Triple-DES CBC decryption process ........................................................ 707
25. EDE algorithm ........................................................................................ 708
26. DED process ........................................................................................... 709
27. MAC calculation method ........................................................................ 710
28. Multiple encipherment of single-length keys ............................................. 713
29. Multiple decipherment of single-length keys ............................................ 714
30. Multiple encipherment of double-length keys .......................................... 715
31. Multiple decipherment of double-length keys .......................................... 716
32. Multiple encipherment of triple-length keys ............................................. 717
33. Multiple decipherment of triple-length keys ............................................. 718
34. Syntax, sample routine in C ................................................................. 743
35. Syntax, sample routine in Java ............................................................... 748
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Verbs that ignore AUTOSELECT</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Key types</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>Key subtypes specified by the <code>rule_array</code> keyword</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>Access Control Points Used by ATM remote key loading</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>Summary of CCA nodes and resource control verbs</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>Summary of CCA AES, DES, and HMAC verbs</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>Summary of PKA key token sections</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>Summary of PKA verbs</td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>TR-31 symmetric key management verbs</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>Keywords for Cryptographic Facility Query control information</td>
<td>79</td>
</tr>
<tr>
<td>11</td>
<td>Cryptographic Facility Query information returned in the <code>rule_array</code></td>
<td>81</td>
</tr>
<tr>
<td>12</td>
<td>Output data format for the <code>STATDECT</code> keyword</td>
<td>93</td>
</tr>
<tr>
<td>13</td>
<td>Output data format for <code>STATICS</code> operational key parts</td>
<td>94</td>
</tr>
<tr>
<td>14</td>
<td>Output data format for <code>STATICS</code> operational key parts</td>
<td>96</td>
</tr>
<tr>
<td>15</td>
<td>Output data format for <code>STATICE</code> operational key parts</td>
<td>99</td>
</tr>
<tr>
<td>16</td>
<td>Output data format for <code>STATICSX</code> operational key parts</td>
<td>101</td>
</tr>
<tr>
<td>17</td>
<td>Output data format for <code>STATKPR</code> operational key parts</td>
<td>104</td>
</tr>
<tr>
<td>18</td>
<td>Output data format for <code>STATV</code> operational key parts</td>
<td>105</td>
</tr>
<tr>
<td>19</td>
<td>Keywords for Cryptographic Resource Allocate control information</td>
<td>111</td>
</tr>
<tr>
<td>20</td>
<td>Keywords for Cryptographic Resource Deallocate control information</td>
<td>114</td>
</tr>
<tr>
<td>21</td>
<td>Keywords for Key Storage Initialization control information</td>
<td>116</td>
</tr>
<tr>
<td>22</td>
<td>Keywords for Master Key Process control information</td>
<td>120</td>
</tr>
<tr>
<td>23</td>
<td>Keywords for Random Number Tests control information</td>
<td>124</td>
</tr>
<tr>
<td>24</td>
<td>Keywords for Multiple Clear Key Import control information</td>
<td>131</td>
</tr>
<tr>
<td>25</td>
<td>Keywords for Control Vector Translate control information</td>
<td>138</td>
</tr>
<tr>
<td>26</td>
<td>Keywords for Diversified Key Generate control information</td>
<td>149</td>
</tr>
<tr>
<td>27</td>
<td>Skeleton key-tokens</td>
<td>154</td>
</tr>
<tr>
<td>28</td>
<td>Keywords for EC Diffie-Hellman control information</td>
<td>156</td>
</tr>
<tr>
<td>29</td>
<td>Keywords for the Key Generate verb key_form parameter</td>
<td>168</td>
</tr>
<tr>
<td>30</td>
<td>Key length values for the Key Generate verb</td>
<td>169</td>
</tr>
<tr>
<td>31</td>
<td>Key Generate - key lengths for each key type</td>
<td>170</td>
</tr>
<tr>
<td>32</td>
<td>Keywords for Key Generate, valid key types and key forms for a single key</td>
<td>174</td>
</tr>
<tr>
<td>33</td>
<td>Keywords for Key Generate2 control information</td>
<td>174</td>
</tr>
<tr>
<td>34</td>
<td>Keywords for Key Generate2 control information</td>
<td>177</td>
</tr>
<tr>
<td>35</td>
<td>Keywords and associated algorithms for <code>key_type_1</code> parameter</td>
<td>178</td>
</tr>
<tr>
<td>36</td>
<td>Keywords and associated algorithms for <code>key_type_2</code> parameter</td>
<td>178</td>
</tr>
<tr>
<td>37</td>
<td>Key Generate2 valid key type and key form for one key</td>
<td>182</td>
</tr>
<tr>
<td>38</td>
<td>Key Generate2 valid key type and key forms for two keys</td>
<td>182</td>
</tr>
<tr>
<td>39</td>
<td>AES KEK strength required for generating an HMAC key under an AES KEK</td>
<td>182</td>
</tr>
<tr>
<td>40</td>
<td>Keywords for Key Part Import control information</td>
<td>188</td>
</tr>
<tr>
<td>41</td>
<td>Keywords for Key Part Import2 control information</td>
<td>192</td>
</tr>
<tr>
<td>42</td>
<td>Key Test parameter changes</td>
<td>195</td>
</tr>
<tr>
<td>43</td>
<td>Key Test GENERATE outputs and VERIFY inputs</td>
<td>195</td>
</tr>
<tr>
<td>44</td>
<td>Keywords for Key Test control information</td>
<td>196</td>
</tr>
<tr>
<td>45</td>
<td>Keywords for Key Test2 control information</td>
<td>201</td>
</tr>
<tr>
<td>46</td>
<td>Keywords for Key Test Extended control information</td>
<td>206</td>
</tr>
<tr>
<td>47</td>
<td>Keywords for Key Token Build control information</td>
<td>210</td>
</tr>
<tr>
<td>48</td>
<td>Keywords for Key Token Build2 control information</td>
<td>215</td>
</tr>
<tr>
<td>49</td>
<td>Keywords for Key Token Change control information</td>
<td>218</td>
</tr>
<tr>
<td>50</td>
<td>Keywords for Key Token Change2 control information</td>
<td>221</td>
</tr>
<tr>
<td>51</td>
<td>Keywords for Key Token Parse control information</td>
<td>225</td>
</tr>
<tr>
<td>52</td>
<td>Keywords for Key Token Parse2 control information</td>
<td>231</td>
</tr>
<tr>
<td>53</td>
<td>Keywords for Key Translate2 control information</td>
<td>242</td>
</tr>
<tr>
<td>54</td>
<td>Keywords for PKA Decrypt control information</td>
<td>245</td>
</tr>
<tr>
<td>55</td>
<td>Keywords for PKA Encrypt control information</td>
<td>248</td>
</tr>
<tr>
<td>56</td>
<td>Keywords for Restrict Key Attribute control information</td>
<td>256</td>
</tr>
<tr>
<td>57</td>
<td>Keywords for Random Number Generate form parameter</td>
<td>260</td>
</tr>
<tr>
<td>58</td>
<td>Keywords for Random Number Generate Long control information</td>
<td>262</td>
</tr>
<tr>
<td>59</td>
<td>Keywords for Symmetric Key Export control information</td>
<td>266</td>
</tr>
<tr>
<td>60</td>
<td>AES EXPORTER strength required for exporting an HMAC key under an AES EXPORTER</td>
<td>269</td>
</tr>
</tbody>
</table>
61. Minimum RSA modulus strength required to contain a PKOAEP2 block when exporting an AES key ........................................ 269
62. Minimum RSA modulus length to adequately protect an AES key ................................................................. 269
63. Keywords for Symmetric Key Generate control information ................................................................. 272
64. Keywords for Symmetric Key Import control information ................................................................. 276
65. Keywords for Symmetric Key Import2 control information ................................................................. 280
66. PKCS#1 OAEP encoded message layout (PKOAEP2) ................................................................. 283
67. Keywords for Decipher control information .............................................................................. 290
68. Keywords for Encipher control information ........................................................................ 295
69. Keywords for Symmetric Algorithm Decipher control information ........................................... 300
70. Keywords for Symmetric Algorithm Encipher control information ........................................... 307
71. Keywords for HMAC Generate control information .............................................................................. 316
72. Keywords for HMAC Verify control information ........................................................................... 319
73. Keywords for MAC Generate control information ............................................................................. 324
74. Keywords for MAC Verify control information ............................................................................. 328
75. Keywords for MDC Generate control information ............................................................................. 334
76. Keywords for One-Way Hash control information ............................................................................. 342
77. Valid symbols for the name token ......................................................................................... 347
78. Key labels that are not valid ...................................................................................................... 347
79. Keywords for AES Key Record Delete control information ......................................................... 355
80. Keywords for AES Key Record Write control information .......................................................... 362
81. Keywords for DES Key Record Delete control information .......................................................... 367
82. Keywords for PKA Key Record Delete control information ......................................................... 378
83. Keywords for PKA Key Record Write control information ........................................................ 385
84. ANSI X9.8 PIN - Allow only ANSI PIN blocks ............................................................................. 398
85. Format of a PIN profile ............................................................................................................ 399
86. Format values of PIN blocks .................................................................................................. 400
87. PIN block format and PIN extraction method keywords ......................................................... 400
88. Verbs affected by enhanced PIN security mode .................................................................... 402
89. Format of a pad digit ............................................................................................................... 402
90. Pad digits for PIN block formats ............................................................................................ 402
91. Format of the Current Key Serial Number Field .................................................................... 403
92. Keywords for Clear PIN Encrypt control information .............................................................. 406
93. Keywords for Clear PIN Generate control information ............................................................. 409
94. Array elements for the Clear PIN Generate verb ....................................................................... 409
95. Array elements for Clear PIN Generate ..................................................................................... 410
96. Keywords for Clear PIN Generate Alternate control information ........................................ 413
97. Array elements for Clear PIN Generate Alternate, data_array (IBM-PINO) ......................... 414
98. Array elements for Clear PIN Generate Alternate, data_array (VISA-PVV) ......................... 415
99. Keywords for CVV Generate control information ........................................................................... 418
100. Key-wrapping matrix for the CVV Key Combine verb .......................................................... 423
101. Keywords for CVV Verify control information .............................................................................. 426
102. Keywords for Encrypted PIN Generate control information .................................................... 431
103. Array elements for Encrypted PIN Generate data_array parameter ........................................... 432
104. Keywords for Encrypted PIN Generate control information ....................................................... 432
105. Keywords for Encrypted PIN Translate control information ...................................................... 437
106. Additional names for PIN formats ............................................................................................ 440
107. Keywords for Encrypted PIN Verify control information .............................................................. 443
108. Array elements for Encrypted PIN Verify data_array parameter ............................................... 444
109. Array elements required by the process rule ............................................................................. 444
110. Keywords for PIN Change/Unblock control information ............................................................. 447
111. Keywords for Secure Messaging for Keys control information ..................................................... 453
112. Keywords for Secure Messaging for PINs control information ...................................................... 458
113. Keywords for Transaction Validation control information ............................................................. 462
114. Values for Transaction Validation validation_values parameter ................................................. 464
115. Keywords for Digital Signature Generate control information ..................................................... 469
116. Keywords for Digital Signature Verify control information ............................................................. 474
117. Keywords for PKA Key Generate control information ................................................................. 479
118. Keywords for PKA Key Import control information ........................................................................... 485
119. Keywords for PKA Key Token Build control information ............................................................ 489
120. PKA Key Token Build - Key value structure length maximum values ..................................... 490
121. PKA Key Token Build - Key value structure elements .................................................................... 490
122. Keywords for PKA Key Token Change control information ............................................................. 498
123. Keywords for PKA Key Translate control information ................................................................. 501
124. Keywords for Remote Key Export certificate_parms parameter ..................................................... 510
125. Keywords for Trusted Block Create control information ............................................................... 519
126. Export translation table for a TR-31 BDK base derivation key (BDK) ........................................... 531
127. Export translation table for a TR-31 CVK card verification key (CVK) ........................................ 533
<p>| 128. Export translation table for a TR-31 data encryption key (ENC). | 534 |
| 129. Export translation table for a TR-31 key encryption or wrapping, or key block protection key (KEK or KEK-WRAP). | 535 |
| 130. Export translation table for a TR-31 ISO MAC algorithm key (ISOMACn). | 536 |
| 131. Commands | 537 |
| 132. Export translation table for a TR-31 PIN encryption or PIN verification key (PINENC, PINV, PINV3624, VISAIPV). | 540 |
| 133. Export translation table for a TR-31 EMV/Chip issuer master-key key (DKYGENK, DATA). | 543 |
| 134. Import translation table for a TR-31 DPK base derivation key (usage &quot;B0&quot;). | 558 |
| 135. Import translation table for a TR-31 CVK card verification key (usage &quot;C0&quot;). | 558 |
| 136. Import translation table for a TR-31 data encryption key (usage &quot;D0&quot;). | 560 |
| 137. Import translation table for a TR-31 key encryption or wrapping, or key block protection key (usages &quot;K0&quot;, &quot;K1&quot;). | 560 |
| 138. Import translation table for a TR-31 ISO MAC algorithm key (usages &quot;M0&quot;, &quot;M1&quot;, &quot;M3&quot;). | 562 |
| 139. Import translation table for a TR-31 PIN encryption or PIN verification key (usages &quot;P0&quot;, &quot;V0&quot;, &quot;V1&quot;, &quot;V2&quot;). | 563 |
| 140. Commands | 566 |
| 141. Import translation table for a TR-31 EMV/Chip issuer master-key key (usages &quot;E0&quot;, &quot;E1&quot;, &quot;E2&quot;, &quot;E3&quot;, &quot;E4&quot;, &quot;E5&quot;) | 568 |
| 142. TR31 Key Import CV sources | 572 |
| 143. TR31 Key Import protection methods | 573 |
| 144. Return code values | 593 |
| 145. Reason codes for return code 0 | 594 |
| 146. Reason codes for return code 4 | 595 |
| 147. Reason codes for return code 8 | 595 |
| 148. Reason codes for return code 12 | 606 |
| 149. Reason codes for return code 16 | 607 |
| 150. AES Internal key token format, version X'04' | 609 |
| 151. AES internal key-token flag byte | 610 |
| 152. Internal clear key token format. | 611 |
| 153. DES internal key token format. | 612 |
| 154. DES external key token format. | 613 |
| 155. External RKX DES key-token format, version X'10' | 614 |
| 156. DES null key token format | 615 |
| 157. RSA Public Key Token format | 615 |
| 158. RSA private external key token basic record format | 616 |
| 159. RSA private key token, 1024-bit Modulus-Exponent external format | 617 |
| 160. RSA Private Key Token, 4096-bit Modulus-Exponent External Format | 618 |
| 161. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') external form | 619 |
| 162. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') external form | 621 |
| 163. RSA Private Key Token, 4096-bit Chinese Remainder Theorem External Format | 623 |
| 164. RSA private internal key token basic record format | 625 |
| 165. RSA private internal key token, 1024-bit Modulus-Exponent format for cryptographic coprocessor feature | 626 |
| 166. RSA private internal key token, 1024-bit Modulus-Exponent starting with CEX3C | 627 |
| 167. Private key, 4096-bit Modulus-Exponent format section (X'09') | 629 |
| 168. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') internal form | 630 |
| 169. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') internal form | 632 |
| 170. RSA Private Internal Key Token, 4096-bit Chinese Remainder Theorem Internal Format. | 634 |
| 171. RSA variable Modulus-Exponent token format | 636 |
| 172. Supported Prime elliptic curves by size, name, and object identifier | 637 |
| 173. Supported Brainpool elliptic curves by size, name, and object identifier | 637 |
| 174. ECC private-key section | 637 |
| 175. ECC public-key section | 639 |
| 176. PKA null key token format | 640 |
| 177. HMAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) | 640 |
| 178. HMAC symmetric null key token format | 646 |
| 179. IBM optional block data in a TR-31 key block | 647 |
| 180. Trusted block sections and their use | 648 |
| 181. Trusted block header format | 650 |
| 182. Trusted block trusted RSA public key section (X'11') | 651 |
| 183. Trusted block rule section (X'12') | 652 |
| 184. Summary of trusted block X'12' subsections | 653 |
| 185. Transport key variant subsection (X'0001') of trusted block rule section (X'12') | 654 |
| 186. Transport key rule reference subsection (X'0002') of trusted block rule section (X'12') | 655 |
| 187. Common export key parameters subsection (X'0003') of trusted block rule section (X'12') | 655 |
| 188. Source key rule reference subsection (X'0004') of trusted block rule section (X'12') | 657 |
| 189. Export key CCA token parameters subsection (X'0005') of trusted block rule section (X'12') | 657 |
| 190. Trusted block key label (name) section (X'13') | 659 |
| 191. Trusted block information section (X'14') | 659 |
| 192. Summary of trusted block information subsections | 660 |
| 193. Protection information subsection (X'0001') of trusted block information section (X'14') | 660 |
| 194. Activation and expiration dates subsection (X'0002') of trusted block information section (X'14') | 661 |
| 195. Trusted block application-defined data section (X'15') | 662 |
| 196. Default control vector values | 667 |
| 197. Main key type bits | 674 |
| 198. Key subtype bits | 674 |</p>
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculation method keyword bits</td>
<td>675</td>
</tr>
<tr>
<td>INGEN, IPINENC, and OPINENC key bits</td>
<td>676</td>
</tr>
<tr>
<td>Generic key type bits</td>
<td>676</td>
</tr>
<tr>
<td>DKYGENKY key type bits</td>
<td>677</td>
</tr>
<tr>
<td>Versions of the MDC calculation method</td>
<td>701</td>
</tr>
<tr>
<td>MDC calculation procedures</td>
<td>701</td>
</tr>
<tr>
<td>PKA96 clear DES key record</td>
<td>718</td>
</tr>
<tr>
<td>Access Control Points and corresponding CCA verbs</td>
<td>724</td>
</tr>
<tr>
<td>Verbs called by the sample routines</td>
<td>743</td>
</tr>
<tr>
<td>CCA groups</td>
<td>764</td>
</tr>
</tbody>
</table>
About this document

This document describes how to use the verbs provided in the Common
Cryptographic Architecture (CCA) Release 4.2.10, 4.2.0, 4.1.0, 4.0.0, and 3.60 APIs
for Linux on IBM® System z®.

See “Terminology” on page xxi for the correct CCA feature terminology.

The CCA functions perform cryptographic operations using the following in
coprocessor mode:
- IBM 4765 Crypto Express4 feature (CEX4C)
- IBM 4765 Crypto Express3 feature (CEX3C)
- IBM 4764 Crypto Express2 feature (CEX2C)

See “Concurrent installations” on page 767 for details.

This book is for planning and programming purposes only.

The CCA host software provides an application programming interface through
which applications request secure, high-speed cryptographic services from the
hardware cryptographic features.

Where CCA Release 4.2.10 has been changed or enhanced from CCA Release 4.2.0,
these changes have been noted.

Revision history

Revision history for this document for each CCA Support Program release.

Fifth edition, January 2014, CCA Support Program Release
4.2.10

This edition describes the IBM CCA Basic Services API for Release 4.2.10.

For the supported environments and product ordering information, see:

http://www.ibm.com/security/cryptocards

For Linux for IBM System z, Release 4.2.10, changes to the CCA API include the
following added support:
- DEV-ANY (AUTOSELECT) support added via CSUACRA, CSUACRD,
environment variable CSU_DEFAULT_ADAPTER, for automatic (simple) load
balancing. See “AUTOSELECT option” on page 9.
- CCA key storage support for DES key tokens with Zero CVs.
- CEX4C support and recognition for all verbs and utilities.

Fourth edition, February 2012, CCA Support Program Release
4.2

This edition describes the IBM CCA Basic Services API for Release 4.2.

For the supported environments and product ordering information, see:
Two problems have been discovered with the CCA microcode related to the reenciphering of master keys. Although similar, the two problems are slightly different and exist in different levels of the microcode. These problems could lead to a loss of operational private keys after a master key change. Symmetric keys are not affected. Although it is expected few customers will be impacted this document describes the problems and how to recover.

For details, see http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/FLASH10764.

The PKA Key Token Change (CSNDKTC) verb has been changed to not permit the use of the RTNMK keyword for processor firmware levels that have this problem.

For Linux for IBM System z, Release 4.2, changes to the CCA API include:

- These new verbs:
  - CVV Key Combine (CSNBCKC)
  - EC Diffie-Hellman (CSNDEDH)
  - Key Export to TR31 (CSNB31X)
  - Key Token Parse2 (CSNBKTP2)
  - TR31 Key Import (CSNB31I)
  - TR31 Key Token Parse (CSNB31P)
  - TR31 Optional Data Build (CSNB31O)
  - TR31 Optional Data Read (CSNB31R)

- Added support to improve security and control for PIN decimalization tables, and changes to the required access control points for these verbs:
  - Clear PIN Generate (CSNBPGN)
  - Clear PIN Generate Alternate (CSNBCPA)
  - Encrypted PIN Generate (CSNBEGP)
  - Encrypted PIN Verify (CSNBEPV)

- Added support for double-length keys to these verbs:
  - CVV Generate (CSNBCSG)
  - CVV Verify (CSNBCSV)

- Added support for variable-length AES keys to these verbs:
  - Symmetric Algorithm Decipher (CSNBSAD)
  - Symmetric Algorithm Encipher (CSNBSAE)

- Added rule-array keywords SHA-1 and SHA-256 to these verbs:
  - Symmetric Key Generate (CSNDSYG)
  - Symmetric Key Import (CSNDSYI)

- Added key usage NOT31XPT and T31XPTOK to this verb:
  - Control Vector Generate (CSNBCVG)

- Updated control vector key combinations for fixed-length DES key tokens, for these verbs:
  - Control Vector Generate (CSNCCVG)
  - Key Token Build (CSNCCBTB)

- Added new rule-array keyword AES to these verbs:
  - Key Part Import2 (CSNCKPI2)
Third edition, March 2011, CCA Support Program Release 4.1.0

This edition describes the IBM CCA Basic Services API for Release 4.1.0.

For the supported environments and product ordering information, see:

http://www.ibm.com/security/cryptocards

For Linux for IBM System z, Release 4.1.0 changes to the CCA API include:

- Enhanced PIN security with the addition of ANSI X9.8 restriction capabilities
  Three new access control points are added to enhance PIN security by blocking
  PIN attacks. See the Required commands sections of the Clear PIN Generate
  Alternate (CSNBCPA), Encrypted PIN Translate (CSNBPTR), and Secure
  Messaging for PINs (CSNBSPN) verbs.

- Wrap CCA keys in Cipher-Block Chaining (CBC) mode
  A second key-wrapping method is added for DES that is a more secure version
  of Triple-DES ECB mode currently used by CCA. The enhanced version of key
  wrapping complies with current cryptographic standards that require key
  bundling. This new key-wrapping method can coexist with the CCA legacy ECB
  mode of wrapping Triple-DES keys. The two methods can coexist on the same or
  multiple systems.

- Elliptic Curve Cryptography (ECC) support
  New Elliptic Curve Cryptography (ECC) key generation, along with support for
digital signature generation and verification using the Elliptic Curve Digital
Signature Algorithm (ECDSA). This enhancement includes a new PKA key-token
for housing ECC public-key cryptographic keys and a new asymmetric APKA
master-key (32-byte AES key) for wrapping an ECC key-token, along with added
support to the Master Key Process verb.
Hashed Message Authentication Code (HMAC) support for key generation and processing, but not for key storage.

These new verbs:
- HMAC Generate (CSNBHMG)
- HMAC Verify (CSNBHMV)
- Key Generate2 (CSNBKGN2)
- Key Part Import2 (CSNBKPI2)
- Key Test2 (CSNBKTYT2)
- Key Token Build2 (CSNBKTB2)
- Key Token Change2 (CSNBKTC2)
- Key Translate2 (CSNBKTR2)
- Restrict Key Attribute (CSNBRKA)
- Symmetric Key Import2 (CSNDSYI2)

**Second edition, April 2010, CCA Support Program Release 4.0.0**

This edition describes the IBM CCA Basic Services API for Release 4.0.0. For the supported environments and product ordering information.


For Linux for IBM System z, release 4.0.0 changes to the CCA API include:
- Support for the IBM Crypto Express3 feature (CEX3C) in coprocessor mode
- A Java™ Native Interface (JNI) form for most of the verbs
- Central Processor Assist for Cryptographic Functions (CPACF) support
- These new verbs:
  - AES Key Record Create (CSNBAKRC)
  - AES Key Record Delete (CSNBAKRD)
  - AES Key Record List (CSNBAKRL)
  - AES Key Record Read (CSNBAKRR)
  - AES Key Record Write (CSNBAKRW)
  - Control Vector Translate (CSNBCVT)
  - Cryptographic Facility Version (CSUACFV)
  - Cryptographic Variable Encipher (CSNBCVE)
  - Key Test Extended (CSNBKTYTX)
  - MDC Generate (CSNBMGDG)
  - PKA Key Translate (CSNDPKT)
  - Prohibit Export Extended (CSNBPEXX)
  - Random Number Generate Long (CSNBRNGL)
  - Remote Key Export (CSNDRKX)
  - Retained Key Delete (CSNDRKD)
  - Retained Key List (CSNDRKL)
  - Symmetric Algorithm Decipher (CSNSAD)
  - Symmetric Algorithm Encipher (CSNSAE)
  - Trusted Block Create (CSNDTBC)
Who should use this document

This document is intended for application programmers who are responsible for writing application programs that use the security application programming interface (API) to access cryptographic functions.

Distribution-specific information

In order to use the full set of CCA Release 4.2.0 and later functions, a Linux on System z distribution with support for the CEX*C feature is required.

- CCA Linux on System z maintenance Release 4.2.10 adds support for CEX4C.

These distributions introduced CEX4C exploitation:

- SUSE Linux Enterprise Server 11 SP3 (SLES 11 SP3) 64-bit only
- Red Hat Enterprise Linux 6 Update 4 (64-bit only)

- SUSE Linux Enterprise Server 11 SP2 (SLES 11 SP2) 64-bit only
- SUSE Linux Enterprise Server 10 SP4 (SLES 10 SP4) 64-bit only
- Red Hat Enterprise Linux 5 Update 7 (64-bit only)
- Red Hat Enterprise Linux 6 Update 2 (64-bit only)
- Distributions listed for CCA 4.1.0 below

The following Linux on System z distributions support CCA Release 4.1.0 host software for use with CEXC3. Support also extends to CEX2C within the functional scope of the CEX2C feature:

- SUSE Linux Enterprise Server 11 SP1 (SLES 11 SP1)
- SUSE Linux Enterprise Server 10 SP3 (SLES 10 SP3)
- Red Hat Enterprise Linux 5 Update 6
- Red Hat Enterprise Linux 6

Note:

1. With the addition of CEX4C support, the name of the default install path (/opt/IBM/CEX3C/) has changed. For Release 4.2.10, a new default install path is created at /opt/IBM/CCA/. The former install path /opt/IBM/CEX3C/ is linked to the new path to assist migration. If upgrading, note that the 4.2.10 rpm will copy your key storage files from the old default location to the new default location. The old directory will also be renamed to be /opt/IBM/CEX3C-old/.

2. CEX*C can be loaded with firmware newer than release 4.2.10. This host package supports the newer firmware, up to the verbs and keywords that it is aware of, consistent with firmware as of 4.2.10.

3. Applications linked with prior CCA host software will continue to function if the CCA host software is upgraded in place, however IBM always recommends full testing of upgrades before implementing production roll-out.

4. CCA 4.2 host software supports all prior levels of CEX3C adapter firmware and may also be used for support of the CEX2C subset of functionality. Full CEX2C support is included in CCA Release 4.2.0 and 4.1.0. However, note that because of limits in the CEX2C hardware and firmware available, this is a limited subset of the CEX*C functions described in this document.

5. For current restrictions and recommendations about the usage of CCA 4.2.0 and later releases with Linux on System z distributions, refer to http://www.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml

For a summary, see “Restrictions” on page xx.
6. Only 64-bit versions of this software are provided. 31-bit support is not provided.

---

**Restrictions**

This topic preserves known restrictions for current and prior releases to assist system administrators.

CCA 4.2 and later releases on Linux on System z:

**Linux distribution applicability**


**General note**

The CCA host library (libcsulccas.so) was changed to allow more flexible preparation of requests to be sent to the adapter. The change allows very large key block support, among other changes. The z90crypt device driver as it exists in currently available Linux distributions has the same limitation as older CCA host library code, and so a patch was prepared and submitted for maintenance releases to 'in-service' Linux distributions.

IBM is working with our distribution partners to include a patch to remove these noted restrictions in future distributions.

**General description of the restriction**

Verbs that may send or return a lot of data (of certain types, such as lists of key labels or key tokens) or large key tokens are limited by an issue in the current version of the z90crypt device driver buffer handling to a smaller amount of data or key token size than would normally be allowed.

The following scenarios clarify what to avoid to prevent this restriction from leading to errors.

**Restriction scenario: Sending or requesting a large amount of certain types of data**

**CSNDRKXL**

This verb returns a list of labels or tokens for a specified set of retained keys. Specifying a large number for the key_labels_count or retained_keys_count parameter can result in more return data than the cryptographic device driver can handle. Because a key label is 64 bytes, do not specify key_labels_count values greater than 75.

Crossing this limit results in return code 8 reason code 1106 error, indicating the data is too large to be returned (it would be truncated).

**CSNDRKXX**

This verb has the potential to send large objects for parameters certificate, certificateParms and extra_data. Avoid using a combined value for these parameters that greatly exceeds 4096 Bytes. The actual value of the threshold varies with the size of other parameters and so cannot be specified exactly.
Restriction scenario: Processing extremely large key tokens

CSNBT31I, CSNBT31X, CSNBKYT2

These verbs handle TR-31 key blocks, which can be up to 9992 Bytes (if 9 KB or more of optional block sections have been added). Due to the z90crypt restriction, TR-31 key blocks should be built specifying no more than 4096 Bytes of optional block sections.

Crossing this limit results in error return code 8 reason code 343.

Terminology

The following terms are used in this document for CCA releases and features.

CCA 4.2

CCA Release 4.2.0 and later releases

CEX2C

An IBM 4764 Crypto Express2 feature, configured in CCA coprocessor mode.

CEX3C

An IBM 4765 Crypto Express3 feature, configured in CCA coprocessor mode.

CEX4C

An IBM 4765 Crypto Express4 feature, configured in CCA coprocessor mode.

CEX3C or CEX4C

Either the CEX3C or the CEX4C.

CEX*C

Either the CEX2C, CEX3C, CEX4C, or (if plural) any combination of these.

Hardware requirements

In order to make use of the verbs provided in the Common Cryptographic Architecture (CCA) API for Linux on IBM System z, your hardware must meet certain minimum requirements.

These minimum requirements are as follows:

- IBM System z10® with the CEX3C support that became available in March 2010.
- One CEXC3 feature, with one CEXC3 adapter mapped to the z/VM® image or LPAR that uses it. The CEXC3 must have CCA 4.2.0z or greater firmware loaded (in order to have available all of the CCA 4.2.0z function). Older levels of firmware are supported with reduced function available.
- If you plan to use a Trusted Key Entry (TKE) workstation, you must have a TKE V6.0 or higher workstation in order to see supported CEXC3s. They are not seen when using TKE V5 or earlier workstations.

This is the maximum supported hardware configuration:

- IBM zEnterprise® EC12
- A maximum of eight CEX3Cs (each with two cryptographic processors), or
- A maximum of 16 CEX4Ss (each with one cryptographic processor), or
- Any combination of CEX3Cs and CEX4Cs, as long as the total does not exceed 16 processors.

This hardware configuration is also supported:
- IBM System z10 with the CEX3C support that became available in March 2010.
- One or more CEX*Cs, with CCA 4.2.0z or CCA 4.0.3z firmware loaded.
- One or more CEX2Cs, with a supported level of CCA 3.x firmware loaded. See "Legacy support" on page 767 for details.

See "Concurrent installations" on page 767 for details about a mixed environment of CEX2C and CEX*C. Note that CEX4Cs are only supported by TKE 7.2 or later if ICSF reports them as CEX4Cs. If you are running in toleration mode, and ICSF reports them as CEX3Cs, then TKE 6.0 will be able to manage them as CEX3Cs.

To determine if a card is a CEX2C, a CEX3C, or a CEX4C, see "Listing CCA coprocessors" on page 11.

In order to use the CEXC3 feature under z/VM Versions 6.1, and 5.4, you need to apply these APAR fixes:

**VM64656**
- Introduces CEXC3 support.

**VM64727**
- Fixes problem with shared coprocessors.

**VM64793**
- Introduces protected key CPACF support.

In order to use the CEX4C feature under z/VM Versions 5.4, 6.1 or 6.2, you must apply this APAR fix:

**VM65007**
- Introduces CEX4C support.

### How to use this document

An overview of the information in each section of this document.

For encryption, CCA supports Advanced Encryption Standard (AES), Data Encryption Standard (DES), public key cryptography (PKA or RSA), and Elliptic Curve Cryptography (ECC). These are very different cryptographic systems. Additionally, CCA provides APIs for generating and verifying Message Authentication Codes (MACs), Hashed Message Authentication Codes (HMACs), hashes, and PINS, as well as other cryptographic functions.

Part 1, "IBM CCA programming," on page 1 focuses on IBM CCA programming. It includes the following chapters:

- Chapter 1, "Introduction to programming for the IBM Common Cryptographic Architecture," on page 3 describes the programming considerations for using the CCA verbs. It also explains the syntax and parameter definitions used in verbs. Information about concurrency is also provided.
- Chapter 2, "Using AES, DES, and HMAC cryptography and verbs," on page 29 gives an overview of AES, DES, and HMAC cryptography, and provides general guidance information on how these verbs use different key types and key forms.
Chapter 3, “Introducing PKA cryptography and using PKA verbs,” on page 63 introduces Public Key Algorithm (PKA) support and describes programming considerations for using the CCA PKA and ECC verbs, such as the PKA key token structure and key management.

Chapter 4, “TR-31 symmetric-key management,” on page 71 introduces TR-31 support and how CCA uses an IBM-defined optional block in a TR-31 key block.

Part 2, “CCA verbs,” on page 75 focuses on CCA verbs and includes the following chapters:

- Chapter 5, “Using the CCA nodes and resource control verbs,” on page 77 describes using the CCA resource control verbs.
- Chapter 9, “Key storage mechanisms,” on page 345 describes the use of key storage, key tokens, and associated verbs.
- Chapter 6, “Managing AES and DES cryptographic keys,” on page 127 describes the verbs for generating and maintaining DES and AES cryptographic keys, the Random Number Generate verb (which generates 8-byte random numbers), the Random Number Generate Long verb (which generates up to 8192 bytes of random content), and the Secure Sockets Layer (SSL) security protocol. This chapter also describes utilities to build DES and AES tokens, generate and translate control vectors, and describes the PKA verbs that support DES and AES key distribution.
- Chapter 7, “Protecting data,” on page 285 describes the verbs for enciphering and deciphering data.
- Chapter 8, “Verifying data integrity and authenticating messages,” on page 313 describes the verbs for generating and verifying Message Authentication Codes (MACs), generating and verifying Hashed Message Authentication Codes (HMACs), generating Modification Detection Codes (MDCs), and generating hashes (SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, MD5, RIPEMD-160).
- Chapter 10, “Financial services,” on page 393 describes the verbs for use in support of finance-industry applications. This includes several categories.
  - Verbs for generating, verifying, and translating personal identification numbers (PINS).
  - Verbs that generate and verify VISA card verification values and American Express card security codes.
  - Verbs to support smart card applications using the EMV (Europay MasterCard Visa) standards.
- Chapter 11, “Using digital signatures,” on page 467 describes the verbs that support using digital signatures to authenticate messages.
- Chapter 12, “Managing PKA cryptographic keys,” on page 477 describes the verbs that generate and manage PKA keys.
- Chapter 13, “TR-31 symmetric key management verbs,” on page 523 describes the verbs used to manage TR-31 key blocks and TR-31 functions.

The appendixes include the following information:

- Appendix A, “Return codes and reason codes,” on page 593 explains the return and reason codes returned by the verbs.
- Appendix B, “Key token formats,” on page 609 describes the formats for AES, DES internal, external, and null key tokens, for PKA public, private external, and private internal key tokens containing Rivest-Shamir-Adleman (RSA) information, PKA null key tokens, ECC key tokens, HMAC key tokens, Transaction Validation Values (TVVs), and trusted blocks.
Where to find more information

In addition to the documents listed in this topic, you might want to refer to other CCA product publications.

These publications might be of use with applications and systems you might develop for use with the IBM 4765 and IBM 4764. While there is substantial commonality in the API supported by the CCA products, and while this document seeks to guide you to a common subset supported by all CCA products, other individual product publications might provide further insight into potential issues of compatibility.

**IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors**

All of the IBM 4765-related and 4764-related publications can be obtained from the Library page that you can reach from the product Web site at [http://www.ibm.com/security/cryptocards](http://www.ibm.com/security/cryptocards)

*IBM 4765 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual* and the *IBM 4764 PCI-X Cryptographic Coprocessor CCA Support Program Installation Manual*

Describe the installation of the CCA Support Program and the operation of the Cryptographic Node Management utility.

Describe the physical installation of the IBM 4765 and the IBM 4764, and also the battery-changing procedure.

Custom Programming for the IBM 4765 and the IBM 4764

The Library portion of the product Web site also includes programming information for creating applications that perform within the IBM 4765 and the IBM 4764. See the reference to custom programming. The product Web site is located at [http://www.ibm.com/security/cryptocards](http://www.ibm.com/security/cryptocards).

Other documents referenced in this book are:

- IBM CCA Basic Services Reference and Guide for the IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors
- IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference, SC40-1675
- z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User’s Guide, SA23-2211-05
- For Linux on IBM System z:
  - Device Drivers, Features, and Commands, SC33-8411

  See one of these Web Sites for the version of this book that is correct for your distribution of Linux:

Cryptography publications

The following publications describe cryptographic standards, research, and practices relevant to the coprocessor.

- American National Standards Institute (ANSI). ANSI is the official U.S. representative to the International Organization for Standardization (ISO) and, via the U.S. National Committee, the International Electrotechnical Commission (IEC). ANSI is also a member of the International Accreditation Forum (IAF).


- Federal Information Processing Standards (FIPS), issued by the U.S. National Institute of Standards and Technology (S).

- International Organization for Standardization (ISO). ISO is the world's largest developer and publisher of International Standards. ISO is a network of the national standards institutes of many countries, one member per country, with a Central Secretariat in Geneva, Switzerland, that coordinates the system.


- National Institute of Standards and Technology (NIST) Special Publications (SP), U.S. Dept. of Commerce
  - *PKCS #1 v2.0: RSA Cryptography Standard*, October 1, 1998.
  - *PKCS #1 v2.1: RSA Cryptography Standard*, June 14, 2002.
- *Visa Integrated Circuit Card Specification (VIS) 1.4.0 Corrections and Updates*, Visa International.

**Do you have problems, comments, or suggestions?**

Your suggestions and ideas can contribute to the quality and the usability of this document.

If you have problems using this document, or if you have suggestions for improving it, complete and mail the Reader's Comment Form found at the back of the document.
Part 1. IBM CCA programming

This part of the document introduces programming for the IBM CCA, AES, DES, and PKA cryptography.

The topics in this part explain how to use CCA nodes and AES, DES and PKA verbs.

- Chapter 1, “Introduction to programming for the IBM Common Cryptographic Architecture,” on page 3 describes the programming considerations for using the CCA verbs. It also explains the syntax and parameter definitions used in the verbs. Information about concurrency is also provided.

- Chapter 2, “Using AES, DES, and HMAC cryptography and verbs,” on page 29 gives an overview of AES, DES, and ECC cryptography and provides general guidance information on how these verbs use different key types and key forms.

- Chapter 3, “Introducing PKA cryptography and using PKA verbs,” on page 63 introduces Public Key Algorithm (PKA) and Elliptic Curve Cryptography (ECC) support, and describes programming considerations for using the CCA PKA verbs, such as the PKA key token structure and key management.

- Chapter 4, “TR-31 symmetric-key management,” on page 71 introduces X9 TR-31 (Technical Report 31) support, and provides details about the TR-31 key block.
Chapter 1. Introduction to programming for the IBM Common Cryptographic Architecture

This topic introduces the IBM CCA application programming interface (API).

The topic explains basic concepts and describes how you can obtain cryptographic and other services from the CEX*C feature and CCA. It includes the following topics:

- “Available Common Cryptographic Architecture verbs”
- “In-depth Common Cryptographic Architecture functional overview” on page 4
- “CPACF support” on page 12
- “Security API programming fundamentals” on page 17
- “How to compile and link CCA application programs” on page 22

Available Common Cryptographic Architecture verbs

CCA products provide a variety of cryptographic processes and data-security techniques.

Your application program can call verbs (sometimes called services) to perform the following functions:

**Data confidentiality**

Encrypt and decrypt information, typically using the AES or DES algorithms in Cipher Block Chaining (CBC) mode to enable data confidentiality.

**Data integrity**

Hash data to obtain a digest, or process the data to obtain a Message Authentication Code (MAC) or keyed hash MAC (HMAC), that is useful in demonstrating data integrity.

**Nonrepudiation**

Generate and verify digital signatures using either the RSA algorithm or the ECDSA algorithm, to demonstrate data integrity and form the basis for nonrepudiation.

**Authentication**

Generate, encrypt, translate, and verify finance industry personal identification numbers (PINs) and American Express, MasterCard, and Visa card security codes with a comprehensive set of finance-industry-specific services.

**Key management**

Manage the various AES, DES, ECC, and RSA keys necessary to perform the above operations.

**Java interaction**

Interact with the Java Native Interface (JNI). Some of the CCA verbs have a specific version that can be used for JNI work.

**CCA management**

Control the initialization and operation of CCA.
Subsequent sections group the many available verbs by topic. Each section lists the verbs in alphabetical order by verb pseudonym.

The remainder of this section provides an overview of the structure of a CCA cryptographic framework and introduces some important concepts and terms.

Common Cryptographic Architecture functional overview

This topic provides conceptual information about the framework for positioning the CCA security API.

Figure 1 provides a conceptual framework for positioning the CCA security API, which you use to access a common cryptographic architecture. Application programs make procedure calls to the CCA security API to obtain cryptographic and related I/O services. The CCA security API is designed so that a call can be issued from essentially any high-level programming language. The call, or request, is forwarded to the cryptographic services access layer and receives a synchronous response; that is, your application program loses control until the access layer returns a response after processing your request.

The products that implement the CCA security API consist of both hardware and software components.

CCA software support: The software consists of application development and runtime software components.
The application development software primarily consists of language bindings that can be included in new applications to assist in accessing services available at the API. Language bindings are provided for the C and Java programming languages.

The runtime software can be divided into the following categories:
- Service-requesting programs, including application and utility programs.
- The security API, an agent function that is logically part of the calling application program or utility.
- The cryptographic services access layer: an environment-dependent request routing function, key-storage support services, and device driver to access one or more hardware cryptographic engines.
- The cryptographic engine software that gives access to the cryptographic engine hardware.

The cryptographic engine is implemented in the hardware of the CEX*C coprocessor. Security-sensitive portions of CCA are implemented in the cryptographic engine software running in the protected coprocessor environment.

Utility programs and tools provide support for administering CCA secret keys, interacting with CCA managed symmetric and public key cryptography key storage, and configuring the software support.

You can create application programs that employ the CCA security API or you can purchase applications from IBM or other sources that use the products. This document is the primary source of information for designing systems and application programs that use the CCA security API with the cryptographic coprocessors.

**Cryptographic engine:** The CCA architecture defines a cryptographic subsystem that contains a cryptographic engine operating within a protected boundary. The coprocessor's tamper-resistant, tamper-responding environment provides physical security for this boundary and the CCA architecture provides the logical security needed for the full protection of critical information.

**CEX2C Coprocessor:** The coprocessor provides a secure programming and hardware environment wherein AES, DES and RSA processes are performed. Each cryptographic coprocessor includes a general-purpose processor, non-volatile storage, and specialized cryptographic electronics. These components are encapsulated in a protective environment to enhance security. The IBM CCA Support Program enables applications to employ a set of AES, DES and RSA-based cryptographic services utilizing the coprocessor hardware. Services include:
- DES key and RSA key-pair generation
- DES and RSA host-based key record management
- Digital signature generation and verification
- Cryptographic key wrapping and unwrapping
- Data encryption, decryption and MAC generation/verification
- PIN processing for the financial services industry
- Other services, including DES key-management based on CCA's control-vector-enforced key separation

**CEX*C Coprocessor:** The coprocessor provides a secure programming and hardware environment wherein AES, DES, RSA, Elliptic Curve, and HMAC processes are performed. Each cryptographic coprocessor includes a
general-purpose processor, non-volatile storage, and specialized cryptographic
electronics. These components are encapsulated in a protective environment to
enhance security. The IBM CCA Support Program enables applications to employ a
set of AES, DES, RSA, Elliptic Curve, and HMAC-based cryptographic services
utilizing the coprocessor hardware. Services include:

- DES, AES, RSA, Elliptic Curve, and HMAC key-pair generation
- DES, AES, RSA, Elliptic Curve, and HMAC host-based key record management
- Digital signature generation and verification
- Cryptographic key wrapping and unwrapping
- Data encryption, decryption and MAC generation/verification
- PIN processing for the financial services industry
- Other services, including DES key-management based on CCA’s
  control-vector-enforced key separation

**CEX4C Coprocessor:** The coprocessor provides the same cryptographic functions as
the CEX3C coprocessor.

**CCA:** Common Cryptographic Architecture (CCA) is the basis for a consistent
cryptographic product family. Applications employ the CCA security API to obtain
services from, and to manage the operation of, a cryptographic system that meets
CCA architecture specifications.

**CCA access control:** Each CCA node has an access-control system enforced by the
hardware and protected software. The robust UNIX style access controls integrated
into the Linux operating system are used to protect the integrity of the underlying
CCA hardware environment. The specialized processing environment provided by
the cryptographic engine can be kept secure because selected services are provided
only when certain requirements are met or a Trusted Key-Entry console is used to
enable access. The access-control decisions are performed within the secured
environment of the cryptographic engine and cannot be subverted by rogue code
that might run on the main computing platform.

**Coprocessor certification:** After quality checking a newly manufactured
coprocessor, IBM loads and certifies the embedded software. Following the loading
of basic, authenticated software, the coprocessor generates an RSA key-pair and
retains the private key within the cryptographic engine. The associated public key
is signed by a certification key securely held at the manufacturing facility and then
the certified device key is stored within the coprocessor. The manufacturing facility
key has itself been certified by a securely held key unique to the CEX°C product
line.

The private key within the coprocessor, known as the device private key, is
retained in the coprocessor. From this time on, if tampering is detected or if the
coprocessor batteries are removed or lose power in the absence of bus power, the
coprocessor sets all security-relevant keys and data items to zero. This process is
irreversible and results in the permanent loss of the factory-certified device key, the
device private key, and all other data stored in battery-protected memory.
Security-sensitive data stored in the coprocessor flash memory is encrypted. The
key used to encrypt such data is itself retained in the battery-protected memory.

**CCA master key:** When using the CCA architecture, working keys, including
session keys and the RSA and ECC private keys used at a node to form digital
signatures or to unwrap other keys, are generally stored outside the
cryptographic-engine protected environment. These working keys are wrapped
(DES triple-encrypted or AES encrypted) by the CCA master key. The master key is held in the clear (not enciphered) within the cryptographic engine.

The number of keys usable with a CCA subsystem is thus restricted only by the host server storage, not by the finite amount of storage within the coprocessor secure module. In addition, the working keys can be used by additional CCA cryptographic engines which have the same master key. This CCA characteristic is useful in high-availability and high-throughput environments where multiple cryptographic processors must function in parallel.

**Establishing a CCA master key:** To protect working keys, the master key must be generated and initialized in a secure manner. One method uses the internal random-number generator for the source of the master key. In this case, the master key is never external to the node as an entity and no other node has the same master key unless master-key cloning is authorized and in use (unless, out of all the possible values, another node randomly generates the same master-key data). If an uncloned coprocessor loses its master key, for example, the coprocessor detects tampering and destroys the master key; there is no way to recover the working keys that it wrapped. The number of possible values is:

- For DES and RSA master keys, $2^{168}$
- For AES and APKA master keys, $2^{256}$

Another master-key-establishment method enables authorized users to enter multiple, separate key parts into the cryptographic engine. As each part is entered, that part is XORed with the contents of the new master-key register. When all parts have been accumulated, a separate command is issued to promote the contents of the current master-key register to the old master-key register and to promote the contents of the new master-key register to the current master-key register. The length of the key parts is:

- For DES and RSA master keys, 168 bits
- For AES and APKA master keys, 256 bits

**CCA verbs:** Application and utility programs called requestors obtain service from the CCA Support Program by issuing service requests (verb calls or procedure calls) to the runtime subsystem (see Appendix H, “Sample verb call routines,” on page 743 for sample routines). To fulfill these requests, the Support Program obtains service from the coprocessor software and hardware.

The available services are collectively described as the CCA security API. All the software and hardware accessed through the CCA security API should be considered an integrated subsystem. A command processor performs the verb request within the cryptographic engine.

**Commands and access control, roles, profiles:** In order to ensure that only designated individuals (or programs) can run commands such as master-key loading, each command processor that performs sensitive processing interrogates one or more control-point values within the cryptographic engine access-control system for permission to perform the request.

The access-control system includes one or more roles. Each role defines the permissible control points for users of that role. In the System z environment, all application programs run using the permissions defined in the DEFAULT role for their domain. The DEFAULT role can only be modified using the TKE workstation.
For a description of the functions that are permitted by the default version of the DEFAULT role, see z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User’s Guide.

How application programs obtain service

Application programs and utility programs obtain services from the security product by issuing service requests to the runtime subsystem of software and hardware. Use a procedure call according to the rules of your application language.

The available services are collectively described as the security API. All the software and hardware accessed through the security API should be considered an integrated subsystem.

When the cryptographic services access layer receives requests concurrently from multiple application programs, it serializes the requests and returns a response for each request. There are other multiprocessing implications arising from the existence of a common master-key and a common key-storage facility. These topics are covered later in this book.

The way application programs and utilities are linked to the API services depends on the computing environment. In the Linux environment, the operating system dynamically links application security API requests to the subsystem shared object library code. Compile application programs that use CCA and link the compiled programs to the CCA library. The library and its default distribution location is /usr/lib64/libcsulcca.so.

Together, the security API shared library and the environment-dependent request routing mechanism act as an agent on behalf of the application and present a request to the server. Requests can be issued by one or more programs. Each request is processed by the server as a self-contained unit of work. The programming interface can be called concurrently by applications running as different processes. The security API can be used by multiple threads in a process and is thread safe.

In each server environment, a device driver provided by IBM supplies low-level control of the hardware and passes the request to the hardware device. Requests can require one or more I/O commands from the security server to the device driver and hardware.

The security server and a directory server manage key storage. Applications can store locally used cryptographic keys in a key-storage facility. This is especially useful for long-life keys. Keys stored in key storage are referenced using a key label. Before deciding whether to use the key-storage facility or to let the application retain the keys, consider system design trade-off factors, such as key backup, the impact of master-key changing, the lifetime of a key, and so forth.

Overlapped processing and load balancing

You can maximize throughput by organizing your application or applications to make multiple, overlapping calls to the CCA API.

Calls to the CCA security API are synchronous, that is, your program loses control until the verb completes. Multiple processing-threads can make concurrent calls to the API.
You can maximize throughput by organizing your application or applications to make multiple, overlapping calls to the CCA API. You can also increase throughput by employing multiple coprocessors, each with CCA. Another way to maximize throughput is to make use of the AUTOSELECT option for automatic load-balancing. See “Multi-coprocessor capabilities.”

Within the coprocessor, the CCA software is organized into multiple threads of processing. This multiprocessing design is intended to enable concurrent use of the coprocessor’s main engine, PCIe communications, DES and Secure Hash Algorithm-1 (SHA-1) engine, and modular-exponentiation engine.

**Host-side key caching**
CCA provides caching of key records obtained from key storage within the CCA host code.

Calls to the CCA security API are synchronous, that is, your program loses control until the verb completes. Multiple processing-threads can make concurrent calls to the API.

CCA provides caching of key records obtained from key storage within the CCA host code. However, the host cache is unique for each host process. If different host processes access the same key record, an update to a key record caused in one process does not affect the contents of the key cache held for other processes. Caching of key records within the key-storage system can be suppressed so all processes access the most current key-records. To suppress caching of key records, use the SET command to set the environment variable CSUCACHE to NO. If this environment variable is not set, or is set to anything other than NO, caching of key records will not be suppressed. The CSUCACHE environment variable does not impact CPACF translated key caching.

**Multi-coprocessor capabilities**
Multi-coprocessor capabilities allow you to employ more than one CCA coprocessor.

When more than one CCA coprocessor is installed, an application program can control which CCA coprocessor to use. It can explicitly select a CCA coprocessor, it can switch on the AUTOSELECT option, or it can optionally employ the default CCA coprocessor.

**AUTOSELECT option**
If switched on, the AUTOSELECT option overrides an explicit CCA coprocessor selection and default CCA coprocessor selection for all verbs (except those listed in Table 1 on page 10).

When the AUTOSELECT option is switched on, the CCA coprocessor to be used by a verb will be selected by the operating system (the Linux device driver) from the set of available CCA coprocessors, including any coprocessors loaded with CCA user defined function (UDX) code. The Linux device driver chooses a CCA coprocessor based on a policy for load balancing. This may allow higher application performance due to enhanced throughput.
To switch on the AUTOSELECT option, use Cryptographic Resource Allocate verb (CSUACRA). Alternatively, the AUTOSELECT option can be switched on at program start by setting the environment variable CSU_DEFAULT_ADAPTER to the value "DEV-ANY". For example:

```
export CSU_DEFAULT_ADAPTER=DEV-ANY
```

To switch off the AUTOSELECT option, use Cryptographic Resource De-allocate verb (CSUACRD).

**Master Key coherence for AUTOSELECT**

When using the AUTOSELECT option, all CCA-coprocessors accessible by the operating system must have the same state. In particular, they must be configured with the same Master Key as appropriate for the services in use. For example, if your application uses only DES functions and you enable AUTOSELECT, then the SYM-MK should be the same across all CCA coprocessors accessible. If you use any RSA functions from PKA verbs then the ASYM-MKs must be the same. For AES usage the AES-MKs must match, and for ECC the APKA-MK must match.

**Verbs that ignore AUTOSELECT**

The following verbs ignore the AUTOSELECT option and use the explicitly selected or default CCA-coprocessor instead. These verbs act as if the AUTOSELECT option does not exist, acting exactly as they did in prior releases in which AUTOSELECT was not present.

*Table 1. Verbs that ignore AUTOSELECT*

<table>
<thead>
<tr>
<th>Verb</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSUACFQ</td>
<td>When querying CCA coprocessor state it is important to retrieve data from the explicitly queried CCA coprocessor.</td>
</tr>
<tr>
<td>CSUARNT</td>
<td>When testing adapter health it is also important to receive explicit results.</td>
</tr>
<tr>
<td>CSNDRKL, CSNDRKD</td>
<td>Listing and managing the retained keys for a CCA coprocessor requires dealing with explicitly selected CCA coprocessors. Note that it is specifically not recommended to use any retained key functions (such as choosing to use retained keys with PKA verbs) when AUTOSELECT is active. By their very nature retained keys are tied to the specific CCA coprocessor where they were created.</td>
</tr>
<tr>
<td>CSNBMKP</td>
<td>Managing the Master Keys for a CCA coprocessor also requires explicit allocation and de-allocate to achieve expected results.</td>
</tr>
</tbody>
</table>

HOST-only verbs

Some verbs are not impacted by AUTOSELECT simply because they never use the CCA coprocessor. These verbs take actions that do not involve secrets guarded by the card Master Keys. These include:

- CSUACFV, CSNBKSI, CSNBVCV, CSNBKTB, CSNBKTB2, CSNBKTP, CSNBKTP2, CSNBHT310, CSNBHT31R, CSNBHT31P, CSNBAKRC, CSNBAKRD, CSNBKAKRL, CSNBAKRR, CSNBAKRW, CSNBAKRC, CSNBKRD, CSNBJKRL, CSNBKRR, CSNBKRW, CSNDKRC, CSNDKRD, CSNDKRL, CSNDKRR, CSNDKRW, CSNPKB, CSNPKX

CPACF using verbs

Some verbs use the CPACF according to configuration of appropriate environment variables. These verbs will ignore AUTOSELECT if told to use CPACF. These include:

- CSNBOWH, CSNBDEC, CSNBENC, CSNBSAE, CSNBSAD
Explicit CCA coprocessor selection

To explicitly select a CCA coprocessor, use the Cryptographic Resource Allocate verb (CSUACRA). This verb allocates a CCA coprocessor loaded with the CCA software. When a CCA coprocessor is allocated and the AUTOSELECT option is not on, CCA requests are routed to it until it is de-allocated. Similarly, when a CCA coprocessor is allocated and one of the verbs that ignore the AUTOSELECT option is used, then CCA requests are routed to it until it is de-allocated.

To de-allocate an allocated coprocessor, use the Cryptographic Resource De-allocate verb (CSUACRD). When a coprocessor is not allocated (either before an allocation occurs or after the cryptographic resource is de-allocated), requests are routed to the default coprocessor unless the AUTOSELECT option is on.

Note: The scope of the Cryptographic Resource Allocate and the Cryptographic Resource Deallocate verbs is to a thread. A multi-threaded application program can use all of the installed CCA coprocessors simultaneously. A program thread can use only one of the installed coprocessors at any given time, but it can switch to a different installed coprocessor as needed. To perform the switch, a program thread must deallocate an allocated cryptographic resource, if any, and then it must allocate the desired cryptographic resource. The Cryptographic Resource Allocate verb fails if a cryptographic resource is already allocated.

Listing CCA coprocessors

With the first call to CCA from a process, CCA associates coprocessor designators CRP01, CRP02, and so on with specific coprocessors. The host determines the total number of coprocessors installed through a call to the coprocessor device driver. Adding, removing, or relocating coprocessors can alter the number associated with a specific coprocessor. The host then polls each coprocessor in turn to determine which ones contain the CCA firmware. As each coprocessor is evaluated, the CCA host associates the identifiers CRP01, CRP02, and so forth to the CCA coprocessors. CCA coprocessors loaded with a UDX extension are also assigned a CRPnn identifier.

For a specific device driver, names such as these are used: CRPnn, cardnn, APnn, and so forth, where the nn values normally do not match (for example, some start with 0, others start with 1).

To determine if a card is a CEX2C, CEX3C, or CEX4C, use one of these methods:

- Invoke the Cryptographic Facility Query verb (see “Determining if a card is a CEX2C or CEX3C or a CEX4C” on page 77).
- Use the lszcrypt command (see “Confirming your cryptographic devices” on page 754).
- Use the hwtype attribute of your cryptographic devices in sysfs (see the version of Device Drivers, Features, and Commands that applies to your distribution).
- Run panel.exe -x using the panel.exe utility installed with the RPM, to get a quick summary of cards available and their status. See “The panel.exe utility” on page 773.
- Run ivpe, another utility installed with the RPM, which gives more detailed information about each card available. See Appendix L, “Utilities,” on page 773.

Note that the mapping of logical card identifiers such as CRP01 and CRP02 to physical cards in your machine is not defined. This is because the mapping can...
change depending on the machine and its configuration. If your application needs to identify specific coprocessor cards, you can do one of the following:

- Use the Cryptographic Facility Query verb (see "Cryptographic Facility Query (CSUACFQ)" on page 77) with the STATCARD or STATCRD2 rule_array keyword.
- Use the panel.exe utility program with option -x, in order to read a card's serial number (see "The panel.exe utility" on page 773).

**Default CCA coprocessor**

The selection of a default device occurs with the first CCA call to a coprocessor. When the default device is selected, it remains constant throughout the life of the thread. Changing the value of the environment variable after a thread uses a coprocessor does not affect the assignment of the default coprocessor. If a thread with an allocated coprocessor ends without first de-allocating the coprocessor, excess memory consumption results. It is not necessary to deallocate a cryptographic resource if the process itself ends; it is suggested only if individual threads end while the process continues to run.

You can alter the default designation by explicitly setting the CSU_DEFAULT_ADAPTER environment variable. This is accomplished by issuing the following command:

```
export CSU_DEFAULT_ADAPTER=CRPxx
```

Replace CRPxx with the identifier for the resource you wish to use, such as CRP02.

When cards of multiple types (CEX2C and CEX*C) are active in the same system, note the following:

- The CCA library will detect CEX2C, CEX3C, or CEXC4 adapters and intermingle them in the CRPnn adapter instance list. This is a list of all available adapters, in the order that they were discovered by the device driver.
- The default adapter will be the lowest numbered CEX4C instance found by the device driver, if a CEX4C instance is found. Otherwise it will be the lowest numbered CEX3C instance found by the device driver, if a CEX3C instance is found. Otherwise it will be the lowest numbered CEX2C instance found by the device driver.
- A user can specify the proper 'CRPnn' number to allocate and work with any card (CEX2C, CEX3C, or CEX4C).
- For a specific device driver, names such as these are used: CRPnn, cardnn, APnn, and so forth, where the nn values normally do not match (for example, some start with 0, others start with 1).

**CPACF support**

Central Processor Assist for Cryptographic Functions (CPACF) must be configured and enabled on the system before you can use it.

CPACF support has these features:

- "Environment variables that affect CPACF usage" on page 13
- "Access control points that affect CPACF protected key operations" on page 14
- "CPACF operation (protected key)" on page 14
- "CCA library CPACF preparation at startup" on page 16
Environment variables that affect CPACF usage

The CSU_HCPUACLR and CSU_HCPUAPRT environment variables control whether the CPACF is used for certain CCA functions.

These variables are overridden by the explicit use of the Cryptographic Resource Allocate (CSUACRA) and Cryptographic Resource Deallocate (CSUACRD) verbs to enable or disable these access patterns. To avoid confusion, the environment variables are given similar names to the keywords used by Cryptographic Resource Allocate (CSUACRA) and Cryptographic Resource Deallocate (CSUACRD).

**Note:** The default values listed here are valid even if these environment variables are not defined. Their settings represent default policy decisions made in the library code.

**CSU_HCPUACLR**

Use the CSU_HCPUACLR variable to allow CPACF for clear key operations and hashing algorithms.

Set CSU_HCPUACLR to '1' in a profile setup file or with this command:

```
export CSU_HCPUACLR=1
```

Setting this variable to any other value (except for the case where the variable has not been set, as noted above) results in disabling the use of the CPACF for clear key operations and hashing algorithms. The default is '1', meaning that the function is enabled.

**Affected verbs**

- MDC Generate (CSNBMDG)
- One-Way Hash (CSNBOWH)
- Symmetric Algorithm Decipher (CSNBSAD) (clear key AES)
- Symmetric Algorithm Encipher (CSNBSAE) (clear key AES)

**CSU_HCPUAPRT**

Use the CSU_HCPUAPRT variable to use CPACF for protected key (translated secure key) operations.

Set CSU_HCPUAPRT to '1' in a profile setup file or with this command:

```
export CSU_HCPUAPRT=1
```

Setting this variable to any other value (except for the case where the variable has not been set, as noted above) results in disabling the use of the CPACF for protected key (translated secure key) operations. The default is '0', meaning that the function is disabled.

**Affected verbs**

- Decipher (CSNBDEC)
- Encipher (CSNBENC)
- MAC Generate (CSNBMGN)
- MAC Verify (CSNBMVR)
Access control points that affect CPACF protected key operations

There are two access points that enable the protected key feature.

These two access points are:

**Symmetric Key Encipher/Decipher - Encrypted DES keys**

This is bit X'0295', and is set ON by default.

This ACP enables translating DES keys for use with the CPACF. Without this bit set ON, the call to the CEX*C to rewrap the key under the CPACF wrapping key will fail with a return code 8 and reason code 90, which will in turn imply disabling the use of this function by the host user. This error will not be returned to the user, instead the operation will be sent to the CEX*C. Because the default value of the bit is ON, it is assumed that the user will know that it is set OFF on purpose. A return code 8 and reason code 90 will cause no further requests to go to the CEX*C verb that translates keys, in an effort to preserve normal path performance.

**Symmetric Key Encipher/Decipher - Encrypted AES keys**

This is bit X'0296', and is set ON by default.

This ACP enables translating AES keys for use with the CPACF. Without this bit set ON, the call to the CEX*C to rewrap the key under the CPACF wrapping key will fail with a return code 8 and reason code 90, which will in turn imply disabling the use of this function by the host user. This error will not be returned to the user, instead the operation will be sent to the CEX*C. Because the default value of the bit is ON, it is assumed that the user will know that it is set OFF on purpose. A return code 8 and reason code 90 will cause no further requests to go to the CEX*C verb that translates keys, in an effort to preserve normal path performance.

CPACF operation (protected key)

These are details for Central Processor Assist for Cryptographic Functions (CPACF) usage by the host library.

Note that at system power-on, the CPACF generates a new Key Encryption Key (KEK, kek-t) for wrapping translated keys.

Figure 2 on page 15 illustrates the CPACF layer as it relates to the security access API and cryptographic engine. The CPACF exploitation layer examines commands received by the security server to see if they can be redirected to the CPACF. If so, this layer makes preparations (including translating secure keys to protected keys), and then call the CPACF directly. If all preparations and the CPACF operations are successful, the results are returned as a normal return through the security server. For any errors, the command is redirected back through the security server to the normal path, using the allocated CEX*C for the thread making the call.
Clear key or No key: For operations that do not use keys (such as hash algorithms) or operations that use keys that are not encrypted under the card master key, (called clear keys), no translation is necessary and the CPACF is used immediately.

Protected key: The device driver and the other layers are used for protected key support, for translating keys. This relationship is similar to the ‘directory server’ relationship: a translation layer invisible to the customer. After translation the ‘translated-key’ is stored in an invisible runtime cache so that the next use of the key can avoid the translation step. For protected key usage, a CEX*C feature must be available and allocated for use by the thread.

Important note about CPACF service actions and running applications

This note applies to processes using protected keys.

The CPACF is an independent hardware unit, like the CEX*C itself, and can be independently configured available or unavailable while an S/390® Linux instance is running by service technicians performing service actions. If the CPACF is cycled it will generate a new wrapping key for translated keys, invalidating all of the keys in the CCA library key translation cache. Therefore, it is never advisable to attempt such a service action while there are system instances with applications running that use the CPACF.

If such an action is undertaken, applications should be stopped and restarted so that the libcsulcca.so is unloaded from memory and reloaded. This will cause the key cache to be cycled. A more complete measure would be to reboot system images. If these precautions are disregarded and a CPACF service action is
undertaken as described, application crashes may ensue with a SIGSEGV error. This could occur due to translated keys wrapped under outdated CPACF wrapping keys being used.

A normal system-wide power cycle will cause the CPACF to generate a new wrapping key by design, however, this action also of course cycles all of the hosted system LPARs and VM system images so there is no problem; translated keys are not cached in permanent storage.

**Using keys with CPACF, protected key**

Follow the steps in this procedure to use keys with CPACF, protected key.

**Procedure**

1. An eligible CCA verb call (see lists in Access control points that affect CPACF protected key operations on page 14) specifying a key token or key identifier for a key token that is a normal internal CCA key token, called key-e here, comes into the CCA library.
2. The CCA library verifies that a CEXC is available for key translation. If not, then the standard ‘no-available-device’ error will be returned.
3. The CCA library tries to find an already translated version (key-t) that matches the key-e passed into the CCA library.
   - The user application (CCA library in this case) must cache translated key-t objects in RAM, using the key-e tokens as references.
4. If a key-t is not found for the key-e used:
   The CCA library translates the key-e to a key-t for use with the CPACF using CCA secure services, then caches the key pair.
5. At this point, either a fresh key-t has been obtained, or a key-t was found in RAM cache for the operation.
6. The CCA library directs the operation to the CPACF using the key-t.

**Results**

The `panel.exe -m` command displays all the supported CPACF functions. This is especially useful on a z/VM system, to make sure that the protected key functions are available. For details, see The panel.exe utility on page 773.

**Using keys with CPACF, clear key or no key**

Follow the steps in this procedure to use keys with CPACF, clear key or no key.

**Procedure**

1. An eligible CCA verb call (see lists in Access control points that affect CPACF protected key operations on page 14) comes into the CCA library.
2. No CEX3C/CEX4C is necessary, so no check for availability or Cryptographic Resource Allocate (CSUACRA) call will be implied.
3. The CCA library prepares an appropriate CPACF clear key (key-c) structure using the clear key passed to the CCA verb (key-v).
4. The CCA library directs the operation to the CPACF using the key-c.

**CCA library CPACF preparation at startup**

When the CCA library first starts up, it must prepare for use of the Central Processor Assist for Cryptographic Functions (CPACF) by taking the initialization steps described in this topic.
Procedure
1. Check configuration options to see if either is set to 'on', allowing some use of the CPACF. If neither is on, skip the rest of initialization
2. Check for existence and configuration of the CPACF.

Interaction between the 'default card' and use of Protected Key CPACF

While the CPACF can be used to encrypt and decrypt data in the absence of a CEX3C or CEX4C, for protected key operations a CEX3C or CEX4C is still necessary and it must be the allocated or default adapter for the thread doing the processing.

This is necessary because the users' key tokens are translated with a service only available on the CEX3C or CEX4C for use with the CPACF. Note also that for mixed CEX2C and CEX3C or CEX4C configurations, the allocated adapter for the thread must be a CEX3C or CEX4C because the service that translates the keys is not available on the CEX2C, for any CCA firmware version.

Using the AUTOSELECT option and the use of Protected Key CPACF

While the CPACF can be used to encrypt and decrypt data in the absence of a CEX3C or CEX4C, for protected key operations a CEX3C or CEX4C is still necessary, because the users' key tokens are translated with a service only available on the CEX3C or CEX4C for use with the CPACF.

Therefore, when enabling the AUTOSELECT option, all CCA coprocessor adapters available to the operating system must be CEX3C or CEX4C coprocessors. See "Multi-coprocessor capabilities" on page 9 for more information.

Security API programming fundamentals

You obtain CCA cryptographic services from the coprocessor through procedure calls to the CCA security application programming interface (API).

Most of the services provided are considered an implementation of the IBM Common Cryptographic Architecture (CCA). Most of the extensions that differ from other IBM CCA implementations are in the area of the access-control services. If your application program is used with other CCA products, compare the product literature for differences.

Your application program requests a service through the security API by using a procedure call for a verb. The term verb implies an action that an application program can initiate; other systems and publications might use the term callable service instead. The procedure call for a verb uses the standard syntax of a programming language, including the entry-point name of the verb, the parameters of the verb, and the variables for the parameters. Each verb has an entry-point name and a fixed-length parameter list.

The security API is designed for use with high-level languages, such as C, COBOL, or RPG and for low-level languages, such as assembler language. It is also designed to enable you to use the same verb entry-point names and variables in the various supported environments. Therefore, application code you write for use in one environment generally can be ported to additional environments with minimal change.
Verbs, variables, and parameters

This topic provides information about how each verb is described and provides an explanation of the characteristics of the security API.

Each verb has an entry-point name and a fixed-length parameter list. Part 2, “CCA verbs,” on page 75 describes each verb, and includes the following information for each verb:

- Pseudonym
- Entry-point name
- Description
- Format
- Parameters
- Restrictions
- Required commands
- Usage notes
- Related information
- JNI version

Pseudonym
Also known as a general-language name or verb name, this name describes the function that the verb performs, such as Key Generate.

Entry-point name
Also known as a computer-language name, this name is used in your program to call the verb. Each verb’s 7 or 8 character, entry-point name begins with one of the following prefixes:

Prefix Type of verb
CSNB Generally, the AES and DES verbs
CSND Public key cryptography verbs, including RSA and Elliptic Curve
CSUA Cryptographic-node and hardware-control verbs

The last three or four letters in the entry-point name after the prefix identify the specific verb in a group and are often the first letters of the principal words in the verb pseudonym.

When verbs are described throughout this publication, they are sometimes referred to by the pseudonym, and at other times by the pseudonym followed by the verb entry point name in parenthesis. An example of this is: Key Generate (CSNBKGN).

The verb prefixes used here are different from those used by IBM’s Integrated Cryptographic Service Facility (ICSF).

Description
The verb is described in general terms. Be sure to read the parameter descriptions because these add additional detail.

Format
The format section for each verb lists the entry-point name on the first line. This is followed by the list of parameters for the verb. You must code all the parameters, and they must be in the order listed.
entry-point name(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  parameter_5,
  parameter_6,
  ...
  parameter_n)

Parameters
All information exchanged between your application program and a verb is through the variables identified by the parameters in the procedure call. These parameters are pointers to the variables contained in application program storage that contain information to be exchanged with the verb. Each verb has a fixed-length parameter list and though all parameters are not always used by the verb, they must be included in the call.

The first four parameters are the same for all of the verbs. For a description of these parameters, see "Parameters common to all verbs" on page 20 and the individual verbs. The remaining parameters are unique for each verb. For descriptions of these parameters, see the definitions with the individual verbs.

In the description for each parameter, data flow direction and data type are indicated, as follows.

**Direction: Direction**
Type: Data type

*Direction:* The parameter descriptions use the following terms to identify the flow of information:

**Input**  The application program sends the variable to the verb (to the called routine).

**Output**  The verb returns the variable to the application program.

**Input/Output**  The application program sends the variable to the verb or the verb returns the variable to the application program, or both.

*Type:* Data identified by a verb parameter can be a single value or a one-dimensional array. If a parameter identifies an array, each data element of the array is of the same data type. If the number of elements in the array is variable, a preceding parameter identifies a variable that contains the actual number of elements in the associated array. Unless otherwise stated, a variable is a single value, not an array.

For each verb, the parameter descriptions use the following terms to describe the type of variable:

**Integer**  A CCA integer (CCAINT). On Linux on IBM System z this is defined as the system type "long". On other platforms this has been defined as a 4-byte (32-bit), signed, two's-complement binary number. (CCA for Linux on IBM System z has always defined the CCA Integer as a "long".)

**String**  A series of bytes where the sequence of the bytes must be maintained. Each byte can take on any bit configuration. The string...
consists only of the data bytes. No string terminators, field-length values, or typecasting parameters are included. Individual verbs can restrict the byte values within the string to characters or numerics.

Character data must be encoded in the native character set of the computer where the data is used. Exceptions to this rule are noted where necessary.

Array  An array of values, which can be integers or strings. Only one-dimensional arrays are permitted. For information about the parameters that use arrays, see “The rule_array and other keyword parameters” on page 21.

Restrictions  Any restrictions are noted.

Required commands  Any access control points required to use the verb are described here.

Usage notes  Usage notes about this verb are listed.

Related information  Any related information is noted.

JNI version  If the verb has a Java Native Interface version, it is described.

Commonly encountered parameters

Some parameters are common to all verbs; other parameters are used with many of the verbs and this topic describes several groups of these parameters.

- “Parameters common to all verbs”
- “The rule_array and other keyword parameters” on page 21
- “Key tokens, key labels, and key identifiers” on page 21

Parameters common to all verbs

A parameter is an address pointer to the associated variable in application program storage.

The first four parameters (return_code, reason_code, exit_data_length, and exit_data) are the same for all verbs:

**return_code**

The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes,” on page 593 lists the return codes.

**reason_code**

The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems. Appendix A, “Return codes and reason codes,” on page 593 lists the reason codes.

**exit_data_length**

A pointer to an integer value containing the length of the string (in bytes) that is returned by the exit_data value. This parameter should point to a value of zero, to ensure compatibility with any future extension or other operating environment.
exit_data
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

Restriction: The exit_data_length and exit_data variables must be declared in the parameter list. The exit_data_length parameter should be set to B’0’.

Return code and reason code overview:
The return_code variable provides a general indication of the results of verb processing and is the value your application program should generally use in determining the course of further processing.

For a list of return codes and their meanings, see “Return codes” on page 593. The reason_code variable provides more specific information about the outcome of verb processing. Reason code values generally differ between CCA product implementations. Therefore, the reason code values should generally be returned to individuals who can understand the implications in the context of your application on a specific platform.

See Appendix A, “Return codes and reason codes,” on page 593 for a complete list of all return codes and reason codes.

The rule_array and other keyword parameters
rule_array parameters and some other parameters use keywords to transfer information.

Generally, a rule_array consists of a variable number of data elements that contain keywords that direct specific details of the verb process. Almost all keywords, in a rule_array or otherwise, are eight bytes in length, and should be uppercase, left-aligned, and padded on the right with space characters. Not all implementations fold lowercase characters to uppercase so you should always code the keywords in uppercase.

The number of keywords in a rule_array is specified by a rule_array_count variable, an integer that defines the number of 8-byte elements in the array.

In some cases, a rule_array is used to convey information other than keywords between your application and the server. This is, however, an exception. For a list of key types that are passed in the rule_array keyword, see Table 3 on page 41.

Key tokens, key labels, and key identifiers
Essentially all cryptographic operations employ one or more keys. In CCA, keys are retained within a structure called a key token.

A verb parameter can point to a variable that contains a key token. Generally you do not need to be concerned with the details of a key token and can deal with it as an entity.

Key tokens are described as either internal, operational, or external, as follows:

Internal
A key token that contains an encrypted key for local use. The cryptographic engine decrypts an internal key to use the key in a local operation. When a key is entered into the system, it is always encrypted if it appears outside the protected environment of the cryptographic engine.
The engine has a special key-encrypting key designated as a master key. This key is held within the engine to wrap and unwrap locally used keys.

**Operational**

An internal key token that is complete and ready for use and contains a key that is encrypted under a master key. During entry of a key, the internal key-token can have a flag set indicating the key information is incomplete.

**External**

A key token that contains a key that is either in the clear or is encrypted by some key-encrypting key other than the master key. Generally, when a key is to be transported from place to place or is to be held for a significant period of time, the key must be encrypted with a transport key. A key wrapped by a (transport) key-encrypting key is designated as being external.

RSA and ECC public-keys are not encrypted values and, when not accompanied by private-key information, are retained in an external key-token.

Internal key tokens can be stored in a file maintained by the directory server. These key tokens are referenced by use of a key label. A *key label* is an alphanumeric string you place in a variable and reference with a verb parameter.

Verb descriptions specify how you can provide a key using these terms:

**Key token**

The variable must contain a proper key-token structure.

**Key label**

The variable must contain a key-label string used to locate a key record in key storage.

**Key identifier**

The variable must contain either a key token or a key label. The first byte in the variable indicates whether the variable contains a key token or a key label. When the first byte is in the range X'20' - X'FE', the variable is processed as a key label. There are additional restrictions on the value of a key label. The first byte in all key-token structures is in the range of X'01' - X'1F'. The value X'00' indicates a DES null key-token. The value X'FF' as the first byte of a key-related variable passed to the API raises an error condition.

---

**How to compile and link CCA application programs**

The CCA RPM includes C libraries that you can use to build CCA C applications.

One of these libraries also supports a Java Native Interface (JNI) that you can use to build CCA Java applications. The CCA RPM also includes Java libraries with front end classes for the JNI.

The C libraries and their default distribution locations are:

`/usr/lib64/libcsulcca.so.*`

This library contains all CCA verbs apart from the Master Key Process (CSNBMKP) verb. This library also contains the C support for the JNI.

`/usr/lib64/libcsulccamk.so.*`

This library contains the Master Key Process (CSNBMKP) verb and is
required for applications that use the master key process (see “Using the Master Key Process (CSNBMKP) verb”).

CCA 4.2 includes two versions of the Java libraries:

**New version introduced with CCA 4.2**

As of CCA 4.2, new Java libraries that use the Java package infrastructure are included. These Java libraries and their default distribution locations are:

- `/opt/IBM/CCA/cnm/CNM.jar`
  - This library contains the data classes used by the JNI (see “Entry points and data types used in the JNI” on page 24) and most JNI verb front end classes.

- `/opt/IBM/CCA/cnm/CNMMK.jar`
  - This library contains the JNI front end class for the Master Key Process (CSNBMKP) verb.

**Deprecated version**

CCA 4.2 still includes the Java libraries of earlier CCA versions to support your existing applications. These libraries contain all verbs available with CCA 4.2 but future additions will not be available in these libraries. The default distribution locations of these deprecated libraries are:

- `/opt/IBM/CCA/cnm/HIKM.zip`
  - This library contains the data classes used by the JNI (see “Entry points and data types used in the JNI” on page 24) and most JNI verb front end classes.

- `/opt/IBM/CCA/cnm/HIKMMK.zip`
  - This library contains the JNI front end class for the Master Key Process (CSNBMKP) verb.

See “Methods for calling the CCA JNI” on page 24 for more information about the two versions of the Java libraries.

The CCA RPM also includes C and Java sample programs that help you develop your application (see “Building C applications to use with the CCA libraries” on page 24 and “JNI sample modules and sample code” on page 25).

**Using the Master Key Process (CSNBMKP) verb**

`/usr/lib64/libcsulccamk.so` contains the Master Key Process (CSNBMKP) verb.

Any use of the `libcsulccamk.so` library is restricted because the library is installed so that only the root user (user ID of 0) and members of the group `cca_admin` have read access. The `cca_admin` group is added by the CCA RPM installation procedure. This is done to limit the ability of an untrusted user to copy the library with the purpose of reverse-engineering the master-key access methods inside it.

Furthermore, use of some specific access methods through the Master Key Process (CSNBMKP) verb are restricted to corresponding Linux group membership of the user trying to make that access. Table 208 on page 764 contains a list of the groups and their functions.

Users without the required group membership are denied use. For more information, see Master key load (Step 7 on page 762).
Building C applications to use with the CCA libraries

Perform the steps described in this topic to build a C program from a make file, `<makefile>`.

**Procedure**

1. Change to the directory that contains the make file.
2. Issue this command to compile the program:
   
   ```
   make -f <makefile>
   ```
   
   For example, to use the make file of “Sample program in C” on page 743, issue:
   
   ```
   make -f makefile.lnx
   ```

Using the CCA JNI

This topic explains how to use the CCA JNI and provides some background information that you must be aware of when working with the JNI.

**Methods for calling the CCA JNI**

For CCA 4.2, there are two methods for calling the CCA JNI.

See “JNI sample modules and sample code” on page 25 about sample programs that contrast the two methods.

**Method using the Java package infrastructure:**

This method was introduced with CCA 4.2.

When using this method, you need import statements in the Java source files. The new JNI JAR files used by this method are CNM.jar and, for the master key process, CNMMK.jar.

This is the preferred method to use when developing new applications.

See “Building the Java Byte code” on page 26 for details about compiling and running Java applications with this method.

**Deprecated method used with prior CCA versions:**

With CCA versions before 4.2, the CCA JNI was called using the HIKM.zip JAR file and, for the master key process, the HIKMMK.zip JAR file.

To support existing applications, these JAR files are included in CCA 4.2, and they provide access to all CCA 4.2 functions.

This method is deprecated. Do not use it when developing new applications.

See “Building Java applications to use with the CCA JNI (deprecated method)” on page 27 for details about compiling and running Java applications with this method.

**Entry points and data types used in the JNI**

The Java entry points of CCA verbs are similar to the C entry points, except that a letter J is appended to the entry point name.
For example, CSNBKGN is the C entry point for the Key Generate verb, and CSNBKGNJ is the Java entry point for this verb. The detailed verb descriptions in Part 2, “CCA verbs,” on page 75 include a section for the JNI interface.

These two data types are defined and used in the JNI:

**hikmNativeInteger**
64-bit native signed integer (type long), matching the C interface

**Byte **
General pointer type to unsigned byte

A file named hikmNativeInteger.html provides information about this class. The default location of this file is /opt/IBM/CCA/doc directory.

### JNI Sample Modules and Sample Code

To illustrate the two JNI access methods and also how to use the CCA JNI to call CCA verbs, sample modules are provided.

**mac.java**
Illustrates JNI calls versus C calls to CCA.

This sample program calls the same CCA verbs as the sample C language program named mac.c.

This sample program uses the package infrastructure of the new JNI access method. For more information about this sample program, see “Sample Program in Java” on page 747.

**RNGpk.java and RNG.java**
Illustrate the two methods available for calling the JNI.

The RNGpk.java sample file is different from RNG.java in how the CCA JNI is invoked. Take note of these differences when building your application:

- Java class implementations (*.java files) that call CCA JNI functions need an import line, such as this:
  ```
  import com.ibm.crypto.cca.jni.*/
  ```
- The Java classpath must point to CNM.jar.

The default location of the sample code is /opt/IBM/CCA/samples for both, SUSE and Red Hat distributions.

### Preparing Your Java Environment

Before you can compile and run Java applications that use the CCA JNI, you must install a Java version that is supported by your distribution.

Install Java from the distribution installation media or from other authorized sources for that distribution.

The CCA JNI has been tested with these Java versions:

- Java 1.6.0 for Red Hat Enterprise Linux 6
- Java 1.4.2 for SUSE Linux Enterprise Server 11

Later versions of Java, as provided with a distribution, might work, although they have not been tested.

For compiling and running applications that use the CCA JNI, you need access to the java and javac executables. Use one of the following methods to ensure that you can call the command without preconditions:
• Add the path to the java and javac executables to your PATH environment variable.

• Create soft links from the java and javac executables from wherever they are located to a directory that is in your PATH environment variable by default, such as /usr/bin.

Building Java applications to use with the CCA JNI

This topic describes how to call the CCA JNI using the Java package infrastructure.

This method was introduced with CCA 4.2 and is the preferred method for new applications.

See “Building Java applications to use with the CCA JNI (deprecated method)” on page 27 about working with Java applications that use the prior call method.

Building the Java Byte code:

Perform these steps to compile a Java source file.

Procedure

1. Ensure that your LD_LIBRARY_PATH variable points to the CCA libraries. This example points to the default location:
   
   export LD_LIBRARY_PATH=/usr/11b64

2. Change to the directory that contains the source code file of the program you want to compile.

3. Issue this command to compile a Java source code file <program>.java:
   
   javac -classpath <fullpath>/CNM.jar <program>.java
   
   where <fullpath> is the location of CNM.jar.

   The following example uses the default path and the sample program of “Sample program in Java” on page 747:

   javac -classpath /opt/IBM/CCA/cnm/CNM.jar RNGpk.java

   If the program uses the Master Key Process (CSNBMKP) verb, you must also include CNMMK.jar in the class path (see “Using the Master Key Process (CSNBMKP) verb” on page 23). Issue this command to compile such programs:

   javac -classpath <fullpath>/CNM.jar:<fullpath>/CNMMK.jar <program>.java

   Tip: Instead of using the -classpath option, you can set the CLASSPATH variable to point to CNM.jar and, if required, CNMMK.jar.

Running the Java Byte code:

Perform these steps to run a compiled Java program.

Procedure

1. Change to the directory that contains the compiled Java program you want to run.

2. Issue this command to run a program <program>.class:
   
   java -classpath <fullpath>/CNM.jar:. <program>.class

   where <fullpath> is the location of CNM.jar. The period (.) at the end of the class path ensures that Java can find <program>.class in the current directory.

   The following example uses the default path and the compiled sample program of “Sample program in Java” on page 747
Building Java applications to use with the CCA JNI (deprecated method)

The method for calling the CCA JNI described in this section is deprecated. This method is provided only to support applications that were written for versions earlier than CCA 4.2.

See "Building Java applications to use with the CCA JNI" on page 26 for the method to use with new applications.

Building the Java Byte code:

Perform these steps to compile a Java source file.

Procedure
1. Change to the directory that contains the source code file of the program you want to compile.
2. Issue the following command to compile a program <program>.java:
   
   ```
   javac -classpath <fullpath>/CNM.jar <program>.java
   ```

   where <fullpath> is the location of HIKM.zip.

   The following example uses the default path and the sample program of "Sample program in Java" on page 747:

   ```
   javac -classpath /opt/IBM/CCA/cnm/HIKM.zip RNG.java
   ```

   If the program uses the Master Key Process (CSNBMKP) verb in addition to other verbs, you must also include HIKM.zip in the class path (see "Using the Master Key Process (CSNBMKP) verb" on page 23).

   3. Issue this command to compile such programs:

   ```
   javac -classpath <fullpath>/HIKM.jar:<fullpath>/HIKMMK.jar:. <program>.java
   ```

   Tip: Instead of using the -classpath option, you can set the CLASSPATH variable to point to HIKM.jar and, if required, HIKMMK.jar.

Running the Java Byte code:

Perform these steps to run a compiled Java program.

Procedure
1. Change to the directory that contains the compiled Java program you want to run.
2. Issue this command to run the program:

   ```
   java -classpath <fullpath>/HIKM.zip:. <program>.class
   ```
where `<fullpath>` is the location of HIKM.zip. The period (.) at the end of the class path ensures that Java can find `<program>.class` in the current directory.

The following example uses the default path and the compiled sample program of "Sample program in Java" on page 747:

```bash
javac -classpath /opt/IBM/CCA/crm/HIKM.zip:. RNG.class
```

If the program uses the Master Key Process (CSNBMKP) verb in addition to other verbs, you must also include HIKMMK.zip in the class path (see "Using the Master Key Process (CSNBMKP) verb" on page 23). Issue this command to compile such programs:

```bash
java -classpath <fullpath>/HIKM.zip:<fullpath>/HIKMMK.zip <program>.class
```

**Tip:** Instead of using the `-classpath` option, you can set the CLASSPATH variable to point to HIKM.zip and, if required, HIKMMK.zip.
Chapter 2. Using AES, DES, and HMAC cryptography and verbs

This topic provides an overview of the AES, DES, and HMAC cryptographic functions provided by CCA, explains the functions of the cryptographic keys, and introduces the topic of building key tokens.

The CEX4C protects data from unauthorized disclosure or modification. This coprocessor protects data stored within a system, stored in a file off a system on magnetic tape, and sent between systems. The coprocessor also authenticates the identity of customers in the financial industry and authenticates messages from originator to receiver. The coprocessor uses cryptography to perform these functions.

The CCA API for the coprocessor provides access to cryptographic functions through verbs. A verb is a routine that receives control using a function call from an application program. Each verb performs one or more cryptographic functions, including:

- Generating and managing cryptographic keys
- Enciphering and deciphering data with encrypted keys using either the U.S. National Institute of Standards and Technology (NIST) Data Encryption Standard (DES) or Advanced Encryption Standard (AES)
- Re-enciphering text from encryption under one key to encryption under another key
- Encoding and decoding data with clear keys
- Generating random numbers
- Ensuring data integrity and verifying message authentication
- Generating, verifying, and translating personal identification numbers (PINs) that identify a customer on a financial system

Functions of the AES, DES and HMAC cryptographic keys

The CCA API provides functions to create, import, and export AES, DES and HMAC keys and this topic gives an overview of these cryptographic keys.

Key separation

The cryptographic coprocessor controls the use of keys by separating them into unique types, allowing you to use a specific type of key only for its intended purpose.

For example, a key used to protect data cannot be used to protect a key.

A CCA system has only one DES or AES master key. However, to provide for key separation, the cryptographic coprocessor automatically encrypts each type of key in a fixed-length token under a unique variation of the master key. Each variation of the master key encrypts a different type of key. Although you enter only one master key, you have a unique master key to encrypt all other keys of a certain type.
Master key variant for fixed-length tokens

Whenever the master key is used to encipher a key, the cryptographic coprocessor produces a variation of the master key according to the type of key that the master key will encipher.

These variations are called **master key variants**. The cryptographic coprocessor creates a master key variant by XORing a fixed pattern, called a **control vector**, onto the master key. A unique control vector is associated with each type of key. For example, all the different types of data-encrypting, PIN, MAC, and transport keys each use a unique control vector which is XORed with the master key in order to produce the variant. The different key types are described in "Types of keys" on page 36.

Each master key variant protects a different type of key. It is similar to having a unique master key protect all the keys of a certain type.

The master key, in the form of master key variants, protects keys operating on the system. A key can be used in a cryptographic function only when it is enciphered under a master key. When systems want to share keys, transport keys are used to protect keys sent outside of systems. When a key is enciphered under a transport key, the key cannot be used in a cryptographic function. It must first be brought on to a system and enciphered under the system's master key, or exported to another system where it will then be enciphered under that system's master key.

Transport key variant for fixed-length tokens

Like the master key, the coprocessor creates variations of a transport key to encrypt a key according to its type.

This allows for key separation when a key is transported off the system. A **transport key variant**, also called **key-encrypting key variant**, is created the same way a master key variant is created. The transport key’s clear value is XORed with a control vector associated with the key type of the key it protects.

**Note**: To exchange keys with systems that do not recognize transport key variants, the coprocessor allows you to encrypt selected keys under a transport key itself, not under the transport key variant. For more information, see NOCV Importers and Exporters on page 39.

Key forms

A key that is protected under the master key is in **operational form**, which means the coprocessor can use it in cryptographic functions on the system.

When you store a key with a file or send it to another system, the key is enciphered under a transport key rather than the master key. The transport key is a key shared by your system and another system for the purpose of securely exchanging other keys. When CCA enciphers a key under a transport key, the key is not in operational form and cannot be used to perform cryptographic functions.

When a key is enciphered under a transport key, the sending system considers the key in **exportable form**. The receiving system considers the key in **importable form**. When a key is re-enciphered from under a transport key to under a system's master key, it is in operational form again.
Enciphered keys appear in three forms. The form you need depends on how and when you use a key.

- **Operational** key form is used at the local system. Many verbs can *use* an operational key form.
  The Key Generate, Key Import, Data Key Import, Clear Key Import, and Multiple Clear Key Import verbs can *create* an operational key form.

- **Exportable** key form is transported to another cryptographic system. It can be passed only to another system. The CCA verbs cannot use it for cryptographic functions. The Key Generate, Data Key Export, and Key Export verbs produce the exportable key form.

- **Importable** key form can be transformed into operational form on the local system. The Key Import verb (CSNBKIM) and the Data Key Import verb (CSNBDKM) can *use* an importable key form. Only the Key Generate verb (CSNBKGN) can *create* an importable key form.

For more information about the key types, see “Functions of the AES, DES and HMAC cryptographic keys” on page 29. See Appendix C, “Key forms and types used in the Key Generate verb,” on page 663 for more information about key form.

**Symmetric key (DES, AES) flow**
The conversion from one key to another key is considered to be a one-way flow.

An operational key form cannot be turned back into an importable key form. An exportable key form cannot be turned back into an operational or importable key form. The flow of CCA key forms can be in only one direction:

- IMPORTABLE —to→ OPERATIONAL —to→ EXPORTABLE

**Key token**
CCA supports two types of symmetric key tokens, fixed-length and variable-length.

An AES or DES fixed-length token is a 64-byte field composed of a key value and control information in the control vector. An HMAC key token is a variable-length token composed of a key value and control information. The control information is assigned to the key when the coprocessor creates the key. The key token can be either an internal key token, an external key token, or a null key token. Through the use of key tokens, CCA can do the following:

- Support continuous operation across a master key change
- Control use of keys in cryptographic services

If the first byte of the key identifier is X’01’, the key identifier is interpreted as an internal key token. An internal key token is a token that can be used only on the CCA system that created it or another CCA system with the same host master key. It contains a key that is encrypted under the master key.

An application obtains an internal key token by using one of the verbs such as those listed below. The verbs are described in detail in Chapter 6, “Managing AES and DES cryptographic keys,” on page 127.

- AES Key Record Read
- Clear Key Import
- Data Key Import
- DES Key Record Read
- Key Generate
The master key could be dynamically changed between the time that you invoke a verb, such as the Key Import verb, to obtain a key token, and the time that you pass the key token to the Encipher verb. When a change to the master key occurs, the coprocessor will still successfully use the key, because it stores a copy of the old master key as well as the new one.

Attention: If an internal key token held in user storage is not used while the master key is changed twice, the internal key token is no longer usable. A return code of 0 with a reason code of 10001 notifies you that the master key used to decrypt the key used in your operation was an old master key, as a reminder that you should use one of the Key Token Change verbs to re-encipher your key under the current or new master key (as desired, see verbs for description).

For debugging information, see Appendix B, “Key token formats,” on page 609 for the format of an internal key token.

If the first byte of the key identifier is X'02', the key identifier is interpreted as an external key token. By using the external key token, you can exchange keys between systems. It contains a key that is encrypted under a key-encrypting key.

An external key token contains an encrypted key and control information to allow compatible cryptographic systems to:
- Have a standard method of exchanging keys
- Control the use of keys through the control vector
- Merge the key with other information needed to use the key

An application obtains the external key token by using one of the verbs such as those listed below. They are described in detail in Chapter 6, “Managing AES and DES cryptographic keys,” on page 127.
- Key Generate
- Key Export
- Data Key Export
- Symmetric Key Export

For debugging information, see Appendix B, “Key token formats,” on page 609 for the format of an external key token.

If the first byte of the key identifier is X'00', the key identifier is interpreted as a null key token. Use the null key token to import a key from a system that cannot produce external key tokens. That is, if you have an 8 or 16-byte key that has been encrypted under an importer key, but is not imbedded within a token, place the encrypted key in a null key token and then invoke the Key Import verb to get the key in operational form.
Key wrapping

This topic explains how symmetric keys are wrapped with master and key-encrypting keys.

CCA supports two methods of wrapping the key value in a fixed-length symmetric key token for DES and AES keys: the original ECB wrapping and an enhanced CBC wrapping method which is ANSI X9.24 compliant. These methods use the 64-byte token. HMAC keys use a variable length token with associated data and the payload wrapping method. Variable-length tokens are wrapped using the AESKW wrapping method defined in ANSI X9.102.

AES key wrapping

The key value in AES tokens are wrapped using the AES algorithm and cipher block chaining (CBC) mode of encryption.

The key value is left-aligned in a 32-byte block, padded on the right with zero, and encrypted.

The enhanced wrapping of an AES key (*K) using an AES *MK is defined as:

\[ e^{MK}(K) = e^{cbcMK}(K) \]

DES key wrapping

The key value in a DES key token are wrapped using one of two possible methods.

The two methods are:

Original method

The key value in DES tokens are encrypted using triple-DES encryption, and key parts are encrypted separately. See "ECB wrapping of DES keys (Original method)."

Enhanced method

The key value for keys is bundled with other token data and encrypted using triple-DES encryption and cipher block chaining mode. The enhanced method applies only to DES key tokens. The enhanced method of symmetric key wrapping is designed to be ANSI X9.24 compliant. This method was introduced with CCA 4.1.0. See "Enhanced CBC wrapping of DES keys (Enhanced method)" on page 34.

ECB wrapping of DES keys (Original method)

How to wrap a double length key (*K) using a double-length key-encrypting key (*KEK) is described in this topic.

The definition is as follows

\[ e^{KEK}(KL) || e^{KEK}(KR) = e^{KEKL}(d^{KEKR}(e^{KEKL}(KL))) || e^{KEKL}(d^{KEKR}(e^{KEKL}(KR))) \]

Where:

KL Is the left 64 bits of *K
KR Is the right 64 bits of *K
KEKL Is the left 64 bits of *KEK
KEKR Is the right 64 bits of *KEK
**Enhanced CBC wrapping of DES keys (Enhanced method)**

The enhanced wrapping of a double length key (*K) using a double-length key-encrypting key (*KEK) is described in this topic.

The enhanced CBC wrapping method uses triple-DES encryption, an internal chaining of the key value, and CBC mode. This method was introduced with CCA 4.1.0.

The enhanced wrapping of a double-length key (*K) using a double-length key-encrypting key (*KEK) is defined as:

\[ e^{*KEK(*KL)} = ecbcKEKL(dcbcKEKR(ecbcKEKL(KL' || KR))) \]

\[ KL' = KL \oplus SHA1(KR) \]

Where:

- **KL** Is the left 64 bits of *K
- **KR** Is the right 64 bits of *K
- **KL'** Is the 64-bit modified value of KL
- **KEK** Is the left 64 bits of *KEK
- **KEKR** Is the right 64 bits of *KEK
- **SHA1(X)** Is the 160-bit SHA-1 hash of X

**Wrapping key derivation for enhanced wrapping of DES keys:**

The wrapping key is exactly the same key that is used by the legacy wrapping method (the only method used by CCA 4.0.0), with one exception.

Instead of using the base key itself (master key or key-encrypting key), a key that is derived from that base key is used. The derived key will have the control vector applied to it in the standard CCA manner, and then use the resulting key to wrap the new-format target key token.

The reason for using a derived key is to ensure that no attacks against this wrapping scheme are possible using the existing CCA functions. For example, it was observed that an attack was possible by copying the wrapped key into an ECB CCA key token, if the wrapping key was used instead of a derivative of that key.

The key will be derived using a method defined in the U.S. National Institute of Standards and Technology (NIST) standard SP 800-108, *Recommendation for Key Derivation Using Pseudorandom Functions* (October, 2009). Derivation will use the method *KDF in Counter Mode* using pseudorandom function (PRF) HMAC-SHA256. This method provides sufficient strength for deriving keys for any algorithm used.

The HMAC algorithm is defined as:
HMAC(K, text) = H((K0 XOR opad) || H((K0 XOR ipad) || text))

Where:

H   Is an approved hash function.
K   Is a secret key shared between the originator and the intended receivers.
K0  The key K after any necessary preprocessing to form a key of the proper length.
ipad Is the constant X'36' repeated to form a string the same length as K0
opad Is the constant X'5C' repeated to form a string the same length as K0
text Is the text to be hashed.
I I  Means concatenation
XOR  Means bitwise exclusive OR

If the key K is equal in length to the input block size of the hash function (512 bits for SHA-256), K0 is set to the value of K. Otherwise, K0 is formed from K by hashing or padding.

The Key Derivation Function (KDF) specification calls for inputs optionally including two byte strings, Label and Context. The Context will not be used. The Label will contain information on the usage of this key, to distinguish it from other derivations that CCA may use in the future for different purposes. Because the security of the derivation process is rooted in the security of the derivation key and in the HMAC and Key Derivation Functions (KDF) themselves, it is not necessary for this label string to be of any particular minimum size. The separation indicator byte of X'00' specified in the NIST document will follow the label.

The label value will be defined so that it is unique to derivation for this key wrapping process. This means that any future designs that use the same KDF must use a different value for the label. The label will be the 16 byte value consisting of the following ASCII characters:
ENHANCEDWRAP2010 (X'454E4841 4E434544 57524150 32303130'

The parameters for the counter mode KDF defined in NIST standard SP 800-108 are:

Fixed values:

<table>
<thead>
<tr>
<th>h</th>
<th>Length of output of PRF, 256 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Length of the counter, in bits, 32. The counter will be an unsigned 4-byte value.</td>
</tr>
</tbody>
</table>

Inputs:

- KI (input key) - The key we are deriving from.
- Label - The value shown above (ASCII ENHANCEDWRAP2010).
- Separator byte - X'00' following the label value.
- Context - A null string. No context is used.
- L - The length of the derived key to be produced, rounded up to the next multiple of 256.
- PRF - HMAC-SHA256.
Variable length token (AESKW method)
The wrapping method for the variable-length key tokens with AESKW is defined in standard ANSI X9.102.

The wrapping of the payload of a variable length key (*K) using an AES *MK is defined as:
\[ e^{*MK}(*K) = e^{AESKW}^{*MK}(P) \]
\[ P = ICV \| Pad\ length \| Hash\ length \| Hash\ options \| Data\ hash \| *K \| Padding \]

Where:
- **ICV** is the 6 byte constant X'A6A6A6A6A6A6'.
- **Pad length** is the length of the padding in bits.
- **Hash length** is the length of the Data Hash in bytes.
- **Hash options** is a 4-byte field.
- **Data hash** is the hash of the associated data block.
- **Padding** is the number of bytes of X'00' used to make the overall length of P a multiple of 16.
- **eAESKW** means encryption using the AESKW method.

Control vector
A unique control vector exists for each type of CCA key.

For an internal key token, the coprocessor XORs the master key with the control vector associated with the type of key the master key will encipher. The control vector ensures that an operational key is used only in cryptographic functions for which it is intended. For example, the control vector for an input PIN-encrypting key ensures that such a key can be used only in the Encrypted PIN Translate and Encrypted PIN Verify functions.

Types of keys
The cryptographic keys are grouped into the following categories based on the functions that they perform.

**Symmetric keys master key (SYM-MK)**
The SYM-MK master key is a triple-length (192-bit) key that is used only to encrypt other DES keys on the coprocessor. The administrator installs and changes the SYM-MK master key using the panel.exe utility, the clear key entry panels, the z/OS® clear key entry panels, or the optional Trusted Key Entry (TKE) workstation. The master key always remains within the secure boundary of the coprocessor. It is used only to encipher and decipher keys that are in operational form.

For details about panel.exe, see "The panel.exe utility" on page 773.

**Note:** If the coprocessor is shared with z/OS, the SYM-MK key must be a double-length (128-bit) key. This means that the first 64 bits and the last 64...
bits of the key must be identical. If the master key is loaded by z/OS CCA or from a TKE workstation, it will automatically be a double-length key.

**AES keys master key (AES-MK)**

The AES-MK master key is a 256-bit key that is used to encrypt other AES keys and HMAC keys on the coprocessor. The administrator installs and changes the AES-MK master key using the panel.exe utility, the clear key entry panels, the z/OS clear key entry panels, or the optional Trusted Key Entry (TKE) workstation. The master key always remains within the secure boundary of the coprocessor. It is used only to encipher and decipher keys that are in operational form.

For details about panel.exe, see "The panel.exe utility" on page 773.

**Asymmetric keys master key (ASYM-MK)**

The ASYM-MK is a triple-length (192-bit) key that is used to protect RSA private keys on the coprocessor. The administrator installs and changes the ASYM-MK master key using the panel.exe utility, the clear key entry panels, the z/OS clear key entry panels, or the optional Trusted Key Entry (TKE) workstation. The master key always remains within the secure boundary of the coprocessor. It is used only to encipher and decipher keys that are in operational form.

For details about panel.exe, see "The panel.exe utility" on page 773.

**AES CIPHER keys**

The AES cipher keys are 128-bit, 192-bit, and 256-bit keys that protect data privacy. If you intend to use a cipher key for an extended period, you can store it in key storage so that it will be reenciphered if the master key is changed.

**AES PKA master key (APKA-MK)**

The APKA-MK key, introduced to CCA beginning with Release 4.1.0, is used to encrypt and decrypt the Object Protection Key (OPK) that is itself used to wrap the key material of an Elliptic Curve Cryptography (ECC) key. ECC keys are asymmetric. The APKA-MK is a 256-bit (32-byte) value. The administrator installs and changes the APKA-MK master key using the panel.exe utility, the clear key entry panels, the z/OS clear key entry panels, or the optional Trusted Key Entry (TKE) workstation.

**Data-encrypting keys**

The data-encrypting keys are single-length DES (64-bit), double-length DES (128-bit), or triple-length DES (192-bit) keys, or 128-bit, 192-bit or 256-bit AES keys that protect data privacy. Single-length DES data-encrypting keys can also be used to encode and decode data and authenticate data sent in messages. If you intend to use a data-encrypting key for an extended period of time, you can store it in the CCA key storage file so that it will be re-enciphered if the master key is changed.

You can use single-length DES data-encrypting keys in the Encipher and Decipher verbs to manage data, and also in the MAC Generate and MAC Verify verbs. Double-length DES and triple-length DES data-encrypting keys can be used in the Encipher and Decipher verbs for more secure data privacy. DATAC is also a double-length DES data encrypting key.

AES data-encrypting keys can be used in services similar to DES data-encrypting key services.

**DES CIPHER keys**

These consist of CIPHER, ENCIPHER, and DECIPHER keys. They are single and double length DES keys for enciphering and deciphering data.
**HMAC keys**

HMAC keys are variable-length symmetric keys. The length is in the range of 80 - 2024. HMAC keys are used to generate and verify HMACs using the FIPS-198 algorithm, with the HMAC Generate and HMAC Verify verbs.

- Operational keys will be encrypted under the AES master key
- HMAC keys can be imported and exported under an RSA key.
- HMAC keys will be stored in the AES key storage file. The AES master key must be active.

For more information about HMAC keys and verb processing, see Chapter 8, “Verifying data integrity and authenticating messages,” on page 313.

**MAC keys**

The MAC keys are single-length DES (64-bits - DATAM, DATAMV, MAC, and MACVER, ) and double-length DES (128-bits - DATAM, DATAMV, MAC, and MACVER) keys used for the verbs that generate and verify MACs.

**PIN keys**

The personal identification number (PIN) is a basis for verifying the identity of a customer across financial industry networks. PIN keys are used in cryptographic functions to generate, translate, and verify PINs, and protect PIN blocks. They are all double-length DES (128 bits) keys. PIN keys are used in the Clear PIN Generate, Encrypted PIN Verify, and Encrypted PIN Translate verbs.

For installations that do not support double-length DES 128-bit keys, effective single-length DES keys are provided. For a single-length DES key, the left key half of the key equals the right key half.

“Processing personal identification numbers” on page 49 gives an overview of the PIN algorithms you need to know to write your own application programs.

**AES transport keys (or key-encrypting keys)**

Transport keys are also known as key-encrypting keys. They are used to protect AES and HMAC keys when you distribute them from one system to another.

There are two types of AES transport keys:

- **Exporter key-encrypting key**
  
  This type of key protects keys of any type that are sent from your system to another system. The exporter key at the originator is the same key as the importer key of the receiver.

- **Importer key-encrypting key**
  
  This type of key protects keys of any type that are sent from another system to your system. It also protects keys that you store externally in a file that you can import to your system at another time. The importer key at the receiver is the same key as the exporter key at the originator.

**DES transport keys (or key-encrypting keys)**

Transport keys are also known as key-encrypting keys. They are double-length (128 bits) DES keys used to protect keys when you distribute them from one system to another.

There are several types of DES transport keys:
**Exporter or OKEYXLAT key-encrypting key**
This type of key protects keys of any type that are sent from your system to another system. The exporter key at the originator is the same key as the importer key of the receiver.

**Importer or IKEYXLAT key-encrypting key**
This type of key protects keys of any type that are sent from another system to your system. It also protects keys that you store externally in a file that you can import to your system later. The importer key at the receiver is the same key as the exporter key at the originator.

**NOCV importers and exporters**
These keys are key-encrypting keys used to exchange keys with systems that do not recognize key-encrypting key variants. There are some requirements and restrictions for the use of NOCV key-encrypting keys:

- The use of NOCV IMPORTERs and EXPORTERs is controlled by access control points in the coprocessor's role-based access control system.
- Only programs in system or supervisor state can use the NOCV key-encrypting key in the form of tokens in verbs. Any program can use NOCV key-encrypting keys with label names from the key storage.
- Access to NOCV key-encrypting keys should be carefully controlled, because use of these keys can reduce security in your key management process.
- NOCV key-encrypting key can be used to encrypt single or double length DES keys with standard CVs for key types DATA, DATAC, DATAM, DATAMV, DATA XLAT, EXPORTER, IKEYXLAT, IMPORTER, IPINENC, single-length MAC, single-length MACVER, OKEYXLAT, OPINENC, PINGEN and PINVER.
- NOCV key-encrypting keys can be used with triple length DATA keys. Because DATA keys have 0 CVs, processing will be the same as if the key-encrypting keys are standard key-encrypting keys (not the NOCV key-encrypting key).

**Note:** A key-encrypting key should be as strong or stronger than the key that it is wrapping.

You use key-encrypting keys to protect keys that are transported using any of the following verbs: Data Key Export, Key Export, Key Import, Clear Key Import, Multiple Clear Key Import, Key Generate, Key Generate2, Key Translate and Key Translate2.

For installations that do not support double-length key-encrypting keys, effective single-length keys are provided. For an effective single-length key, the clear key value of the left key half equals the clear key value of the right key half.

**Key-generating keys**
Key-generating keys are double-length keys used to derive other keys. This is often used in smart card applications.

*Table 2 on page 40* describes the key types.
<table>
<thead>
<tr>
<th>Key type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESDATA</td>
<td>Data encrypting key. Use the AES 128-bit, 192-bit, or 256-bit key to encipher and decipher data.</td>
</tr>
<tr>
<td>AESTOKEN</td>
<td>Can contain an AES key.</td>
</tr>
<tr>
<td>CIPHER</td>
<td>AES 128-bit, 192-bit, or 256-bit key is used to encrypt or decrypt data. It can be used in the Symmetric Algorithm Decipher and Symmetric Algorithm Encipher verbs. DES This single or double-length key is used to encrypt or decrypt data. It can be used in the Encipher and Decipher verbs. Used only to encrypt or decrypt data. This is a single or double length key and can be used in the Encipher or Decipher verbs.</td>
</tr>
<tr>
<td>CLRAES</td>
<td>Data encrypting key. The key value is not encrypted. Use this AES 128-bit, 192-bit, or 256-bit key to encipher and decipher data.</td>
</tr>
<tr>
<td>CLRDES</td>
<td>Data encrypting key. The key value is not encrypted. Use this DES single-length, double-length, or triple-length key to encipher and decipher data.</td>
</tr>
<tr>
<td>CVARDEC</td>
<td>The cryptographic variable decipher service, which is available in some CCA implementations, uses a CVARDEC key to decrypt plaintext by using the Cipher Block Chaining (CBC) method. This is a single-length key.</td>
</tr>
<tr>
<td>CVARENC</td>
<td>The cryptographic variable encipher service, which is available in some CCA implementations, uses a CVARENC key to encrypt plaintext by using the Cipher Block Chaining (CBC) method. This is a single-length key.</td>
</tr>
<tr>
<td>CVARPINE</td>
<td>Used to encrypt a PIN value for decryption in a PIN-printing application. This is a single-length key.</td>
</tr>
<tr>
<td>CVARXCVL</td>
<td>Used to encrypt special control values in DES key management. This is a single-length key.</td>
</tr>
<tr>
<td>CVARXCVR</td>
<td>Used to encrypt special control values in DES key management. This is a single-length key.</td>
</tr>
<tr>
<td>DATA</td>
<td>Data encrypting key. Use this DES single-length, double-length, or triple-length key to encipher and decipher data. Use the AES 128-bit, 192-bit, or 256-bit key to encipher and decipher data.</td>
</tr>
<tr>
<td>DATAC</td>
<td>Used to specify a DATA-class key that will perform in the Encipher and Decipher verbs, but not in the MAC Generate or MAC Verify verbs. This is a double-length key. Only available with a CEXC.</td>
</tr>
<tr>
<td>DATAM</td>
<td>Key-encrypting keys that have a control vector with this attribute formerly could only be used to transport keys with a key type of DATA, CIPHER, ENCIIPHER, DECIPHER, MAC, and MACVER. The meaning of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.</td>
</tr>
<tr>
<td>DATAMV</td>
<td>Used to specify a DATA-class key that performs in the MAC Verify verb, but not in the MAC Generate, Encipher, or Decipher verbs.</td>
</tr>
<tr>
<td>DATAXLAT</td>
<td>Data translation key. Use this single-length key to reencipher text from one DATA key to another.</td>
</tr>
<tr>
<td>DECIPHER</td>
<td>Used only to decrypt data. DECIPHER keys cannot be used in the Encipher (CSNBENC) verb. This is a single-length key.</td>
</tr>
<tr>
<td>DKYGENKY</td>
<td>Used to generate a diversified key based on the key-generating key. This is a double-length key.</td>
</tr>
<tr>
<td>ENCIIPHER</td>
<td>Used only to encrypt data. ENCIIPHER keys cannot be used in the Decipher (CSNBDEC) verb. This is a single-length key.</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>Exporter key-encrypting key. Use this double-length DES key or 128-bit, 192-bit or 256-bit AES key to convert a key from operational form into exportable form.</td>
</tr>
<tr>
<td>HMAC</td>
<td>Variable-length HMAC generation key. Use this key to generate or verify a Message Authentication Code using the keyed-hash MAC algorithm.</td>
</tr>
<tr>
<td>Key type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>HMACVER</td>
<td>Variable-length HMAC verification key. Use this key to verify a Message Authentication Code using the keyed-hash MAC algorithm.</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>Used to decrypt an input key in the Key Translate and Key Translate2 verbs. This is a double-length key.</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>Importer key-encrypting key. Exporter key-encrypting key. Use this double-length DES key or 128-bit, 192-bit or 256-bit AES key to convert a key from importable form into operational form.</td>
</tr>
<tr>
<td>IMP-PKA</td>
<td>Double-length limited-authority importer key used to encrypt PKA private key values in PKA external tokens.</td>
</tr>
<tr>
<td>IPINENC</td>
<td>Double-length input PIN-encrypting key. PIN blocks received from other nodes or automatic teller machine (ATM) terminals are encrypted under this type of key. These encrypted PIN blocks are the input to the Encrypted PIN Translate, Encrypted PIN Verify, and Clear PIN Generate Alternate verbs.</td>
</tr>
<tr>
<td>KEYGENKY</td>
<td>Used to generate a key based on the key-generating key. This is a double-length key.</td>
</tr>
<tr>
<td>MAC</td>
<td>Single, double-length, or variable-length MAC generation key. Use this key to generate a message authentication code.</td>
</tr>
<tr>
<td>MACVER</td>
<td>Single, double-length, or variable-length MAC verification key. Use this key to verify a message authentication code.</td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>Used to encrypt an output key in the Key Translate and Key Translate2 verbs. This is a double-length key.</td>
</tr>
<tr>
<td>OPINENC</td>
<td>Output PIN-encrypting key. Use this double-length output key to translate PINs. The output PIN blocks from the Encrypted PIN Translate, Encrypted PIN Generate, and Clear PIN Generate Alternate verbs are encrypted under this type of key.</td>
</tr>
<tr>
<td>PINGEN</td>
<td>PIN generation key. Use this double-length key to generate PINs.</td>
</tr>
<tr>
<td>PINVER</td>
<td>PIN verification key. Use this double-length key to verify PINs.</td>
</tr>
<tr>
<td>SECMSG</td>
<td>Used to encrypt PINs or keys in a secure message. This is a double-length key.</td>
</tr>
<tr>
<td>TOKEN</td>
<td>A key token that might contain a key.</td>
</tr>
</tbody>
</table>

Table 2. Key types (continued)

Table 3 lists key subtypes passed in the `rule_array` keyword.

<table>
<thead>
<tr>
<th>rule_array keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMEX-CSC</td>
<td>A MAC key that can be used for the AMEX CSC transaction validation process MAC calculation method, used with the Transaction Validation (CSNBTRV) verb.</td>
</tr>
<tr>
<td>ANSI X.9.9</td>
<td>A MAC key that can be used for the ANSI X.9.9 MAC calculation method, either for MAC Generate (CSNBMGN), MAC Verify (CSNBMV), or Transaction Validation (CSNBTRV). Other Control Vector bits could limit these usages.</td>
</tr>
<tr>
<td>ANY</td>
<td>Key-encrypting keys that have a control vector with this attribute can be used to transport any type of key. The meaning of this keyword has been discontinued, and its usage is allowed for backward compatibility reasons only.</td>
</tr>
<tr>
<td>ANY-MAC</td>
<td>Can be used with any function or MAC calculation method that uses a MAC key, such as MAC Generate (CSNBMGN), MAC Verify (CSNBMV), or Transaction Validation (CSNBTRV). This is the default configuration for a MAC key control vector.</td>
</tr>
<tr>
<td>CVVKEY-A</td>
<td>Can be used as 'Key A' in either the CVV Generate (CSNBCSG) or CVV Verify (CSNBCSV) verbs, as controlled by the CVV generation and verification Control Vector bits (bits 20 and 21 respectively).</td>
</tr>
<tr>
<td>CVVKEY-B</td>
<td>Can be used as 'Key B' in either the CVV Generate (CSNBCSG) or CVV Verify (CSNBCSV) verbs, as controlled by the CVV generation and verification Control Vector bits (bits 20 and 21 respectively).</td>
</tr>
<tr>
<td>rule_array keyword</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DATA</td>
<td>Data encrypting key. Use this 8-byte, 16-byte or 24-byte DES key or 16-byte, 24-byte or 32-byte AES key to encipher and decipher data.</td>
</tr>
<tr>
<td>EPINGENA</td>
<td>Legacy key subtype, used to turn on bit 19 of a PIN Generating Key Control Vector. The default PIN Generating Key type will have this bit on. No PIN generating or processing behavior is currently influenced by this key subtype parameter. EPINGENA is no longer supported, although the bit retains this definition for compatibility. There is no Encrypted Pin Generate Alternate verb</td>
</tr>
<tr>
<td>LMTD-KEK</td>
<td>Key-encrypting keys that have a control vector with this attribute formerly could only be used to exchange keys with key-encrypting keys that carry NOT-KEK, PIN, or DATA key-type ciphering restrictions. The usage of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.</td>
</tr>
<tr>
<td>NOT-KEK</td>
<td>Key-encrypting keys that have a control vector with this attribute formerly could not be used to transport key-encrypting keys. The meaning of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.</td>
</tr>
<tr>
<td>PIN</td>
<td>Key-encrypting keys that have a control vector with this attribute formerly could only be used to transport keys with a key type of PINVER, IPINENC, and OPINENC. The usage of this keyword has been discontinued and its usage is allowed for backward compatibility reasons only.</td>
</tr>
</tbody>
</table>
Figure 3. Control Vector Generate and Key Token Build CV keyword combinations for fixed-length DES key tokens
Clear keys
A clear key is the base value of a key, and is not encrypted under another key. Encrypted keys are keys whose base value has been encrypted under another key.

To convert a clear key to an encrypted data key in operational form, use either the Clear Key Import verb or the Multiple Clear Key Import verb.

Verbs for managing AES and DES key storage files
CCA provides API functions to allow application programs to manage the AES and DES key storage file, where key tokens are stored when the program references them by key label name.

The following verbs are used to manage the AES and DES key storage files:
- AES Key Record Create (CSNBAKRC)
- AES Key Record Delete (CSNBAKRD)
- AES Key Record List (CSNBAKRL)
- AES Key Record Read (CSNBAKRR)
- AES Key Record Write (CSNBAKRW)
- DES Key Record Create (CSNBKRC)
- DES Key Record Delete (CSNBKRD)
- DES Key Record List (CSNBKRL)
- DES Key Record Read (CSNBKRR)
- DES Key Record Write (CSNBKRW)

Verbs for managing the PKA key storage file and PKA keys in the cryptographic engine
The PKA key storage file is a repository for RSA keys, similar to the AES and DES key storage files.

An application can store keys in the key storage file and refer to them by label when using any of the verbs which accept RSA key tokens as input. The following verbs are used to manage the PKA key storage file, or PKA keys stored in the cryptographic engine:
- PKA Key Record Create (CSNDKRC)
- PKA Key Record Delete (CSNDKRD)
- PKA Key Record List (CSNDKRL)
- PKA Key Record Read (CSNDKRR)
- PKA Key Record Write (CSNDKRW)
- Retained Key Delete (CSNDRKD)
- Retained Key List (CSNDRKL)

EC Diffie-Hellman key agreement models
This topic discusses how to specify key agreement models.

Token agreement scheme
The caller must have both the required key tokens and both party's identifiers, including a randomly generated nonce.
Combine the exchanged nonce and Party Info into the party identifier. (Both parties must combine this information in the same format.) Then call the EC Diffie-Hellman verb, where “EC” means Elliptic Curve. Specify a skeleton token or the label of a skeleton token as the output key identifier to be used as a container for the computed symmetric key material. Note, both parties must specify the same key type in their skeleton key tokens.

- Specify rule-array keyword `DERIV01` to denote the Static Unified Model key agreement scheme.
- Specify an ECC token as the private key identifier containing this party’s ECC public-private key pair.
- Optionally specify a private KEK key identifier, if the key pair is in an external key token.
- Specify an ECC token as the public key identifier containing the other party’s ECC public key part.
- Specify a skeleton token as the output key identifier to be used as a container for the computed symmetric key material.
- Optionally specify an output KEK key identifier, if the output key is to be in an external key token.
- Specify the combined party info (including nonce) as the party identifier.
- Specify the desired size of the key to be derived (in bits) as the key bit length.

**Obtaining the raw "Z" value**

To use a key agreement scheme that differs from the above, you can obtain the raw shared secret "Z" value, and skip the key derivation step.

The caller must then derive the final key material using a method of their choice. Do not specify any party info.

- Specify rule array keyword `PASSTHRU` to denote no key agreement scheme.
- Specify an ECC token as the private key identifier containing this party’s ECC public-private key pair.
- Optionally specify a private KEK key identifier, if the key pair is in an external key token.
- Specify an ECC token as the public key identifier containing the other party’s ECC public key part.
- The output key identifier will be populated with the resulting shared secret material.

**Improved remote key distribution**

New methods have been added for securely transferring symmetric encryption keys to remote devices, such as Automated Teller Machines (ATMs), PIN-entry devices, and point of sale terminals.

These methods can also be used to transfer symmetric keys to another cryptographic system of any type, such as a different kind of Hardware Security Module (HSM) in an IBM or non-IBM computer server.

**Note:** This improved remote key distribute support is only available on IBM z9® and later.
This change replaces expensive human operations with network transactions that can be processed quickly and inexpensively. This method makes significant interoperability improvements to related cryptographic key-management functions.

For the purposes of this description, the ATM scenario will be used to illustrate operation of the new methods. Other uses of this method are also possible.

**Remote key loading**

*Remote key loading* is the process of installing symmetric encryption keys into a remotely located device from a central administrative site.

This encompasses two phases of key distributions:

- Distribution of initial key encrypting keys (KEKs) to a newly installed device. A KEK is a type of symmetric encryption key that is used to encrypt other keys so that they can be securely transmitted over unprotected paths.
- Distribution of operational keys or replacement KEKs, enciphered under a KEK currently installed in the device.

Access control points are assigned to roles to control keyword usage in the services provided for ATM remote key loading. Table 4 lists the access control points used by the ATM remote key loading function.

<table>
<thead>
<tr>
<th>Verb name</th>
<th>Entry point</th>
<th>Offset</th>
<th>Access Control Point name and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Block Create</td>
<td>CSNDTBC</td>
<td>X’030F’</td>
<td>Trusted Block Create - Create a Trusted Key Block in Inactive form</td>
</tr>
<tr>
<td>Trusted Block Create</td>
<td>CSNDTBC</td>
<td>X’0310’</td>
<td>Trusted Block Create - Activate an Inactive Trusted Key Block</td>
</tr>
<tr>
<td>PKA Key Import</td>
<td>CSNDPKI</td>
<td>X’0311’</td>
<td>PKA Key Import - Import an External Trusted Key Block to internal form</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Convert Trusted Block from external to internal format</td>
</tr>
<tr>
<td>PKA Key Import</td>
<td>CSNDPKI</td>
<td>X’0104’</td>
<td>PKA Key Import</td>
</tr>
<tr>
<td>Remote Key Export</td>
<td>CSNDRKX</td>
<td>X’0312’</td>
<td>Remote Key Export - Generate or export a key for use by a non-CCA node</td>
</tr>
<tr>
<td>Key Generate</td>
<td>CSNBKGN</td>
<td>X’00DB’</td>
<td>Key Generate - SINGLE-R</td>
</tr>
<tr>
<td>Remote Key Export</td>
<td>CSNDRKX</td>
<td></td>
<td>Replication of a single-length source key (which is either an RKX token or a CCA token) if the output symmetric encryption result is to be a CCA token, and the CV in the trusted block's Common Export Key Parameters TLV Object is 16 bytes with key form bits 'H' set to X'010' for the left half and X'001' for the right half.</td>
</tr>
<tr>
<td>Key Import</td>
<td>CSNBKIM</td>
<td>X’027B’</td>
<td>Key Import - Unrestricted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The importer key identifier in the initial code release must have unique halves.</td>
</tr>
<tr>
<td>Key Export</td>
<td>CSNBKEX</td>
<td>X’0276’</td>
<td>Key Export - Unrestricted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The transport key identifier in the initial code release must have unique halves.</td>
</tr>
</tbody>
</table>
Old remote key loading example

Use an ATM as an example of the remote key loading process: a new ATM has none of the purchaser’s keys installed when it is delivered from the manufacturer.

The process of getting the first key securely loaded is difficult.

The installation of the first key on the ATM has typically been done by loading the first KEK into each ATM manually, in multiple cleartext key parts. Using dual control for key parts, two separate people must carry key part values to the ATM, then load each key part manually. After they are inside the ATM, the key parts are combined to form the actual KEK. In this manner, neither of the two people has the entire key, protecting the key value from disclosure or misuse. This method is labor-intensive and error-prone, making it expensive.

New remote key loading methods

New remote key loading methods have been developed to overcome some of the shortcomings of the old manual key loading methods.

These new methods define acceptable techniques using public key cryptography to load keys remotely. Using these new methods, initial KEKs can be loaded without sending people to the remote device. This will reduce labor costs, be more reliable, and be much less expensive to install and change keys.

The new cryptographic features provide new methods for the creation and use of the special key forms needed for remote key distribution of this type. In addition, the new cryptographic features provide ways to solve long-standing barriers to secure key exchange with non-IBM cryptographic systems.

After an ATM is in operation, new keys can be installed as needed, by sending them enciphered under a KEK installed previously. This is straightforward in concept, but the cryptographic architecture in ATMs is often different from that of the host system that is sending the keys, and it is difficult to export the keys in a form understood by the ATM. For example, cryptographic architectures often enforce key-usage restrictions in which a key is bound to data describing limitations on how it can be used (for encrypting data, for encrypting keys, for operating on Message Authentication Codes (MACs), and so forth). The encoding of these restrictions and the method used to bind them to the key itself differs among cryptographic architectures, and it is often necessary to translate the format to that understood by the target device prior to a key being transmitted. It is difficult to do this without reducing security in the system; typically it is done by making it possible to arbitrarily change key-usage restrictions.

The methods described here provide a mechanism through which the system owner can securely control these translations, preventing the majority of attacks that could be mounted by modifying usage restrictions.

A data structure called a trusted block is defined to facilitate the remote key loading methods. The trusted block is the primary vehicle supporting these new methods. See “Trusted blocks” on page 647.

Verbs that support Secure Sockets Layer (SSL)

The Secure Sockets Layer (SSL) protocol, developed by Netscape Development Corporation, provides communications privacy over the Internet. Client/server applications can use the SSL protocol to provide secure communications and prevent eavesdropping, tampering, or message forgery.

Chapter 2. Using AES, DES, and HMAC cryptography and verbs
CCA provides verbs that support the RSA-encryption and RSA-decryption of PKCS 1.2-formatted symmetric key data to produce symmetric session keys. These session keys can then be used to establish an SSL session between the sender and receiver. The verbs provide SSL support:
- PKA Decrypt (CSNDPKD)
- PKA Encrypt (CSNDPKE)

**Enciphering and deciphering data**

To protect data, CCA can use the Data Encryption Standard (DES) or Advanced Encryption Standard (AES) algorithms to encipher or decipher data or keys.

Enciphering data protects it from disclosure to people who do not have authority to access it. Using algorithms that make it difficult and expensive for an unauthorized user to derive the original clear data within a practical time period assures privacy.

The DES algorithm is documented in the Federal Information Processing Standard #46. The AES algorithm is documented in the Federal Information Processing Standard #192. These verbs perform the enciphering and deciphering functions:
- Decipher (CSNBDEC)
- Encipher (CSNBENC)
- Symmetric Algorithm Decipher (CSNBSAD)
- Symmetric Algorithm Encipher (CSNBSAE)

**Managing data integrity and message authentication**

To ensure the integrity of transmitted messages and stored data, CCA provides DES-based Message Authentication Code (MAC) functions and several hashing functions, including Modification Detection Code (MDC), SHA-1, RIPEMD-160 and MD5.


The choice of verb depends on the security requirements of the environment in which you are operating. If you need to ensure the authenticity of the sender and also the integrity of the data, consider Message Authentication Code processing. If you need to ensure the integrity of transmitted data in an environment where it is not possible for the sender and the receiver to share a secret cryptographic key, consider hashing functions.

**Message authentication code processing**

The process of verifying the integrity and authenticity of transmitted messages is called message authentication.

Message authentication code (MAC) processing allows you to verify that a message was not altered or a message was not fraudulently introduced onto the system. You can check that a message you have received is the same one sent by the message originator. The message itself can be in clear or encrypted form. The comparison is performed within the cryptographic coprocessor. Because both the sender and receiver share a secret cryptographic key used in the MAC calculation, the MAC comparison also ensures the authenticity of the message.
In a similar manner, MACs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

CCA key typing makes it possible to give one party a key that can only be used to generate a MAC, and to give another party a corresponding key that can only be used to verify the MAC. This ensures that the second party cannot impersonate the first by generating MACs with their version of the key.

The coprocessor provides support for both single-length and double-length MAC generation and MAC verification keys. With the ANSI X9.9-1 single key algorithm, use the single-length MAC and MACVER keys.

CCA provides support for the use of data-encrypting keys in the MAC Generate and MAC Verify verbs, and also the use of a MAC generation key in the MAC Verify verb. This support permits CCA MAC verbs to interface more smoothly with non-CCA key distribution system.

HMAC codes are computed using the FIPS-198 Keyed-Hash Message Authentication Code method. See Chapter 8, “Verifying data integrity and authenticating messages,” on page 313.

These verbs are used to process MACs:
- MAC Generate (CSNBMGN)
- MAC Verify (CSNBMRV)

**Hashing functions**

Hashing functions are provided by two verbs.

These verbs are:
- MDC Generate (CSNBMMDG)
- One-Way Hash (CSNBOWH)

**Processing personal identification numbers**

The process of validating personal identities in a financial transaction system is called personal authentication.

The personal identification number (PIN) is the basis for verifying the identity of a customer across the financial industry networks. The financial industry needs functions to generate, translate, and verify PINs. These functions prevent unauthorized disclosures when organizations handle personal identification numbers.

The coprocessor supports the following algorithms for generating and verifying personal identification numbers:
- IBM 3624
- IBM 3624 PIN offset
- IBM German Bank Pool
- IBM German Bank Pool PIN Offset (GBP-PINO)
- VISA PIN validation value
- Interbank
You can translate PIN blocks from one format to another without the PIN being exposed in cleartext form. The coprocessor supports the following formats:

- ANSI X9.8
- ISO formats 0, 1, 2, 3
- VISA formats 1, 2, 3, 4
- IBM 4704 Encrypting PINPAD format
- IBM 3624 formats
- IBM 3621 formats
- ECI formats 1, 2, 3

With the capability to translate personal identification numbers into different PIN block formats, you can use personal identification numbers on different systems.

**Verifying credit card data**

The Visa International Service Association (VISA) and MasterCard International, Incorporated have specified a cryptographic method to calculate a value that relates to the personal account number (PAN), the card expiration date, and the service code.

The VISA card-verification value (CVV) and the MasterCard card-verification code (CVC) can be encoded on either track 1 or track 2 of a magnetic striped card and are used to detect forged cards. Because most online transactions use track-2, the CCA verbs generate and verify the CVV by the track-2 method.

The CVV Generate verb calculates a 1 - 5-byte value through the DES-encryption of the PAN, the card expiration date, and the service code using two data-encrypting keys or two MAC keys. The CVV Verify verb calculates the CVV by the same method, compares it to the CVV supplied by the application (which reads the credit card’s magnetic stripe) in the CVV_value, and issues a return code that indicates whether the card is authentic.

The following verbs are used to process and verify credit card data:

- Clear PIN Encrypt (CSNBCPE)
- Clear PIN Generate (CSNBPGN)
- Clear PIN Generate Alternate (CSNBCPA)
- CVV Generate (CSNBCSG)
- CVV Key Combine (CSNBCKC)
- CVV Verify (CSNBCSV)
- Encrypted PIN Generate (CSNBEPG)
- Encrypted PIN Translate (CSNBPTR)
- Encrypted PIN Verify (CSNBPVR)
- PIN Change/Unblock (CSNBPCU)
- Transaction Validation (CSNBTRV)

**Secure messaging**

The following verbs will assist applications in encrypting secret information such as clear keys and PIN blocks in a secure message.

---

1. The VISA CVV and the MasterCard CVC refer to the same value. CVV is used here to mean both CVV and CVC.
These verbs will execute within the secure boundary of the cryptographic coprocessor:
- Secure Messaging for Keys (CSNBSKY)
- Secure Messaging for PINs (CSNBSPN)

**Trusted Key Entry support**

The Trusted Key Entry (TKE) workstation provides a secure method of initializing and administering cryptographic coprocessors.

It is an optional System z feature, but it is mandatory if z/OS and CCA are not available on your system. Initialization of the coprocessor can be done through CCA for both the z/OS and Linux environments, either with or without TKE.

TKE Version 6.0 or higher is required in order to administer the CEX*C coprocessor features. You can use the TKE workstation to load DES master keys, PKA master keys, and operational keys in a secure way. TKE Version 6.0 and 7.0 can also set AES master keys on the CEX*C coprocessor.

You can load keys remotely and for multiple coprocessors, which can be in a single machine or in multiple machines. The TKE workstation eases the administration for using one coprocessor as a production machine and as a test machine at the same time, while maintaining security and reliability.

The TKE workstation can be used for enabling and disabling access control points for verbs executed on the cryptographic coprocessor. See Appendix G, “Access control points and verbs,” on page 723 for additional information.

For complete details about the TKE workstation, see z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User’s Guide.

**Typical sequences of CCA verbs**

Sample sequences in which the CCA verbs might be called:

**Combinations of the verbs**

**Combination A (DATA keys only)**
1. Random number generate
2. Clear key import or multiple clear key import
3. Encipher/decipher
4. Data key export or key export (optional step)

**Combination B**
1. Random number generate
2. Secure key import or multiple secure key import
3. Any service
4. Data key export for DATA keys, or key export in the general case (optional step)

**Combination C**
1. Key generate (OP form only)
2. Any service
3. Key export (optional)
Combination D
1. Key generate (OPEX form)
2. Any service

Combination E
1. Key Generate (IM form only)
2. Key Import
3. Any service
4. Key Export (optional)

Combination F
1. Key Generate (IMEX form)
2. Key Import
3. Any service

Combination G
1. Key Generate
2. AES or DES Key Record Create
3. AES or DES Key Record Write
4. Any service (passing label of the key just generated)

Combination H
1. Key Import
2. AES or DES Key Record Create
3. AES or DES Key Record Write
4. Any service (passing label of the key just generated)

Notes
1. An example of “any service” is CSNBENC.
2. These combinations exclude verbs that can be used on their own; for example, Key Export or encode, or using the Key Generate verb to generate an exportable key.
3. These combinations do not show key communication, or the transmission of any output from an CCA verb.

The key forms are described in Appendix C, “Key forms and types used in the Key Generate verb,” on page 663 and “Key Generate (CSNBKGN)” on page 167.

Using the CCA node and master key management verbs

The verbs listed in this topic are used to use the CCA node and master key management functions.

- Cryptographic Facility Query (CSUACFQ)
- Cryptographic Facility Version (CSUACFV)
- Cryptographic Resource Allocate (CSUACRA)
- Cryptographic Resource Deallocate (CSUACRD)
- Cryptographic Variable Encipher (CSNBCVE)
- Data Key Export (CSNBDKX)
- Data Key Import (CSNBDKM)
- Diversified Key Generate (CSNBDKG)
- EC Diffie-Hellman (CSNDEDH)
Summary of the CCA nodes and resource control verbs

This table lists the CCA nodes and resource control verbs described in this document. The table also references the chapter that describes the verb.

Table 5. Summary of CCA nodes and resource control verbs

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Verb name</th>
<th>Description</th>
<th>Topic/Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSUACFQ</td>
<td>Cryptographic Facility Query</td>
<td>Retrieves information about the coprocessor and the CCA application program in that coprocessor.</td>
<td>“Cryptographic Facility Query (CSUACFQ)” on page 77</td>
</tr>
<tr>
<td>CSUACFV</td>
<td>Cryptographic Facility Version</td>
<td>Retrieve the Security Application Program Interface (SAPI) version and build date.</td>
<td>“Cryptographic Facility Version (CSUACFV)” on page 108</td>
</tr>
<tr>
<td>CSUACRA</td>
<td>Cryptographic Resource Allocate</td>
<td>Allocates specific CCA coprocessor for use by the thread or process, depending on the scope of the verb.</td>
<td>“Cryptographic Resource Allocate (CSUACRA)” on page 110</td>
</tr>
<tr>
<td>CSUACRD</td>
<td>Cryptographic Resource Deallocate</td>
<td>De-allocates a specific CCA coprocessor that is allocated by the thread or process, depending on the scope of the verb.</td>
<td>“Cryptographic Resource Deallocate (CSUACRD)” on page 113</td>
</tr>
<tr>
<td>Entry point</td>
<td>Verb name</td>
<td>Description</td>
<td>Topic/Page</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>CSNBKSI</td>
<td>Key Storage Initialization</td>
<td>This verb initializes a key-storage file using the current symmetric or asymmetric master-key. The initialized key storage does not contain any preexisting key records. The name and path of the key storage data and index file are established differently in each operating environment. Note that HMAC keys are not supported for key storage.</td>
<td>“Key Storage Initialization (CSNBKSI)” on page 116</td>
</tr>
<tr>
<td>CSNBMKP</td>
<td>Master Key Process</td>
<td>Operates on the three master-key registers: new, current, and old. This verb is used to clear the new and the old master-key registers, generate a random master-key value in the new master-key register, XOR a clear value as a key part into the new master-key register, and set the master key, which transfers the current master-key to the old master-key register and the new master-key to the current master-key register.</td>
<td>“Master Key Process (CSNBMKP)” on page 119</td>
</tr>
<tr>
<td>CSUARNT</td>
<td>Random Number Tests</td>
<td>Invokes the USA NIST FIPS PUB 140-1 specified cryptographic operational tests. These tests, selected by a rule_array keyword, consist of known-answer tests of DES, RSA, and SHA-1 processes and, for random numbers, monobit test, poker test, runs test, and log-run test.</td>
<td>“Random Number Tests (CSUARNT)” on page 124</td>
</tr>
</tbody>
</table>

**Summary of the AES, DES, and HMAC verbs**

Hash Message Authentication Code (HMAC) support was added in CCA Release 4.1.0. All of the HMAC verbs and features described in this chapter require CCA 4.1.0 or CCA 4.2.0 in order to run.

Table 6 lists the AES, DES, and HMAC verbs described in this document. The table also references the chapter that describes the verb.

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Verb name</th>
<th>Description</th>
<th>Topic/Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNBCKI</td>
<td>Clear Key Import</td>
<td>Imports an 8-byte clear DATA key, enciphers it under the master key, and places the result into an internal key token. This verb converts the clear key into operational form as a DATA key.</td>
<td>“Clear Key Import (CSNBCKI)” on page 129</td>
</tr>
</tbody>
</table>
Table 6. Summary of CCA AES, DES, and HMAC verbs (continued)

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Verb name</th>
<th>Description</th>
<th>Topic/Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNBCKM</td>
<td>Multiple Clear Key Import</td>
<td>Imports a single-length, double-length, or triple-length clear DATA key that is used to encipher or decipher data. It accepts a clear key and enciphers the key under the host master key, returning an encrypted DATA key in operational form in an internal key token.</td>
<td>“Multiple Clear Key Import (CSNBCKM)” on page 131</td>
</tr>
<tr>
<td>CSNBCVG</td>
<td>Control Vector Generate</td>
<td>Builds a control vector from keywords specified by the key_type and rule_array parameters.</td>
<td>“Control Vector Generate (CSNBCVG)” on page 134</td>
</tr>
<tr>
<td>CSNBCVT</td>
<td>Control Vector Translate</td>
<td>Changes the control vector used to encipher an external DES key.</td>
<td>“Control Vector Translate (CSNBCVT)” on page 137</td>
</tr>
<tr>
<td>CSNBCVE</td>
<td>Cryptographic Variable Encipher</td>
<td>Encrypts plaintext using a CVARENC key to produce ciphertext using the Cipher Block Chaining (CBC) method.</td>
<td>“Cryptographic Variable Encipher (CSNBCVE)” on page 137</td>
</tr>
<tr>
<td>CSNBDKX</td>
<td>Data Key Export</td>
<td>Re-enciphers a DATA key from encryption under the master key to encryption under an exporter key-encrypting key, making it suitable for export to another system.</td>
<td>“Data Key Export (CSNBDKX)” on page 144</td>
</tr>
<tr>
<td>CSNBDKM</td>
<td>Data Key Import</td>
<td>Imports an encrypted source DES single-length or double-length DATA key and creates or updates a target internal key token with the master key enciphered source key.</td>
<td>“Data Key Import (CSNBDKM)” on page 146</td>
</tr>
<tr>
<td>CSNBDKG</td>
<td>Diversified Key Generate</td>
<td>Generates a key based upon the key-generating key, the processing method, and the parameter data that is supplied. The control vector of the key-generating key also determines the type of target key that can be generated.</td>
<td>“Diversified Key Generate (CSNBDKG)” on page 146</td>
</tr>
<tr>
<td>CSNDEDH</td>
<td>EC Diffie-Hellman</td>
<td>Creates symmetric key material from a pair of Elliptic Curve Cryptography (ECC) keys using the Elliptic Curve Diffie-Hellman (ECDH) protocol.</td>
<td>“EC Diffie-Hellman (CSNDEDH)” on page 153</td>
</tr>
<tr>
<td>CSNBKEX</td>
<td>Key Export</td>
<td>Re-enciphers a key from encryption under a master key variant to encryption under the same variant of an exporter key-encrypting key, making it suitable for export to another system.</td>
<td>“Key Export (CSNBKEX)” on page 164</td>
</tr>
<tr>
<td>CSNBKGN</td>
<td>Key Generate</td>
<td>Generates a 64-bit, 128-bit, 192-bit, or 256-bit odd parity key, or a pair of keys; and returns them in encrypted forms (operational, exportable, or importable). Key Generate does not produce keys in plaintext.</td>
<td>“Key Generate (CSNBKGN)” on page 167</td>
</tr>
<tr>
<td>Entry point</td>
<td>Verb name</td>
<td>Description</td>
<td>Topic/Page</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>CSNBKGN2</td>
<td>Key Generate2</td>
<td>Generates either one or two HMAC keys. This verb does not produce keys in clear form and all keys are returned in encrypted form. When two keys are generated, each key has the same clear value, although this clear value is not exposed outside the secure cryptographic feature. This verb returns variable-length CCA key tokens and uses the AESKW wrapping method. Operational keys will be encrypted under the AES master key.</td>
<td>&quot;Key Generate2 (CSNBKGN2)&quot; on page 176</td>
</tr>
<tr>
<td>CSNBKIM</td>
<td>Key Import</td>
<td>Re-enciphers a key from encryption under an importer key-encrypting key to encryption under the master key. The re-enciphered key is in the operational form.</td>
<td>&quot;Key Import (CSNBKIM)&quot; on page 184</td>
</tr>
<tr>
<td>CSNBKPI</td>
<td>Key Part Import</td>
<td>Combines the clear key parts of any key type and returns the combined key value in an internal key token or an update to the CCA key storage file.</td>
<td>&quot;Key Part Import (CSNBKPI)&quot; on page 187</td>
</tr>
<tr>
<td>CSNBKPI2</td>
<td>Key Part Import2</td>
<td>Combines the clear key parts of any HMAC key type from an internal variable-length symmetric key-token, and returns the combined key value in an internal variable-length symmetric key-token or an update to the CCA key storage file.</td>
<td>&quot;Key Part Import2 (CSNBKPI2)&quot; on page 191</td>
</tr>
<tr>
<td>CSNBKYT</td>
<td>Key Test</td>
<td>Generates or verifies (depending on keywords in the rule_array) a secure verification pattern for keys. This verb requires the tested key to be in the clear or encrypted under the master key.</td>
<td>&quot;Key Test (CSNBKYT)&quot; on page 195</td>
</tr>
<tr>
<td>CSNBKYT2</td>
<td>Key Test2</td>
<td>Generates or verifies (depending on keywords in the rule_array) a secure cryptographic verification pattern for keys contained in a variable-length symmetric key-token. The key to test can be in the clear or encrypted under a master key. Requires the tested key to be in the clear or encrypted under the master key.</td>
<td>&quot;Key Test2 (CSNBKYT2)&quot; on page 200</td>
</tr>
<tr>
<td>CSNBKYTX</td>
<td>Key Test Extended</td>
<td>This verb is essentially the same as Key Test, except for the following: • In addition to operating on internal keys and key parts, this verb also operates on external keys and key parts. • This verb does not operate on clear keys, and does not accept rule_array keywords CLR-A128, CLR-A192, CLR-A256, KEY-CLR, and KEY-CLRD.</td>
<td>&quot;Key Test Extended (CSNBKYTX)&quot; on page 204</td>
</tr>
<tr>
<td>Entry point</td>
<td>Verb name</td>
<td>Description</td>
<td>Topic/Page</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
</tbody>
</table>
| CSNBKTB     | Key Token Build   | Builds an internal or external token from the supplied parameters. You can use this verb to build CCA key tokens for all key types that CCA supports. The resulting token can be used as input to the Key Generate, and Key Part Import verbs.                                                                                                                      | “Key Token Build
(CSNBKTB)” on page 209 |
| CSNBKTB2    | Key Token Build2  | Builds variable-length internal or external key tokens for all key types that the coprocessor supports. The key token is built based on parameters that you supply. The resulting token can be used as input to the Key Generate2, and Key Part Import2 verbs. A clear key token built by this verb can be used as input to the Key Test2 verb. This verb supports internal HMAC tokens, both as clear key tokens and as skeleton tokens containing no key. | “Key Token Build2
(CSNBKTB2)” on page 214 |
| CSNBKTC     | Key Token Change  | Re-enciphers a DES key from encryption under the old master key to encryption under the current master key, and to update the keys in internal DES key-tokens.                                                                                                                                                                                         | “Key Token Change
(CSNBKTC)” on page 218 |
| CSNBKTC2    | Key Token Change2 | Re-enciphers a variable-length HMAC key from encryption under the old master key to encryption under the current master key. This verb also updates the keys in internal HMAC key-tokens.                                                                                                                                                                                      | “Key Token Change2
(CSNBKTC2)” on page 221 |
| CSNBKTP     | Key Token Parse   | Disassembles a key token into separate pieces of information. This verb can disassemble an external key-token or an internal key-token in application storage.                                                                                                                                                                                    | “Key Token Parse
(CSNBKTP)” on page 224 |
| CSNBKTP2    | Key Token Parse2  | Disassembles a variable-length symmetric key-token into separate pieces of information. The verb can disassemble an external or internal variable-length symmetric key-token in application storage. The verb returns some of the key-token information in a set of variables identified by individual parameters, and returns the remaining information as keywords in the rule array. | “Key Token Parse2
(CSNBKTP2)” on page 229 |
| CSNBKTR     | Key Translate     | Uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.                                                                                                                                                                                                   | “Key Translate
(CSNBKTR)” on page 239 |
<table>
<thead>
<tr>
<th>Entry point</th>
<th>Verb name</th>
<th>Description</th>
<th>Topic/Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNBKTR2</td>
<td>Key Translate2</td>
<td>Uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment. This verb differs from the Key Translate verb in that Key Translate2 can process both fixed-length and variable-length symmetric key tokens.</td>
<td>&quot;Key Translate2 (CSNBKTR2)&quot; on page 241</td>
</tr>
<tr>
<td>CSNDPKD</td>
<td>PKA Decrypt</td>
<td>Uses an RSA private key to decrypt the RSA-encrypted key value and return the clear key value to the application.</td>
<td>&quot;PKA Decrypt (CSNDPKD)&quot; on page 245</td>
</tr>
<tr>
<td>CSNDPKE</td>
<td>PKA Encrypt</td>
<td>Encrypts a supplied clear key value under an RSA public key. The supplied key can be formatted using the PKCS 1.2 or ZERO-PAD methods prior to encryption.</td>
<td>&quot;PKA Encrypt (CSNDPKE)&quot; on page 248</td>
</tr>
<tr>
<td>CSNBPEX</td>
<td>Prohibit Export</td>
<td>Modifies the control vector of a CCA key token so that the key cannot be exported. This verb operates only on internal key tokens.</td>
<td>&quot;Prohibit Export (CSNBPEX)&quot; on page 252</td>
</tr>
<tr>
<td>CSNBPEXX</td>
<td>Prohibit Export Extended</td>
<td>Modifies an external DES key-token so that the key can no longer be exported after it has been imported. This verb operates only on internal key tokens.</td>
<td>&quot;Prohibit Export Extended (CSNBPEXX)&quot; on page 254</td>
</tr>
<tr>
<td>CSNBRKA</td>
<td>Restrict Key Attribute</td>
<td>Modifies an operational variable-length key so that it cannot be exported.</td>
<td>&quot;Restrict Key Attribute (CSNBRKA)&quot; on page 256</td>
</tr>
<tr>
<td>CSNBRNG</td>
<td>Random Number Generate</td>
<td>Generates an 8-byte cryptographic-quality random number suitable for use as an encryption key or for other purposes. The output can be specified in three forms of parity: RANDOM, ODD, and EVEN.</td>
<td>&quot;Random Number Generate (CSNBRNG)&quot; on page 260</td>
</tr>
<tr>
<td>CSNBRNGL</td>
<td>Random Number Generate Long</td>
<td>Generates a cryptographic-quality random number suitable for use as an encryption key or for other purposes, ranging from 1 - 8192 bytes in length. The output can be specified in three forms of parity: RANDOM, ODD, and EVEN.</td>
<td>&quot;Random Number Generate Long (CSNBRNGL)&quot; on page 262</td>
</tr>
<tr>
<td>CSNDSYX</td>
<td>Symmetric Key Export</td>
<td>Transfers an application-supplied symmetric key (a DATA key) from encryption under the AES, DES or HMAC master key to encryption under an application-supplied RSA public key. The application-supplied DATA key must be an AES, DES or HMAC internal key token, or the label of an AES or DES key token in the CCA key storage file. The Symmetric Key Import and Symmetric Key Import2 verb can import the PKA-encrypted key form at the receiving node. Support for HMAC key was added beginning with CCA 4.1.0.</td>
<td>&quot;Symmetric Key Export (CSNDSYX)&quot; on page 265</td>
</tr>
<tr>
<td>Entry point</td>
<td>Verb name</td>
<td>Description</td>
<td>Topic/Page</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CSNDSYG</td>
<td>Symmetric Key Generate</td>
<td>Generate a symmetric key (a DATA key) and return the key in two forms: DES-encrypted and encrypted under an RSA public key. The DES-encrypted key can be an internal token encrypted under a host DES master key, or an external form encrypted under a KEK. (You can use the Symmetric Key Import verb to import the PKA-encrypted form.)</td>
<td>“Symmetric Key Generate (CSNDSYG)” on page 271</td>
</tr>
<tr>
<td>CSNDSYI</td>
<td>Symmetric Key Import</td>
<td>Import a symmetric AES or DES DATA key enciphered under an RSA public key into operational form enciphered under a DES master key.</td>
<td>“Symmetric Key Import (CSNDSYI)” on page 276</td>
</tr>
<tr>
<td>CSNDSYI2</td>
<td>Symmetric Key Import2</td>
<td>Use this verb to import an HMAC key that has been previously formatted and enciphered under an RSA public key by the Symmetric Key Export verb. The formatted and RSA-enciphered key is contained in an external variable-length symmetric key-token. The key is deciphered using the associated RSA private-key. The recovered HMAC key is re-enciphered under the AES master-key. The re-enciphered key is then returned in an internal variable-length symmetric key-token. The key algorithm for this verb is HMAC.</td>
<td>“Symmetric Key Import2 (CSNDSYI2)” on page 280</td>
</tr>
<tr>
<td>CSNBDEC</td>
<td>Decipher</td>
<td>Deciphers data using cipher block chaining mode of DES. The result is called plaintext.</td>
<td>“Decipher (CSNBDEC)” on page 289</td>
</tr>
<tr>
<td>CSNBENC</td>
<td>Encipher</td>
<td>Enciphers data using the cipher block chaining mode of DES. The result is called ciphertext.</td>
<td>“Encipher (CSNBENC)” on page 293</td>
</tr>
<tr>
<td>CSNBSAD</td>
<td>Symmetric Algorithm Decipher</td>
<td>Deciphers data using the AES cipher block chaining mode.</td>
<td>“Symmetric Algorithm Decipher (CSNBSAD)” on page 298</td>
</tr>
<tr>
<td>CSNBSAE</td>
<td>Symmetric Algorithm Encipher</td>
<td>Enciphers data using the AES cipher block chaining mode</td>
<td>“Symmetric Algorithm Encipher (CSNBSAE)” on page 303</td>
</tr>
<tr>
<td>CSNBHMG</td>
<td>HMAC Generate</td>
<td>Generates a keyed hash message authentication code (HMAC) for the text string provided as input. See Chapter 8, “Verifying data integrity and authenticating messages,” on page 313.</td>
<td>“HMAC Generate (CSNBHMG)” on page 315</td>
</tr>
<tr>
<td>CSNBHMV</td>
<td>HMAC Verify</td>
<td>Verifies a keyed hash message authentication code (HMAC) for the text string provided as input. See Chapter 8, “Verifying data integrity and authenticating messages,” on page 313.</td>
<td>“HMAC Verify (CSNBHMV)” on page 319</td>
</tr>
</tbody>
</table>
### Table 6. Summary of CCA AES, DES, and HMAC verbs (continued)

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Verb name</th>
<th>Description</th>
<th>Topic/Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNBMGN</td>
<td>MAC Generate</td>
<td>Generates a 4, 6, or 8-byte Message Authentication Code (MAC) for a text string that the application program supplies. The MAC is computed using either the ANSI X9.9-1 algorithm or the ANSI X9.19 optional double key algorithm and padding could be applied according to the EMV specification.</td>
<td>“MAC Generate (CSNBMGN)” on page 323</td>
</tr>
<tr>
<td>CSNBMVR</td>
<td>MAC Verify</td>
<td>Verifies a 4, 6, or 8-byte Message Authentication Code (MAC) for a text string that the application program supplies. The MAC is computed using either the ANSI X9.9-1 algorithm or the ANSI X9.19 optional double key algorithm and padding could be applied according to the EMV specification. The computed MAC is compared with a user-supplied MAC.</td>
<td>“MAC Verify (CSNBMVR)” on page 327</td>
</tr>
<tr>
<td>CSNBMDG</td>
<td>MDC Generate</td>
<td>Creates a 128-bit hash value (Modification Detection Code) on a data string whose integrity you intend to confirm.</td>
<td>“MDC Generate (CSNBMDG)” on page 332</td>
</tr>
<tr>
<td>CSNBOWH</td>
<td>One-Way Hash</td>
<td>Generates a one-way hash on specified text.</td>
<td>“One-Way Hash (CSNBOWH)” on page 342</td>
</tr>
<tr>
<td>Chapter 10, “Financial services,” on page 393</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSNBCPE</td>
<td>Clear PIN Encrypt</td>
<td>Formats a PIN into a PIN block format and encrypts the results. You can also use this verb to create an encrypted PIN block for transmission. With the RANDOM keyword, you can have the verb generate random PIN numbers.</td>
<td>“Clear PIN Encrypt (CSNBCPE)” on page 405</td>
</tr>
</tbody>
</table>
| CSNBPGN     | Clear PIN Generate      | Generates a clear personal identification number (PIN), a PIN verification value (PVV), or an offset using one of the following algorithms:  
- IBM 3624 (IBM-PIN or IBM-PINO)  
- IBM German Bank Pool (GBP-PIN or GBP-PINO)  
- VISA PIN validation value (VISA-PVV)  
- Interbank PIN (INBK-PIN) | “Clear PIN Generate (CSNBPGN)” on page 408 |
| CSNBCPA     | Clear PIN Generate Alternate | Generates a clear VISA PIN validation value (PVV) from an input encrypted PIN block. The PIN block might have been encrypted under either an input or output PIN encrypting key. The IBM-PINO algorithm is supported to produce a 3624 offset from a customer selected encrypted PIN. The PIN block must be encrypted under either an input PIN-encrypting key (IPINENC) or output PIN-encrypting key (OPINENC). | “Clear PIN Generate Alternate (CSNBCPA)” on page 412 |
Table 6. Summary of CCA AES, DES, and HMAC verbs (continued)

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Verb name</th>
<th>Description</th>
<th>Topic/Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNBCSG</td>
<td>CVV Generate</td>
<td>Generates a VISA Card Verification Value (CVV) or a MasterCard Card Verification Code (CVC) as defined for track 2.</td>
<td>“CVV Generate (CSNBCSG)” on page 417</td>
</tr>
<tr>
<td>CSNBCKC</td>
<td>CVV Key Combine</td>
<td>Combine two single-length operational DES keys that are suitable for use with the CVV (card-verification value) algorithm into one operational TDES key.</td>
<td>“CVV Key Combine (CSNBCKC)” on page 421</td>
</tr>
<tr>
<td>CSNBCSV</td>
<td>CVV Verify</td>
<td>Verifies a VISA Card Verification Value (CVV) or a MasterCard Card Verification Code (CVC) as defined for track 2.</td>
<td>“CVV Verify (CSNBCSV)” on page 426</td>
</tr>
<tr>
<td>CSNBEPG</td>
<td>Encrypted PIN Generate</td>
<td>Generates and formats a PIN and encrypts the PIN block.</td>
<td>“Encrypted PIN Generate (CSNBEPG)” on page 430</td>
</tr>
<tr>
<td>CSNBPTK</td>
<td>Encrypted PIN Translate</td>
<td>Re-enciphers a PIN block from one PIN-encrypting key to another and, optionally, changes the PIN block format. UKPT keywords are supported. You must identify the input PIN-encrypting key that originally enciphers the PIN. You also need to specify the output PIN-encrypting key that you want the verb to use to encipher the PIN. If you want to change the PIN block format, specify a different output PIN block format from the input PIN block format.</td>
<td>“Encrypted PIN Translate (CSNBPTK)” on page 435</td>
</tr>
<tr>
<td>CSNBPVK</td>
<td>Encrypted PIN Verify</td>
<td>Verifies a supplied PIN using one of the following algorithms: • IBM 3624 (IBM-PIN or IBM-PINO) • IBM German Bank Pool (GBP-PIN or GBP-PINO) • VISA PIN validation value (VISA-PVV) • Interbank PIN (INBK-PIN) UKPT keywords are supported.</td>
<td>“Encrypted PIN Verify (CSNBPVK)” on page 441</td>
</tr>
<tr>
<td>CSNBPCU</td>
<td>PIN Change/Unblock</td>
<td>Supports the PIN change algorithms specified in the VISA Integrated Circuit Card Specification; available only on an IBM z890 or IBM z990 with May 2004 or later version of Licensed Internal Code (LIC).</td>
<td>“PIN Change/Unblock (CSNBPCU)” on page 446</td>
</tr>
<tr>
<td>CSNBSKY</td>
<td>Secure Messaging for Keys</td>
<td>Encrypts a text block, including a clear key value decrypted from an internal or external DES token.</td>
<td>“Secure Messaging for Keys (CSNBSKY)” on page 453</td>
</tr>
<tr>
<td>CSNBSNP</td>
<td>Secure Messaging for PINs</td>
<td>Encrypts a text block, including a clear PIN block recovered from an encrypted PIN block.</td>
<td>“Secure Messaging for PINs (CSNBSNP)” on page 457</td>
</tr>
<tr>
<td>Entry point</td>
<td>Verb name</td>
<td>Description</td>
<td>Topic/Page</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>CSNBTRV</td>
<td>Transaction Validation</td>
<td>Supports the generation and validation of American Express card security codes; available only on an IBM z890 or IBM z990 with May 2004 or later version of Licensed Internal Code (LIC).</td>
<td>&quot;Transaction Validation (CSNBTRV)&quot; on page 462</td>
</tr>
</tbody>
</table>
Chapter 3. Introducing PKA cryptography and using PKA verbs

This topic introduces Public Key Algorithm (PKA) and Elliptic Curve Cryptography (ECC) support, and describes programming considerations for using the CCA PKA verbs, such as the PKA key token structure and key management.

You can use PKA support to exchange symmetric algorithm secret keys securely, and to compute digital signatures for authenticating messages to users.

The preceding chapters focused on AES or DES cryptography or secret-key cryptography. This cryptography is symmetric (senders and receivers use the same key, which must be exchanged securely in advance, to encipher and decipher data).

Public key cryptography does not require exchanging a secret key. It is asymmetric (the sender and receiver each have a pair of keys, a public key and a different but corresponding private key).

PKA key algorithms

Public key cryptography uses a key pair consisting of a public key and a private key.

The PKA public key uses one of the following algorithms:

Rivest-Shamir-Adleman (RSA)

The RSA algorithm is the most widely used and accepted of the public key algorithms. It uses three quantities to encrypt and decrypt text: a public exponent (PU), a private exponent (PR), and a modulus (M). Given these three and some cleartext data, the algorithm generates ciphertext as follows:

\[ \text{ciphertext} = \text{cleartext}^{PU} \pmod{M} \]

Similarly, the following operation recovers cleartext from ciphertext:

\[ \text{cleartext} = \text{ciphertext}^{PR} \pmod{M} \]

Elliptic Curve Digital Signature Algorithm (ECDSA)

The ECDSA algorithm uses elliptic curve cryptography (an encryption system based on the properties of elliptic curves) to provide a variant of the Digital Signature Algorithm.

PKA master keys

On the Cryptographic Coprocessor, PKA keys are protected by the Asymmetric-Keys Master Key (ASYM-MK).

The ASYM-MK is a triple-length DES key used to protect PKA private keys. On the Cryptographic Coprocessor, the ASYM-MK protects RSA private keys.

© Copyright IBM Corp. 2007, 2014
Starting with the IBM zEnterprise 196 configured with a CEX3C, there are two PKA master keys: the ASYM-MK mentioned above, and the 256-bit AES PKA Master Key (APKA-MK), used to protect ECC private keys stored in ECC key tokens.

In order for PKA verbs to function on the processor, the hash pattern of the ASYM-MK must match the hash pattern of the SYM-MK on the Cryptographic Coprocessor Feature. The administrator installs the PKA master keys on the Cryptographic Coprocessor Feature and the ASYM-MK on the coprocessor by using either the pass phrase initialization routine, the Clear Master Key Entry panels, or the optional Trusted Key Entry (TKE) workstation.

**Operational private keys**

Operational private keys are protected under two layers of DES encryption.

They are encrypted under an Object Protection Key (OPK) that in turn is encrypted under the ASYM-MK. You dynamically generate the OPK for each private key at import time or when the private key is generated on a CEX2C or CEX3C. CCA provides a public key storage file for the storage of application PKA keys. Although you cannot change PKA master keys dynamically, the PKA Key Token Change verb can be run to change a private PKA token (RSA or ECC) from encryption under the old ASYM-MK (or APKA-MK) to encryption under the current ASYM-MK (or APKA-MK). This verb requires a CEX2C or CEX3C.

**PKA verbs**

The CEX2C provides application programming interfaces to some PKA function.

These PKA functions are:
- RSA digital signature functions
- Key management and key generation functions
- DES key distribution functions
- Data encryption functions

The CEX3C feature, which became available in March 2010 for the IBM System z10 and newer models, provides all the functions provided by the CEX2C and adds application programming interfaces to the following PKA functions:
- ECC digital signature functions
- ECC key management and key generation functions
- ECC-based and RSA-based services for:
  - DES and AES key derivation
  - Diffie-Hellman key agreement for DES and AES
  - Key distribution functions for DES and AES keys

**Verbs supporting digital signatures**

CCA provides verbs that support digital signatures.

These verbs are:
- Digital Signature Generate (CSNDDSG)
- Digital Signature Verify (CSNDDSV)
Verbs for PKA key management

CCA provides verbs for PKA key management.

These verbs are:
- PKA Key Generate (CSNDPKG)
- PKA Key Import (CSNDPKI)
- PKA Key Token Build (CSNDPKB)
- PKA Key Token Change (CSNDKTC)
- PKA Key Translate (CSNDPKT)
- PKA Public Key Extract (CSNDPKX)
- Remote Key Export (CSNDRKX)
- Trusted Block Create (CSNDTBC)

PKA key tokens

PKA key tokens contain RSA or ECC private or public keys.

PKA tokens are variable length because they contain either RSA or ECC key values, which are variable in length. Consequently, length parameters precede all PKA token parameters. The maximum allowed size is 3500 bytes. PKA key tokens consist of a token header, any required sections, and any optional sections. Optional sections depend on the token type. PKA key tokens can be public or private, and private key tokens can be internal or external. Therefore, there are three basic types of tokens, each of which can contain either RSA or ECC information:
- A public key token
- A private external key token
- A private internal key token

Public key tokens contain only the public key. Private key tokens contain the public and private key pair. Table 7 summarizes the sections in each type of token.

<table>
<thead>
<tr>
<th>Section</th>
<th>Public external key token</th>
<th>Private external key token</th>
<th>Private internal key token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RSA or ECC private key information</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RSA or ECC public key information</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Key name (optional)</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Internal information</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

As with DES key tokens, the first byte of a PKA key token contains the token identifier which indicates the type of token.

A first byte of X'1E' indicates an external token with a cleartext public key and optionally a private key that is either in cleartext or enciphered by a transport key-encrypting key. An external key token is in importable key form. It can be sent on the link.
A first byte of X'1F' indicates an internal token with a cleartext public key and a private key that is enciphered by the PKA master key and ready for internal use. An internal key token is in operational key form. A PKA private key token must be in operational form for the coprocessor to use it. (PKA public key tokens are used directly in the external form.)

Formats for public and private external and internal RSA and ECC key tokens begin in “RSA public key token” on page 615.

PKA key management

You can generate RSA and ECC keys using the CCA PKA Key Generate verb.

- Using the Transaction Security System PKA Key Generate verb, or a comparable product from another vendor.

You can use the PKA Key Generate verb to generate internal and external PKA tokens. You can also generate RSA keys on another system and then import them to the cryptographic coprocessor. To input a clear RSA key, create the token with the PKA Key Token Build verb and import it using the PKA Key Import verb. To input an encrypted RSA key, use the PKA Key Import verb.

In either case, use the PKA Key Token Build verb to create a skeleton key token as input (see “PKA Key Token Build (CSNDPKB)” on page 488).
The PKA Key Import verb uses the clear token from the PKA Key Token Build verb or a clear or encrypted token from the CCA system to securely import the key token into operational form for the coprocessor to use. CCA does not permit the export of the imported PKA key.

The PKA Public Key Extract verb builds a public key token from a private key token.

Application RSA public and private keys can be stored in the PKA key storage file.

**Key identifier for PKA key token**

A *key identifier* for a PKA key token is a variable length (maximum allowed size is 2500 bytes) area that contains either a key label or a key token.

- A **key label** identifies keys that are in the PKA key storage file.
- A **key token** can be either an internal key token, an external key token, or a null key token. Key tokens are generated by an application (for example, using the PKA Key Generate verb), or received from another system that can produce external key tokens.

An **internal key token** can be used only on the local system, because the PKA master key encrypts the key value. Internal key tokens contain keys in operational form only.

An **external key token** can be exchanged with other systems because a transport key that is shared with the other system encrypts the key value. External key tokens contain keys in either exportable or importable form.

A **null key token** consists of eight bytes of binary zeros. The PKA Key Record Create verb can be used to write a null token to the key storage file. This record can subsequently be identified as the target token for the PKA Key Import or PKA Key Generate verb.

The term *key identifier* is used when a parameter could be one of the above items, and indicates that different inputs are possible. For example, you might want to specify a specific parameter as either an internal key token or a key label. The key label is, in effect, an indirect reference to a stored internal key token.

**Key label**

If the first byte of the key identifier is greater than X'20' but less than X'FF', the field is considered to be holding a **key label**.

The contents of a key label are interpreted as the identifier of a key entry in the PKA storage file. The key label is an indirect reference to an internal key token.

If the first byte of the key identifier is X'FF', the identifier is not valid. If the first byte is less than X'20', the identifier is treated as a key token as described below.

A key label is specified on verbs with the *key_identifier* parameter as a 64-byte character string, left-aligned, and padded on the right with blanks. In most cases, the verb does not check the syntax of the key label other than the first byte.

A key label has the following form:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 - 63</td>
<td>64</td>
<td>Key label name</td>
</tr>
</tbody>
</table>
Key token

A key token is a variable length (maximum allowed size is 3500 bytes) field composed of key value and control information.

PKA keys can be either public or private RSA, or ECC keys. Each key token can be either an internal key token (the first byte of the key identifier is X'1F'), an external key token (the first byte of the key identifier is X'1E'), or a null PKA private key token (the first byte of the key identifier is X'00').

See Appendix B, “Key token formats,” on page 609 for descriptions of the PKA key tokens.

Internal key token

An internal key token is a token that can be used only on the system that created it or another system with the same PKA master key.

It contains a key that is encrypted under the PKA master key.

An application obtains an internal key token by using one of the verbs such as those listed below. The verbs are described in detail in Chapter 12, “Managing PKA cryptographic keys,” on page 477.

- PKA Key Generate
- PKA Key Import

The PKA Key Token Change verb can re-encipher private internal tokens from encryption under the old ASYM-MK to encryption under the current ASYM-MK. PKDS Reencipher/Activate options are available to re-encipher RSA and ECC internal tokens in the PKDS when the SYM-MK/ASYM-MK (or APKA-MK) keys are changed.

PKA master keys cannot be changed dynamically.

For debugging information, see Appendix B, “Key token formats,” on page 609 for the format of an internal key token.

External key token

If the first byte of the key identifier is X'1E', the key identifier is interpreted as an external key token.

An external PKA key token contains key (possibly encrypted) and control information. By using the external key token, you can exchange keys between systems.

An application obtains the external key token by using one of the verbs such as those listed below. They are described in detail in Chapter 12, “Managing PKA cryptographic keys,” on page 477.

- PKA Public Key Extract
- PKA Key Token Build
- PKA Key Generate

For debugging information, see Appendix B, “Key token formats,” on page 609 for the format of an external key token.
**Null key token**

If the first byte of the key identifier is X'00', the key identifier is interpreted as a null key token.

For debugging information, see Appendix B, “Key token formats,” on page 609 for the format of a null key token.

**Summary of the PKA verbs**

The table in this section lists the PKA verbs, described in this document, and their corresponding verb names.

The PKA verb names start with CSND. This table also references the topic that describes the verb.

*Table 8. Summary of PKA verbs*

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Verb name</th>
<th>Description</th>
<th>Topic/Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 11, “Using digital signatures,” on page 467</td>
<td>CSNDDSG Digital Signature Generate</td>
<td>Generates a digital signature using an RSA or ECC private key.</td>
<td>&quot;Digital Signature Generate (CSNDDSG)&quot; on page 468</td>
</tr>
<tr>
<td></td>
<td>CSNDDSV Digital Signature Verify</td>
<td>Verifies a digital signature using an RSA or ECC public key.</td>
<td>&quot;Digital Signature Verify (CSNDDSV)&quot; on page 473</td>
</tr>
<tr>
<td>Chapter 12, “Managing PKA cryptographic keys,” on page 477</td>
<td>CSNDPKG PKA Key Generate</td>
<td>Generates an RSA key pair.</td>
<td>&quot;PKA Key Generate (CSNDPKG)&quot; on page 478</td>
</tr>
<tr>
<td></td>
<td>CSNDPKI PKA Key Import</td>
<td>Imports a key token containing either a clear key or an RSA or ECC key enciphered under a transport key.</td>
<td>&quot;PKA Key Import (CSNDPKI)&quot; on page 484</td>
</tr>
<tr>
<td></td>
<td>CSNDPKB PKA Key Token Build</td>
<td>Creates an external PKA key token containing a clear private RSA key. Using this token as input to the PKA Key Import verb returns an operational internal token containing an enciphered private key. Using PKA Key Token Build on a clear public RSA key, returns the public key in a token format that other PKA verbs can directly use. PKA Key Token Build can also be used to create a skeleton token for input to the PKA Key Generate verb for the generation of an internal RSA key token.</td>
<td>&quot;PKA Key Token Build (CSNDPKB)&quot; on page 488</td>
</tr>
<tr>
<td></td>
<td>CSNDKTC PKA Key Token Change</td>
<td>Changes PKA key tokens from encipherment with the old asymmetric-keys master key to encipherment with the current asymmetric-keys master key. This verb changes only private internal tokens.</td>
<td>&quot;PKA Key Token Change (CSNDKTC)&quot; on page 497</td>
</tr>
<tr>
<td>Entry point</td>
<td>Verb name</td>
<td>Description</td>
<td>Topic/Page</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>CSNDPKT</td>
<td>PKA Key Translate</td>
<td>Translates PKA key tokens from encipherment under the old Asymmetric-Keys Master Key to encipherment under the current Asymmetric-Keys Master Key. This verb changes only Private Internal PKA Key Tokens.</td>
<td>&quot;PKA Key Translate (CSNDPKT)&quot; on page 500</td>
</tr>
<tr>
<td>CSNDPKX</td>
<td>PKA Public Key Extract</td>
<td>Extracts a PKA public key token from a supplied PKA internal or external private key token. Performs no cryptographic verification of the PKA private token.</td>
<td>&quot;PKA Public Key Extract (CSNDPKX)&quot; on page 504</td>
</tr>
<tr>
<td>CSNRKX</td>
<td>Remote Key Export</td>
<td>Secure transport of DES keys using asymmetric techniques from a security module (for example, the CEX*C) to a remote device such as an Automated Teller Machine (ATM).</td>
<td>&quot;Remote Key Export (CSNRKX)&quot; on page 506</td>
</tr>
<tr>
<td>CSNTBC</td>
<td>Trusted Block Create</td>
<td>Creates an external trusted block under dual control. A trusted block is an extension of CCA PKA key tokens using new section identifiers.</td>
<td>&quot;Trusted Block Create (CSNTBC)&quot; on page 512</td>
</tr>
</tbody>
</table>
Chapter 4. TR-31 symmetric-key management

This chapter is dedicated to X9 TR-31.

X9 TR-31 is defined in X9 TR-31 2010: Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms. This chapter should be considered an extension of Chapter 6, “Managing AES and DES cryptographic keys,” on page 127. For additional information on symmetric keys, including DES control vectors, see Chapter 6, “Managing AES and DES cryptographic keys,” on page 127.

The TR-31 key block is a format defined by the ANSI Standards Committee to support interchange of symmetric keys in a secure manner and with key attributes included in the exchanged data. Currently, this format supports only DES keys. AES keys are not supported.

TR-31 is a Technical Report. This is different from a standard, which is a mandatory set of rules that must be followed. A Technical Report is not mandatory, but provides guidance to those who are using the standards. In this case, TR-31 is a companion to the standard X9.24-1, which defines requirements for key management performed using symmetric key techniques. TR-31 shows a method that complies with the various requirements that are defined in X9.24-1, and because no other specific method has been defined by the standards committee, the TR-31 method is becoming the apparent standard through which financial organizations will exchange keys.

Prior to TR-31, there were problems with the interchange of symmetric keys. In the banking environment, it is very important that each symmetric key have a specific set of attributes attached to it, specifying such things as the cryptographic operations for which that key can be used. CCA implements these attributes in the form of the control vector (CV), but other vendors implement attributes in their own proprietary ways. Thus, if you are exchanging keys between CCA systems, you can securely pass the attributes using CCA functions and data structures. If, however, that same key were sent to a non-CCA system, there would be no secure way to do that. This is because the two cryptographic architectures have no common key format that could be used to pass both the key and its attributes. As a result, the normal approach has been to strip the attributes and send just the encrypted key, then attach attributes again at the receiving end.

The above scenario has major security problems because it allows an insider to obtain the key without its designated attributes. The insider can then attach other attributes to it, thereby compromising the security of the system. For example, assume the exchanged key is a key-encrypting key (KEK). The attributes of a KEK should restrict its use to key management functions that are designed to prevent exposure of the keys that the KEK is used to encrypt. If that KEK is transmitted without any attributes, an attacker on the inside can turn the key into a type used for data decryption. Such a key can then be used to decipher all of the keys that were previously protected using the KEK. It is clearly very desirable to have a way of exchanging keys that prevents this modification of the attributes. TR-31 provides such a method.

The TR-31 key block has a set of defined key attributes. These attributes are securely bound to the key so that they can be transported together between any two systems that both understand the TR-31 format. This is much of the reason for
its gain in popularity. There are two supported cryptographic methods for
protecting the key block. The original version of TR-31 defined a method that
encrypted the key field in CBC mode and computed a TDES MAC over the header
and key field. The encryption and MAC operations used different keys, created by
applying predefined variants to the input key block protection key. This method is
identified by a Key Block Version ID value of "A" (X'41'). An update to TR-31 adds
a more modern method, identified by a Key Block Version ID value of "B" (X'42')
or "C" (X'43'). The "B" method uses an authenticated encryption scheme and uses
cryptographic key derivation methods to produce the encryption and MAC keys.
The "C" method is exactly the same as the "A" method in terms of wrapping keys.
However, the field values are expected to conform to the updated standard.

Not surprisingly, TR-31 uses some key attributes that are different from those in
the CCA control vector. In some cases, there is a one-to-one correspondence
between CCA and TR-31 attributes. For these cases, conversion is simple and
straightforward. In other cases, the correspondence is one-to-many or many-to-one
and the application program must provide information to help the CCA verbs
decide how to perform the translation between CCA and TR-31 attributes. There
are also CCA attributes that simply cannot be represented using TR-31. CCA keys
with those attributes are not eligible for conversion to TR-31 format.

The TR-31 key block has these two important features:

1. The key is protected in such a way that it meets the "key bundling"
   requirements of various standards. These standards state that the individual
   8-byte blocks of a double-length or triple-length TDES key must be bound in
   such a way that they cannot be individually manipulated. TR-31 accomplishes
   this mainly by computation of a MAC across the entire structure, excluding the
   MAC value itself.

2. Key usage attributes, defined to control how the key can be used, are securely
   bound to the key itself. This makes it possible for a key and its attributes to be
   securely transferred from one party to another while assuring that the attributes
   of the key cannot be modified to suit the needs of an attacker.

CCA support the management of DES keys using TR-31. Table 9 lists the verbs

Table 9. TR-31 symmetric key management verbs

<table>
<thead>
<tr>
<th>Verb</th>
<th>Page</th>
<th>Service</th>
<th>Entry point</th>
<th>Service location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Export to TR31</td>
<td>* &quot;Key Export to TR31*</td>
<td>Exports a CCA external or internal fixed-length symmetric key-token,</td>
<td>CSNB3I</td>
<td>cryptographic</td>
</tr>
<tr>
<td></td>
<td><em>(CSNB3I)</em></td>
<td>converting it into an external X9 TR-31 key block format.</td>
<td></td>
<td>engine</td>
</tr>
<tr>
<td>TR31 Key Import</td>
<td>&quot;TR31 Key Import*</td>
<td>Imports an external X9 TR-31 key block,</td>
<td>CSNB3I</td>
<td>cryptographic</td>
</tr>
<tr>
<td></td>
<td><em>(CSNB3I)</em></td>
<td>converting it into a CCA external or internal fixed-length symmetric</td>
<td></td>
<td>engine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>key-token.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR31 Key Token Parse</td>
<td>&quot;TR31 Key Token Parse*</td>
<td>Parses the information from the standard predefined fields of the TR-31</td>
<td>CSNB3IP</td>
<td>security API</td>
</tr>
<tr>
<td></td>
<td><em>(CSNB3IP)</em></td>
<td>key block header without importing the key.</td>
<td></td>
<td>host software</td>
</tr>
</tbody>
</table>
Table 9. TR-31 symmetric key management verbs (continued)

<table>
<thead>
<tr>
<th>Verb</th>
<th>Page</th>
<th>Service</th>
<th>Entry point</th>
<th>Service location</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR31 Optional Data Build</td>
<td>&quot;TR31 Optional Data Build (CSNBT31O)&quot; on page 583</td>
<td>Constructs the optional blocks of a TR-31 key block, one block at a time.</td>
<td>CSNBT31O</td>
<td>security API host software</td>
</tr>
<tr>
<td>TR31 Optional Data Read</td>
<td>&quot;TR31 Optional Data Read (CSNBT31R)&quot; on page 587</td>
<td>Obtains the contents of any optional fields of a TR-31 key block header.</td>
<td>CSNBT31R</td>
<td>security API host software</td>
</tr>
</tbody>
</table>
Part 2. CCA verbs

This section of the document introduces AES, DES and PKA verbs.

It includes the following chapters:

- **Chapter 5, “Using the CCA nodes and resource control verbs,” on page 77** describes using the CCA resource control verbs.
- **Chapter 9, “Key storage mechanisms,” on page 345** describes the use of key storage, key tokens, and associated verbs.
- **Chapter 6, “Managing AES and DES cryptographic keys,” on page 127** describes the verbs for generating and maintaining AES, DES, and HMAC cryptographic keys, the Random Number Generate verb (which generates 8-byte random numbers), the Random Number Generate Long verb (which generates up to 8192 bytes of random content), and the Secure Sockets Layer (SSL) security protocol. This chapter also describes utilities to build DES and AES tokens, generate and translate control vectors, and describes the PKA verbs that support DES and AES key distribution.
- **Chapter 7, “Protecting data,” on page 285** describes the verbs for enciphering and deciphering data.
- **Chapter 8, “Verifying data integrity and authenticating messages,” on page 313** describes the verbs for generating and verifying Message Authentication Codes (MACs), generating Modification Detection Codes (MDCs) and generating hashes (SHA-1, MD5, RIPEMD-160).
- **Chapter 10, “Financial services,” on page 393** describes the verbs for use in support of finance-industry applications. This includes several categories.
  - Verbs for generating, verifying, and translating personal identification numbers (PINS).
  - Verbs that generate and verify VISA card verification values and American Express card security codes.
  - Verbs to support smart card applications using the EMV (Europay MasterCard Visa) standards.
- **Chapter 11, “Using digital signatures,” on page 467** describes the verbs that support using digital signatures to authenticate messages.
- **Chapter 12, “Managing PKA cryptographic keys,” on page 477** describes the verbs that generate and manage PKA keys.
- **Chapter 13, “TR-31 symmetric key management verbs,” on page 523** described the verbs that manage TR-31 functions.
Chapter 5. Using the CCA nodes and resource control verbs

This section describes CCA nodes and how to use resource control verbs.

The following verbs are described:
- "Cryptographic Facility Query (CSUACFQ)"
- "Cryptographic Facility Version (CSUACFV)" on page 108
- "Cryptographic Resource Allocate (CSUACRA)" on page 110
- "Cryptographic Resource Deallocate (CSUACRD)" on page 113
- "Key Storage Initialization (CSNBKSI)" on page 116
- "Master Key Process (CSNBMKP)" on page 119
- "Random Number Tests (CSUARNT)" on page 124

Cryptographic Facility Query (CSUACFQ)

The Cryptographic Facility Query verb is used to retrieve information about the coprocessor and the CCA application program in that coprocessor.

This information includes the following:
- General information about the coprocessor, its operating system, and CCA application
- The Environment Identifier (EID)
- Diagnostic information from the coprocessor
- Export-control information from the coprocessor
- Time and date information from the coprocessor
- The contents and size of the authorized PIN decimalization tables loaded onto the coprocessor

On input, you specify:
- A rule_array_count of 1 or 2
- Optionally, a rule_array keyword of ADAPTER1 (for backward compatibility)
- The class of information queried with a rule_array keyword

This verb returns information elements in the rule_array and sets the rule_array_count variable to the number of returned elements.

Determining if a card is a CEX2C or CEX3C or a CEX4C

Using Cryptographic Facility Query, the output rule_array for option STATCCA is the most accurate way to determine Cryptographic Coprocessor type.

- If first two characters of the CCA application version field are 'z' followed by '3', then this card is a CEX2C adapter.

An updated device driver might not be available yet for all distributions where this RPM is usable. The CCA host library uses this mechanism to determine card version, and we recommend here that the application developer also use this method. Where this output and the device driver disagree about the version of a particular card, it is the device driver that will be out of date because the Cryptographic Facility Query data is not interpreted in any way; it comes direct from the adapter.
Cryptographic Facility Query (CSUACFQ)

- If first character of the CCA application version field is a number, such as '4' or greater, then this card is not a CEX2C. For example, a '4' in the first character indicates a CEX3C or CEX4C.

- The results of this query come directly from the card itself. If the host device driver is not up to date, it could incorrectly identify a CEX3C or CEX4C as a CEX2C. Therefore, looking at the CCA application version field for the output rule_array for option STATCCA resolves all questions.

The commands `ivp.e` and `panel.exe -x` will also tell you Cryptographic Coprocessor type, by calling the Cryptographic Facility Query verb for all available adapters.

For details about panel.exe, see “The panel.exe utility” on page 773.

Format

The format of CSUACFQ.

```c
CSUACFQ(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    verb_data_length,
    verb_data)
```

Parameters

The parameter definitions for CSUACFQ.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**rule_array_count**

- **Direction:** Input/Output
- **Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. On input, this value must be 1 or 2.

On output, the verb sets the variable to the number of `rule_array` elements it returns to the application program.

**Tip:** With this verb, the number of returned `rule_array` elements can exceed the `rule_array_count` you specified on input. Be sure you allocate adequate memory to receive all the information elements according to the information class you select on input with the information-to-return keyword in the `rule_array`.

**rule_array**

- **Direction:** Input/Output
- **Type:** Array
Cryptographic Facility Query (CSUACFQ)

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters.

On input, set the *rule_array* to specify the type of information to retrieve. There are two input *rule_array* elements, as described in Table 10. This table also indicates to which parameter, *rule_array* or *verb_data* output data is returned for each keyword.

**Table 10. Keywords for Cryptographic Facility Query control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Output in <em>rule_array</em></th>
<th>Output in <em>verb_data</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adapter to use (Optional)</strong></td>
<td>Ad</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>ADAPTER1</strong></td>
<td>This keyword is ignored. It is accepted for backward compatibility.</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>Information to return (One required)</strong></td>
<td>Get</td>
<td>None</td>
<td>See “GET-UDX” on page 93</td>
</tr>
<tr>
<td>GET-UDX</td>
<td>Obtains UDX identifiers. This keyword applies only when using Linux on IBM System z.</td>
<td>None.</td>
<td>See “GET-UDX” on page 93</td>
</tr>
<tr>
<td>NUM-DECT</td>
<td>Returns the number of bytes of data required for the <em>verb_data</em> variable when the STATDECT <em>rule_array</em> keyword is specified. Note: A TKE is used to securely load PIN decimalization tables.</td>
<td>None.</td>
<td>See “NUM-DECT” on page 93</td>
</tr>
<tr>
<td>QPENDING</td>
<td>TKE uses this <em>rule_array</em> keyword to request information about pending changes previously submitted by this TKE or another TKE to this adapter. Only TKE can submit changes to be stored in the Pending Change Buffer queried with this command. The keyword is available for normal users of Cryptographic Facility Query, for informational or debugging reasons (no secrets are exposed). This keyword applies only when using Linux on IBM System z.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATAES</td>
<td>Obtains status information on AES master-key registers and AES key-length enablement.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATAPKA</td>
<td>Obtains status information on APKA master-key registers and APKA key-length enablement. This keyword was introduced with CCA 4.1.0.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATCARD</td>
<td>Obtains coprocessor-related basic status information. This keyword is provided for backwards compatibility. The STATCRD2 should be used instead of STATCARD.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATCCA</td>
<td>Obtains CCA-related status information.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATCCAE</td>
<td>Obtains CCA-related extended status information.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATCRD2</td>
<td>Obtains extended basic status information about the coprocessor.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATDECT</td>
<td>Obtains the information on all of the authorized PIN decimalization tables that are currently stored on the coprocessor. Output is returned in the <em>verb_data</em> variable. Note: A TKE is used to securely load PIN decimalization tables.</td>
<td>None.</td>
<td>See “STATDECT” on page 93</td>
</tr>
<tr>
<td>STATDIAG</td>
<td>Obtains diagnostic information.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>Keyword</td>
<td>Description</td>
<td>Output in <code>rule_array</code></td>
<td>Output in <code>verb_data</code></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>STATEID</td>
<td>Obtains the Environment Identifier (EID).</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATEXPT</td>
<td>Obtains function control vector-related status information.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATICSA</td>
<td>Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. This keyword applies only when using Linux on IBM System z.</td>
<td>See Table 11 on page 81</td>
<td>See “STATICSA” on page 94</td>
</tr>
<tr>
<td>STATICSB</td>
<td>Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. See “STATICSB” on page 96. This keyword was introduced with CCA 4.1.0. This keyword applies only when using Linux on IBM System z.</td>
<td>See Table 11 on page 81</td>
<td>See “STATICSB” on page 96</td>
</tr>
<tr>
<td>STATICSE</td>
<td>Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. This keyword applies only when using Linux on IBM System z.</td>
<td>See Table 11 on page 81</td>
<td>See “STATICSE” on page 99.</td>
</tr>
<tr>
<td>STATICSF</td>
<td>This keyword returns the adapter serial number and status information about the SYM (DES) and ASYM (RSA) master-key registers, including whether a valid key is present in each of the old, current, and new registers. This keyword applies only when using Linux on IBM System z.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATICSX</td>
<td>Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. This keyword applies only when using Linux on IBM System z.</td>
<td>See Table 11 on page 81</td>
<td>See “STATICSX” on page 101</td>
</tr>
<tr>
<td>STATKPR</td>
<td>Obtains non-secret information about an operational key part.</td>
<td>None.</td>
<td>See “STATKPR” on page 103.</td>
</tr>
<tr>
<td>STATKPRL</td>
<td>Obtains the names of the operational key parts.</td>
<td>None.</td>
<td>See “STATKPRL” on page 104</td>
</tr>
<tr>
<td>STATMOFN</td>
<td>Obtains master-key shares distribution information.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>STATVKPL</td>
<td>Obtains the names of all the operational key parts for variable length key token preparation. This keyword applies only when using Linux on IBM System z.</td>
<td>None.</td>
<td>See “STATVKPL” on page 104</td>
</tr>
<tr>
<td>STATVKPR</td>
<td>Obtains non-secret information about an operational key part. This is different from STATKPR in that a register for creating a key in a variable length key token is described. This keyword applies only when using Linux on IBM System z.</td>
<td>None.</td>
<td>See “STATVKPR” on page 104</td>
</tr>
<tr>
<td>TIMEDATE</td>
<td>Reads the current date, time, and day of the week from the secure clock within the coprocessor.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
<tr>
<td>TKESTATE</td>
<td>Indicates whether TKE access is enabled or not.</td>
<td>See Table 11 on page 81</td>
<td>None.</td>
</tr>
</tbody>
</table>
Table 10. Keywords for Cryptographic Facility Query control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Output in rule_array</th>
<th>Output in verb_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAPMTHD</td>
<td>Obtains the default key wrapping method. This keyword was introduced with CCA 4.1.0.</td>
<td>See Table 11</td>
<td>None.</td>
</tr>
</tbody>
</table>

Different sets of rule_array elements are returned, depending on the input keyword. Table 11 describes these rule_array elements for keywords that result in output data in the rule_array parameter.

For rule_array elements that contain numbers, those numbers are represented by numeric characters which are left-aligned and padded on the right with space characters. For example, a rule_array element that contains the number 2 contains the character string “2   ” (the number 2 followed by seven space characters).

For some keywords, there is output data in the verb_data variable. This output data is described in a separate section for each keyword in Verb data returned for Cryptographic Facility Query rule_array keywords on IBM System z on page 93.

Table 11. Cryptographic Facility Query information returned in the rule_array

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output rule_array for option QPENDING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Change type (ASCII number)</td>
<td>An ASCII number that indicates the type of pending change stored in the adapter (if there is one)</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>No pending change</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Role load</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Profile load</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Role delete</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Profile delete</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Domain zeroize</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Enable</td>
</tr>
<tr>
<td>2</td>
<td>user ID (string)</td>
<td>A string of eight ASCII characters for the user ID of the user who initiated the pending change.</td>
</tr>
</tbody>
</table>

Output rule_array for option STATAES

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AES NMK status</td>
<td>State of the AES new master key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Register is clear</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Register contains a partially complete key</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Register contains a complete key</td>
</tr>
<tr>
<td>2</td>
<td>AES CMK status</td>
<td>State of the AES current master key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Register is clear</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Register contains a key</td>
</tr>
<tr>
<td>3</td>
<td>AES OMK status</td>
<td>State of the AES old master key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Register is clear</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Register contains a key</td>
</tr>
</tbody>
</table>
## Cryptographic Facility Query (CSUACFQ)

### Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>AES key length enablement</td>
<td>The maximum AES key length that is enabled by the function control vector. The value is 0 (if no AES key length is enabled in the FCV), 128, 192, or 256.</td>
</tr>
</tbody>
</table>

**Output rule_array for option STATAPKA**

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ECC NMK status</td>
<td>The state of the ECC new master key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>ECC CMK status</td>
<td>The state of the ECC current master key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>ECC OMK status</td>
<td>The state of the ECC old master key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>ECC key length enablement</td>
<td>The maximum ECC curve size that is enabled by the function control vector. The value will be 0 (if no ECC keys are enabled in the FCV) and 521 for the maximum size.</td>
</tr>
</tbody>
</table>

**Output rule_array for option STATCARD**

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of installed adapters</td>
<td>A numeric character string containing the number of active coprocessors installed in the machine. This includes only coprocessors that have CCA software loaded (including those with CCA UDX software). Non-CCA coprocessors are not included in this number.</td>
</tr>
<tr>
<td>2</td>
<td>DES hardware level</td>
<td>A numeric character string containing an integer value identifying the version of DES hardware on the coprocessor.</td>
</tr>
<tr>
<td>3</td>
<td>RSA hardware level</td>
<td>A numeric character string containing an integer value identifying the version of RSA hardware on the coprocessor.</td>
</tr>
<tr>
<td>4</td>
<td>POST version</td>
<td>A character string identifying the version of the coprocessor's Power-On Self Test (POST) firmware. The first four characters define the POST0 version and the last four characters define the POST1 version.</td>
</tr>
<tr>
<td>5</td>
<td>Coprocessor operating system name</td>
<td>A character string identifying the operating system firmware on the coprocessor.</td>
</tr>
<tr>
<td>6</td>
<td>Coprocessor operating system version</td>
<td>A character string identifying the version of the coprocessor's operating system firmware.</td>
</tr>
<tr>
<td>7</td>
<td>Coprocessor part number</td>
<td>A character string containing the 8 character part number identifying the version of the coprocessor.</td>
</tr>
<tr>
<td>8</td>
<td>Coprocessor EC level</td>
<td>A character string containing the 8 character engineering change (EC) level for this version of the coprocessor.</td>
</tr>
</tbody>
</table>
### Cryptographic Facility Query (CSUACFQ)

**Table 11. Cryptographic Facility Query information returned in the rule_array (continued)**

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Miniboot version</td>
<td>A character string identifying the version of the coprocessor's miniboot firmware. This firmware controls the loading of programs into the coprocessor. The first four characters define the MiniBoot0 version and the last four characters define the MiniBoot1 version.</td>
</tr>
<tr>
<td>10</td>
<td>CPU speed</td>
<td>A numeric character string containing the operating speed of the microprocessor chip, in megahertz.</td>
</tr>
<tr>
<td>11</td>
<td>Adapter ID (see also element number 15)</td>
<td>A unique identifier manufactured into the coprocessor. The coprocessor adapter ID is an 8-byte binary value.</td>
</tr>
<tr>
<td>12</td>
<td>Flash memory size</td>
<td>A numeric character string containing the size of the flash EPROM memory on the coprocessor, in 64 KB increments.</td>
</tr>
<tr>
<td>13</td>
<td>DRAM memory size</td>
<td>A numeric character string containing the size of the dynamic RAM (DRAM) memory on the coprocessor, in kilobytes.</td>
</tr>
<tr>
<td>14</td>
<td>Battery-backed memory size</td>
<td>A numeric character string containing the size of the battery-backed RAM on the coprocessor, in kilobytes.</td>
</tr>
<tr>
<td>15</td>
<td>Serial number</td>
<td>A character string containing the unique serial number of the coprocessor. The serial number is factory installed.</td>
</tr>
</tbody>
</table>

**Output rule_array for option STATCCA**

<table>
<thead>
<tr>
<th>1</th>
<th>NMK status</th>
<th>The state of the new master-key register:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>The register is clear.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>The register contains a partially complete key.</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>The register contains a key.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>CMK status</th>
<th>The state of the current master-key register:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>The register is clear.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>The register contains a key.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>OMK status</th>
<th>The state of the old master-key register:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>The register is clear.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>The register contains a key.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>CCA application version</th>
<th>A character string that identifies the version of the CCA application program running in the coprocessor. The results of this query come directly from the card itself. If the host device driver is not up to date, it could incorrectly identify a CEX* as a CEX2C. Therefore, looking at this field resolves all questions.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>If first two characters are 'z' followed by '3', then this card is a CEX2C adapter (no matter what device driver indicates).</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>If first character is a number, such as '4' or greater, then this card is not a CEX2C. For example, a '4' in the first character indicates a CEX*C.</td>
</tr>
</tbody>
</table>

| 5              | CCA application build date                | A character string containing the build date for the CCA application program running in the coprocessor.                                                                                                   |

| 6              | User role                                 | A character string containing the role identifier which defines the host application user's current authority.                                                                                           |
**Cryptographic Facility Query (CSUACFQ)**

Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Symmetric NMK status</td>
<td>The state of the symmetric new master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Symmetric CMK status</td>
<td>The state of the symmetric current master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Symmetric OMK status</td>
<td>The state of the symmetric old master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>CCA application version</td>
<td>A character string that identifies the version of the CCA application program that is running in the coprocessor.</td>
</tr>
<tr>
<td>5</td>
<td>CCA application build date</td>
<td>A character string containing the build date for the CCA application program that is running in the coprocessor.</td>
</tr>
<tr>
<td>6</td>
<td>User role</td>
<td>A character string containing the role identifier which defines the host application user's current authority.</td>
</tr>
<tr>
<td>7</td>
<td>Asymmetric NMK status</td>
<td>The state of the asymmetric new master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Asymmetric CMK status</td>
<td>The state of the asymmetric current master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Asymmetric OMK status</td>
<td>The state of the asymmetric old master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

*Output rule_array for option STATCRD2*

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of installed adapters</td>
<td>A numeric character string containing the number of active coprocessors installed in the machine. This includes only coprocessors that have CCA software loaded (including those with CCA UDX software). Non-CCA coprocessors are not included in this number.</td>
</tr>
<tr>
<td>2</td>
<td>DES hardware level</td>
<td>A numeric character string containing an integer value identifying the version of DES hardware on the coprocessor.</td>
</tr>
<tr>
<td>3</td>
<td>RSA hardware level</td>
<td>A numeric character string containing an integer value identifying the version of RSA hardware on the coprocessor.</td>
</tr>
<tr>
<td>4</td>
<td>POST version</td>
<td>A character string identifying the version of the coprocessor's Power-On Self Test (POST) firmware. The first four characters define the POST0 version and the last four characters define the POST1 version.</td>
</tr>
<tr>
<td>Element number</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5</td>
<td>Coprocessor operating system name</td>
<td>A character string identifying the operating system firmware on the coprocessor.</td>
</tr>
<tr>
<td>6</td>
<td>Coprocessor operating system version</td>
<td>A character string identifying the version of the coprocessor’s operating system firmware.</td>
</tr>
<tr>
<td>7</td>
<td>Coprocessor part number</td>
<td>A character string containing the 8 character part number identifying the version of the coprocessor.</td>
</tr>
<tr>
<td>8</td>
<td>Coprocessor EC level</td>
<td>A character string containing the 8 character engineering change (EC) level for this version of the coprocessor.</td>
</tr>
<tr>
<td>9</td>
<td>Miniboot version</td>
<td>A character string identifying the version of the coprocessor’s miniboot firmware. This firmware controls the loading of programs into the coprocessor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The first four characters define the MiniBoot0 version and the last four characters define the MiniBoot1 version.</td>
</tr>
<tr>
<td>10</td>
<td>CPU speed</td>
<td>A numeric character string containing the operating speed of the microprocessor chip, in megahertz.</td>
</tr>
<tr>
<td>11</td>
<td>Adapter ID (see also element number 15)</td>
<td>A unique identifier manufactured into the coprocessor. The coprocessor adapter ID is an 8-byte binary value.</td>
</tr>
<tr>
<td>12</td>
<td>Flash memory size</td>
<td>A numeric character string containing the size of the flash EPROM memory on the coprocessor, in 64 KB increments.</td>
</tr>
<tr>
<td>13</td>
<td>DRAM memory size</td>
<td>A numeric character string containing the size of the dynamic RAM (DRAM) memory on the coprocessor, in kilobytes.</td>
</tr>
<tr>
<td>14</td>
<td>Battery-backed memory size</td>
<td>A numeric character string containing the size of the battery-backed RAM on the coprocessor, in kilobytes.</td>
</tr>
<tr>
<td>15</td>
<td>Serial number</td>
<td>A character string containing the unique serial number of the coprocessor. The serial number is factory installed.</td>
</tr>
<tr>
<td>16</td>
<td>POST2 version</td>
<td>A character string identifying the version of the coprocessor’s POST2 firmware. The first four characters define the POST2 version, and the last four characters are reserved and valued to space characters.</td>
</tr>
</tbody>
</table>

**Output rule_array for option STATDIAG**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The battery is good.</td>
</tr>
<tr>
<td>2</td>
<td>The battery should be replaced.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The latch is cleared.</td>
</tr>
<tr>
<td>2</td>
<td>The latch is set.</td>
</tr>
</tbody>
</table>
## Cryptographic Facility Query (CSUACFQ)

### Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Error log status A numeric character string containing a value which indicates whether there is data in the coprocessor CCA error log:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>The error log is empty.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>The error log contains abnormal termination data, but is not yet full.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>The error log is full and cannot hold any more data.</td>
</tr>
<tr>
<td>4</td>
<td>Mesh intrusion A numeric character string containing a value to indicate whether the coprocessor has detected tampering with the protective mesh that surrounds the secure module. This indicates a probable attempt to physically penetrate the module:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>No intrusion has been detected.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>An intrusion attempt has been detected.</td>
</tr>
<tr>
<td>5</td>
<td>Low voltage detected A numeric character string containing a value to indicate whether a power-supply voltage was below the minimum acceptable level. This might indicate an attempt to attack the security module:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Only acceptable voltages have been detected.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A voltage has been detected below the low-voltage tamper threshold.</td>
</tr>
<tr>
<td>6</td>
<td>High voltage detected A numeric character string containing a value indicates whether a power-supply voltage was greater than the maximum acceptable level. This might indicate an attempt to attack the security module:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Only acceptable voltages have been detected.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>A voltage has been detected greater than the high-voltage tamper threshold.</td>
</tr>
<tr>
<td>7</td>
<td>Temperature range exceeded A numeric character string containing a value to indicate whether the temperature in the secure module was outside of the acceptable limits. This might indicate an attempt to attack the security module:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>The temperature is acceptable.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>The temperature has been detected outside of an acceptable limit.</td>
</tr>
<tr>
<td>8</td>
<td>Radiation detected A numeric character string containing a value to indicate whether radiation was detected inside the secure module. This might indicate an attempt to attack the security module:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>No radiation has been detected.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Radiation has been detected.</td>
</tr>
<tr>
<td>9, 11, 13, 15, 17</td>
<td>Last 5 commands run These five rule_array elements contain the last five commands that were run by the coprocessor CCA application. They are in chronological order, with the most recent command in element 9. Each element contains the security API command code in the first four characters and the subcommand code in the last four characters.</td>
<td></td>
</tr>
<tr>
<td>10, 12, 14, 16, 18</td>
<td>Last 5 return codes These five rule_array elements contain the security API return codes and reason codes corresponding to the five commands in rule_array elements 9, 11, 13, 15, and 17. Each element contains the return code in the first four characters and the reason code in the last four characters.</td>
<td></td>
</tr>
</tbody>
</table>

Output rule_array for option STATEID
<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>EID</td>
<td>The two elements, when concatenated, provide the 16-byte Environment Identifier (EID) value.</td>
</tr>
</tbody>
</table>

**Output rule_array for option STATEXPT**

<table>
<thead>
<tr>
<th>1</th>
<th>Base CCA services availability</th>
<th>A numeric character string containing a value to indicate whether base CCA services are available:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Base CCA services are not available.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Base CCA services are available.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>56-bit DES availability</th>
<th>A numeric character string containing a value to indicate whether 56-bit DES encryption is available:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>56-bit DES encryption is not available.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>56-bit DES encryption is available.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Triple-DES availability</th>
<th>A numeric character string containing a value to indicate whether Triple-DES encryption is available:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Triple-DES encryption is not available.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Triple-DES encryption is available.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>SET services availability</th>
<th>A numeric character string containing a value to indicate whether SET (secure electronic transaction) services are available:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>SET services are not available.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>SET services are available.</td>
</tr>
</tbody>
</table>

**Note:** The SET services are not supported in the Linux on IBM System z environment.

| 6               | Maximum modulus for symmetric key encryption | A numeric character string containing the maximum modulus size enabled for the encryption of symmetric keys. This defines the longest public-key modulus that can be used for key management of symmetric-algorithm keys. |

**Output rule_array for option STATICSA**

This keyword also has verb data returned in the verb_data field. See "Verb data returned for Cryptographic Facility Query rule_array keywords on IBM System z" on page 93.

<table>
<thead>
<tr>
<th>1</th>
<th>Card serial number</th>
<th>Eight ASCII characters for the adapter serial number</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>DES new master-key register state</th>
<th>An ASCII number showing the state of the DES new master-key register:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Empty</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Partially full</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Full</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>DES current master-key register state</th>
<th>An ASCII number showing the state of the DES current master-key register:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Invalid</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Valid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>DES old master-key register state</th>
<th>An ASCII number showing the state of the DES old master-key register:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Invalid</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Valid</td>
</tr>
</tbody>
</table>
### Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>PKA new master-key register state</td>
<td>An ASCII number showing the state of the PKA new master-key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1 Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Partially full</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 Full</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PKA current master-key register state</td>
<td>An ASCII number showing the state of the PKA current master-key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PKA old master-key register state</td>
<td>An ASCII number showing the state of the PKA old master-key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>AES new master-key register state</td>
<td>An ASCII number showing the state of the AES new master-key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1 Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Partially full</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 Full</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>AES current master-key register state</td>
<td>An ASCII number showing the state of the AES current master-key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>AES old master-key register state</td>
<td>An ASCII number showing the state of the AES old master-key register:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
</tbody>
</table>

**Output rule_array for option STATICSB**

This keyword also has verb data returned in the verb_data field. See “Verb data returned for Cryptographic Facility Query rule_array keywords on IBM System z” on page 93.

| 1              | Card serial number                        | Eight ASCII characters for the adapter serial number                        |
| 2              | DES new master-key register state         | An ASCII number showing the state of the DES new master-key register:       |
|                | Value                                      | Description                                                                 |
|                | 1 Empty                                    |                                                                              |
|                | 2 Partially full                           |                                                                              |
|                | 3 Full                                     |                                                                              |
| 3              | DES current master-key register state      | An ASCII number showing the state of the DES current master-key register:   |
|                | Value                                      | Description                                                                 |
|                | 1 Invalid                                  |                                                                              |
|                | 2 Valid                                    |                                                                              |
| 4              | DES old master-key register state          | An ASCII number showing the state of the DES old master-key register:       |
|                | Value                                      | Description                                                                 |
|                | 1 Invalid                                  |                                                                              |
|                | 2 Valid                                    |                                                                              |
### Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>PKA new master-key register state</td>
<td>An ASCII number showing the state of the PKA new master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1 Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Partially full</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 Full</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PKA current master-key register state</td>
<td>An ASCII number showing the state of the PKA current master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PKA old master-key register state</td>
<td>An ASCII number showing the state of the PKA old master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>APKA new master-key register state</td>
<td>An ASCII number showing the state of the APKA new master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1 Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Partially full</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 Full</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>APKA current master-key register state</td>
<td>An ASCII number showing the state of the APKA current master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>APKA old master-key register state</td>
<td>An ASCII number showing the state of the APKA old master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
</tbody>
</table>

**Output rule_array for option STATICSE**

This keyword also has verb data returned in the *verb data* field. See "Verb data returned for Cryptographic Facility Query rule_array keywords on IBM System z" on page 93.

| 1              | Card serial number                            | Eight ASCII characters for the adapter serial number                        |
| 2              | DES new master-key register state             | An ASCII number showing the state of the DES new master-key register:       |
|                | **Value**                                      | **Description**                                                             |
|                | 1 Empty                                        |                                                                             |
|                | 2 Partially full                               |                                                                             |
|                | 3 Full                                         |                                                                             |
| 3              | DES current master-key register state         | An ASCII number showing the state of the DES current master-key register:   |
|                | **Value**                                      | **Description**                                                             |
|                | 1 Invalid                                      |                                                                             |
|                | 2 Valid                                        |                                                                             |
| 4              | DES old master-key register state             | An ASCII number showing the state of the DES old master-key register:       |
|                | **Value**                                      | **Description**                                                             |
|                | 1 Invalid                                      |                                                                             |
|                | 2 Valid                                        |                                                                             |
### Cryptographic Facility Query (CSUACFQ)

Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>PKA new master-key register state</td>
<td>An ASCII number showing the state of the PKA new master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong> Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Empty</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 Partially full</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 Full</td>
</tr>
<tr>
<td>6</td>
<td>PKA current master-key register state</td>
<td>An ASCII number showing the state of the PKA current master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong> Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 Valid</td>
</tr>
<tr>
<td>7</td>
<td>PKA old master-key register state</td>
<td>An ASCII number showing the state of the PKA old master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong> Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 Valid</td>
</tr>
</tbody>
</table>

**Output rule_array for option STATICSF**

<table>
<thead>
<tr>
<th>1</th>
<th>Card serial number</th>
<th>Eight ASCII characters for the adapter serial number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DES new master-key register state</td>
<td>An ASCII number showing the state of the DES new master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong> Description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Partially full</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 Full</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DES current master-key register state</td>
<td>An ASCII number showing the state of the DES current master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong> Description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>DES old master-key register state</td>
<td>An ASCII number showing the state of the DES old master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong> Description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PKA new master-key register state</td>
<td>An ASCII number showing the state of the PKA new master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong> Description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Partially full</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 Full</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PKA current master-key register state</td>
<td>An ASCII number showing the state of the PKA current master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong> Description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PKA old master-key register state</td>
<td>An ASCII number showing the state of the PKA old master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong> Description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 Invalid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 Valid</td>
<td></td>
</tr>
</tbody>
</table>

**Output rule_array for option STATICSX**

This keyword also has verb data returned in the verb_data field. See “Verb data returned for Cryptographic Facility Query rule_array keywords on IBM System z” on page 93.
## Cryptographic Facility Query (CSUACFQ)

### Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Card serial number</td>
<td>Eight ASCII characters for the adapter serial number</td>
</tr>
<tr>
<td>2</td>
<td>DES new master-key register state</td>
<td>An ASCII number showing the state of the DES new master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Partially full</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DES current master-key register</td>
<td>An ASCII number showing the state of the DES current master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>DES old master-key register state</td>
<td>An ASCII number showing the state of the DES old master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PKA new master-key register state</td>
<td>An ASCII number showing the state of the PKA new master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Partially full</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PKA current master-key register state</td>
<td>An ASCII number showing the state of the PKA current master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PKA old master-key register state</td>
<td>An ASCII number showing the state of the PKA old master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Valid</td>
<td></td>
</tr>
</tbody>
</table>

### Output rule_array for option STATMOFN

Elements 1 and 2 are treated as a 16-byte string, as are elements 3 and 4, with the high-order 15 bytes containing meaningful information and the 16th byte containing a space character. Each byte provides status information about the i-th share, \(1 \leq i \leq 15\), of master-key information.

<table>
<thead>
<tr>
<th>1, 2</th>
<th>Master-key shares generation</th>
<th>The 15 individual bytes are set to one of these character values:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>0</td>
<td>Cannot be generated</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Can be generated</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Has been generated but not distributed</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Generated and distributed once</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Generated and distributed more than once</td>
<td></td>
</tr>
<tr>
<td>3, 4</td>
<td>Master-key shares reception</td>
<td>The 15 individual bytes are set to one of these character values:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>0</td>
<td>Cannot be received</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Can be received</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Has been received</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Has been received more than once</td>
<td></td>
</tr>
</tbody>
</table>
### Table 11. Cryptographic Facility Query information returned in the rule_array (continued)

<table>
<thead>
<tr>
<th>Element number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(m)</td>
<td>The minimum number of shares required to instantiate a master key through the master-key-shares process. The value is returned in two characters, valued from 01 - 15, followed by six space characters.</td>
</tr>
<tr>
<td>6</td>
<td>(n)</td>
<td>The maximum number of distinct shares involved in the master-key shares process. The value is returned in two characters, valued from 01 - 15, followed by six space characters.</td>
</tr>
</tbody>
</table>

**Output rule_array for option TIMEDATE**

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Date</td>
</tr>
<tr>
<td></td>
<td>The current date is returned as a character string of the form YYYYMMDD, where:</td>
</tr>
<tr>
<td></td>
<td>YYYY</td>
</tr>
<tr>
<td></td>
<td>MM</td>
</tr>
<tr>
<td></td>
<td>DD</td>
</tr>
<tr>
<td>2</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>The current UTC time of day is returned as a character string of the form HHMMSS, where:</td>
</tr>
<tr>
<td></td>
<td>HH</td>
</tr>
<tr>
<td></td>
<td>MM</td>
</tr>
<tr>
<td></td>
<td>SS</td>
</tr>
<tr>
<td>3</td>
<td>Day of the week</td>
</tr>
<tr>
<td></td>
<td>The day of the week is returned as a number between 1 (Sunday) and 7 (Saturday).</td>
</tr>
</tbody>
</table>

**Output rule_array for option TKESTATE**

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TKE access enabled</td>
</tr>
<tr>
<td></td>
<td>Indicates whether a TKE can be used to administer this CEX*C. Values are:</td>
</tr>
<tr>
<td></td>
<td>TKEPERM</td>
</tr>
<tr>
<td></td>
<td>TKEDENY</td>
</tr>
</tbody>
</table>

**Output rule_array for option WRAPMTHD**

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Internal tokens</td>
</tr>
<tr>
<td></td>
<td>Default wrapping method for internal tokens.</td>
</tr>
<tr>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>External tokens</td>
</tr>
<tr>
<td></td>
<td>Default wrapping method for external tokens.</td>
</tr>
<tr>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**verb_data_length**

*Direction: Input/Output*

*Type: Integer*

The `verb_data_length` parameter is a pointer to an integer variable containing the number of bytes of data in the `verb-data` variable.

**verb_data**

*Direction: Input/Output*

*Type: String*
Verb data returned for Cryptographic Facility Query *rule_array* keywords on IBM System z

Some keywords return specific data in the *verb_data* parameter, and update the *verb_data_length* field with the count of bytes returned.

The *verb_data* buffer must be large enough to receive the data (see keyword-specific sizes below) and the *verb_data_length* parameter as passed in to Cryptographic Facility Query (CSUACFQ) must indicate that size (or a larger value). If either the *verb_data* or *verb_data_length* fields are not valid, there will be no data returned at all. In this case, a return code of 8 and a reason code of 72 will be returned.

**GET-UDX**

This *rule_array* keyword causes a variable length list of 2-byte UDX identifiers to be returned.

The identifiers represent the authorized UDX verb IDs for the adapter. A UDX is a set of one or more custom CCA APIs added to the adapter, using the installable code feature. Unless the programming source has also provided an updated host library, these UDX calls will not be accessible from the IBM System z Linux host library. If an updated host library is provided, refer to the accompanying documentation for usage.

The maximum number of names to be returned is 100. Using this number, the maximum size buffer is 6400 bytes.

**NUM-DECT**

This *rule_array* keyword causes a 4-byte binary number to be returned.

This is the number of bytes required for the *verb_data* variable when the STATDECT rule-array keyword is specified.

**STATDECT**

This *rule_array* keyword causes a table of up to 100 PIN decimalization tables to be returned.

Table 12 shows the data format for a decimalization table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>3</td>
<td>PIN decimalization table identifier in ASCII digits 001 - 100 (X’303031’ - X’313030’).</td>
</tr>
<tr>
<td>state</td>
<td>1</td>
<td>Table state in ASCII:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Code</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A (X’41’)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L (X’4C’)</td>
</tr>
<tr>
<td>table</td>
<td>16</td>
<td>PIN decimalization table. Contains ASCII digits 0 - 9 (X’30’ - X’39’), in the clear.</td>
</tr>
<tr>
<td>Total byte count</td>
<td>Depends on the number of returned decimalization tables. There are 20 byte for each decimalization table. The total byte count is returned for the NUM-DECT keyword.</td>
<td></td>
</tr>
</tbody>
</table>
Cryptographic Facility Query (CSUACFQ)

**STATICSA**

This `rule_array` keyword causes the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain.

The status variables for the various master key registers returned in the `rule_array` will indicate which of these verification pattern structures returned contain useful data. An empty master key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSA operational key parts is given in Table 13.

**Note:**
1. The fields will be returned in the order given, however the `_ID` fields should be used for verification.
2. The `verb_data_length` parameter will indicate the total size at the bottom of the table describing the `verb_data`.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX®C communication.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_OMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_OMK_MDC4_ID</td>
<td>2</td>
<td>X’0F02’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_OMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key old master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X’0F01’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X’0F00’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_ID</td>
<td>2</td>
<td>X’0F05’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.</td>
</tr>
</tbody>
</table>
### Table 13. Output data format for STATICSA operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X'0F04'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X'0F03'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_OMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_OMK_VP_ID</td>
<td>2</td>
<td>X'0F08'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_OMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_CMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_CMK_VP_ID</td>
<td>2</td>
<td>X'0F07'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_CMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_NMK_VP_ID</td>
<td>2</td>
<td>X'0F06'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_MKAP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Authentication pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Authentication pattern).</td>
</tr>
<tr>
<td>SYM_NMK_MKAP_ID</td>
<td>2</td>
<td>X'0F09'</td>
<td>Hexadecimal identifier indicating the contents of the following Authentication pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_MKAP</td>
<td>8</td>
<td>variable</td>
<td>Authentication pattern over the Symmetric Key new master-key register, calculated using the ICSF specified algorithm.</td>
</tr>
<tr>
<td>AES_OMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>AES_OMK_VP_ID</td>
<td>2</td>
<td>X'0F0C'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
</tbody>
</table>
**STATICSB**

This *rule_array* keyword causes the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain.

The status variables for the various master-key registers returned in the *rule_array* will indicate which of these verification pattern structures returned contain useful data. An empty master-key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSB operational key parts is given in Table 14.

**Note:**
1. The fields will be returned in the order given, however the *_ID* fields should be used for verification.
2. The *verb_data_length* parameter will indicate the total size at the bottom of the table describing the *verb_data*.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

---

**Table 14. Output data format for STATICSB operational key parts**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_OMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the AES Key old master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>AES_CMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>AES_CMK_VP_ID</td>
<td>2</td>
<td>X'0F0B'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>AES_CMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the AES Key current master-key register calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>AES_NMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>AES_NMK_VP_ID</td>
<td>2</td>
<td>X'0F0A'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>AES_NMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the AES Key new master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>Total byte count</td>
<td>204</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 14. Output data format for STATICSB operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X'0F01'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X'0F00'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_ID</td>
<td>2</td>
<td>X'0F05'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X'0F04'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X'0F03'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_OMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_OMK_VP_ID</td>
<td>2</td>
<td>X'0F08'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_OMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_CMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_CMK_VP_ID</td>
<td>2</td>
<td>X'0F07'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
</tbody>
</table>
### Cryptographic Facility Query (CSUACFQ)

Table 14. Output data format for STATICSB operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_CMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_NMK_VP_ID</td>
<td>2</td>
<td>X'0F06'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_MKAP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Authentication pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Authentication pattern).</td>
</tr>
<tr>
<td>SYM_NMK_MKAP_ID</td>
<td>2</td>
<td>X'0F09'</td>
<td>Hexadecimal identifier indicating the contents of the following Authentication pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_MKAP</td>
<td>8</td>
<td>variable</td>
<td>Authentication pattern over the Symmetric Key new master-key register, calculated using the ICSF specified algorithm.</td>
</tr>
<tr>
<td>AES_OMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>AES_OMK_VP_ID</td>
<td>2</td>
<td>X'0F0C'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>AES_OMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key old master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>AES_CMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>AES_CMK_VP_ID</td>
<td>2</td>
<td>X'0F0B'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>AES_CMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key current master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>AES_NMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Authentication pattern).</td>
</tr>
<tr>
<td>AES_NMK_VP_ID</td>
<td>2</td>
<td>X'0F0A'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>AES_NMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key new master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>APKA_OMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>APKA_OMK_VP_ID</td>
<td>2</td>
<td>X'0F0F'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>APKA_OMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key old master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>APKA_CMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
</tbody>
</table>
Table 14. Output data format for STATICSB operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APKA_CMK_VP_ID</td>
<td>2</td>
<td>X’0F0E’</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>APKA_CMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key current master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
<tr>
<td>APKA_NM_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Authentication pattern).</td>
</tr>
<tr>
<td>APKA_NM_VP_ID</td>
<td>2</td>
<td>X’0F0D’</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>APKA_NM_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key new master-key register, calculated using the SHA-256 algorithm.</td>
</tr>
</tbody>
</table>

Total byte count 240

This keyword was introduced with CCA 4.1.0.

**STATICSE**

This rule_array keyword causes the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain.

The status variables for the various master-key registers returned in the rule_array will indicate which of these verification pattern structures returned contain useful data. An empty master-key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSE operational key parts is given in Table 15.

**Note:**

1. The fields will be returned in the order given, however the *_ID fields should be used for verification.
2. The verb_data_length parameter will indicate the total size at the bottom of the table describing the verb_data.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

Table 15. Output data format for STATICSE operational key parts

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_OMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_OMK_MDC4_ID</td>
<td>2</td>
<td>X’0F02’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_OMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key old master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
</tbody>
</table>
**Cryptographic Facility Query (CSUACFQ)**

Table 15. Output data format for STATICSE operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X’0F01’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X’0F00’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_ID</td>
<td>2</td>
<td>X’0F05’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X’0F04’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X’0F03’</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_OMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_OMK_VP_ID</td>
<td>2</td>
<td>X’0F08’</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_OMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_CMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_CMK_VP_ID</td>
<td>2</td>
<td>X’0F07’</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_CMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.</td>
</tr>
</tbody>
</table>
Table 15. Output data format for STATICSE operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_NMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_NMK_VP_ID</td>
<td>2</td>
<td>X'0F06'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_MKAP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Authentication pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Authentication pattern).</td>
</tr>
<tr>
<td>SYM_NMK_MKAP_ID</td>
<td>2</td>
<td>X'0F09'</td>
<td>Hexadecimal identifier indicating the contents of the following Authentication pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_MKAP</td>
<td>8</td>
<td>variable</td>
<td>Authentication pattern over the Symmetric Key new master-key register, calculated using the ICSF specified algorithm.</td>
</tr>
<tr>
<td>Total byte count</td>
<td></td>
<td></td>
<td>168</td>
</tr>
</tbody>
</table>

STATICSX

This *rule_array* keyword causes the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain.

The status variables for the various master key registers returned in the *rule_array* will indicate which of these verification pattern structures returned contain useful data. An empty master key register cannot have a meaningful verification pattern. However, the data structures are returned for all registers indicated, so that interpretation is reliable.

The output data format for STATICSX operational key parts is given in Table 16.

Note:
1. The fields will be returned in the order given, however the *_ID fields should be used for verification.
2. The *verb_data_length* parameter will indicate the total size at the bottom of the table describing the *verb_data*.
3. Multiple byte fields are stored in Big-Endian format, as is typical for CEX/C communication.

Table 16. Output data format for STATICSX operational key parts

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_OMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <em>verb_data</em> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_OMK_MDC4_ID</td>
<td>2</td>
<td>X'0F02'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_OMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key old master-key register, calculated using the MDC4 algorithm.</td>
</tr>
</tbody>
</table>
### Table 16. Output data format for STATICSX operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X'0F01'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X'0F00'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Symmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_ID</td>
<td>2</td>
<td>X'0F05'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_OMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key old master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_ID</td>
<td>2</td>
<td>X'0F04'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_CMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key current master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_LEN</td>
<td>2</td>
<td>20</td>
<td>Length in bytes of this Hash pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Hash pattern).</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_ID</td>
<td>2</td>
<td>X'0F03'</td>
<td>Hexadecimal identifier indicating the contents of the following Hash pattern field.</td>
</tr>
<tr>
<td>ASYM_NMK_MDC4_HP</td>
<td>16</td>
<td>variable</td>
<td>Hash pattern over the Asymmetric Key new master-key register, calculated using the MDC4 algorithm.</td>
</tr>
<tr>
<td>SYM_OMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_OMK_VP_ID</td>
<td>2</td>
<td>X'0F08'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_OMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key old master-key register calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_CMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_CMK_VP_ID</td>
<td>2</td>
<td>X'0F07'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
</tbody>
</table>
Table 16. Output data format for STATICSX operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM_CMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key current master-key register, calculated using the default algorithm.</td>
</tr>
<tr>
<td>SYM_NMK_VP_LEN</td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Verification pattern block in the verb_data (comprising this length field, the following ID field, and the field for the Verification pattern).</td>
</tr>
<tr>
<td>SYM_NMK_VP_ID</td>
<td>2</td>
<td>X'0F06'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td>SYM_NMK_VP</td>
<td>8</td>
<td>variable</td>
<td>Verification pattern over the Symmetric Key new master-key register, calculated using the default algorithm.</td>
</tr>
<tr>
<td>Total byte count</td>
<td></td>
<td></td>
<td>156</td>
</tr>
</tbody>
</table>

STATKPR

This keyword cause non-secret information about a particular named operational key part loaded by the TKE to returned to the user.

The structures for various key types are given under "OUTPUT DATA." An appropriate name for an existing operational key part is expected to be provided as "INPUT DATA." If not, the error return code of 8 and a reason code of 1026 will be returned, meaning 'key name not found'.

INPUT DATA:

A 64-byte key name must be provided in the verb_data field, while the verb_data_length must be set to a value of 64.

The operational key name must match exactly the name returned by a call to STATKPRL.

OUTPUT DATA:

The output data format for STATKPR operational key parts is provided in this topic.

See Table 17 on page 104 for the output data format.

Note:

1. The fields will be returned in the order given.
2. Output data will overwrite the input data in the verb_data field, and set the verb_data_length field to the output value.
3. The verb_data_length parameter will indicate the total size, at the bottom of the table describing the verb_data.
   Notice that the output data is smaller than the input data.
4. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.
Table 17. Output data format for STATKPR operational key parts

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>1</td>
<td>State of the key part register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'03'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'11'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'12'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'13'</td>
</tr>
<tr>
<td>reserved</td>
<td>1</td>
<td>Will have a value of X'00'.</td>
</tr>
<tr>
<td>key_length</td>
<td>1</td>
<td>Length of key in bytes. For DES keys, values are: 8, 16, 24. For AES keys, values are: 16, 24, 32.</td>
</tr>
<tr>
<td>cv_length</td>
<td>1</td>
<td>Length of Control Vector (CV) for key part, in bytes. The value will be 8 or 16 bytes, indicating how much of the CV field to use. Note that CV is NOT a variable length field.</td>
</tr>
<tr>
<td>cv</td>
<td>16</td>
<td>Control Vector for the operational key part.</td>
</tr>
<tr>
<td>reserved_2</td>
<td>8</td>
<td>Will have a value of X'00' for the entire length.</td>
</tr>
<tr>
<td>key_part_hash</td>
<td>20</td>
<td>Hash over the key stored in the key part register. For DES keys, the hash algorithm is SHA-1. For AES keys, the hash algorithm is SHA-256.</td>
</tr>
<tr>
<td>ver_pattern</td>
<td>4</td>
<td>Verification pattern over the key calculated using the default algorithm.</td>
</tr>
<tr>
<td>Total byte count</td>
<td>52</td>
<td></td>
</tr>
</tbody>
</table>

**STATKPRL**

This keyword causes a list of the names of all the operational key parts loaded by the TKE into the CEX*C to be returned.

Each name has a length of 64 bytes. The maximum number of key slots (and thus the count of labels returned) depends on the firmware loaded to the adapter; as of level CCA 4.2 this count was 100 key slots. If not enough space has been provided (using the verb_data_length field passed in by the application) to return the available list, an error is returned.

**STATVKPL**

This keyword causes a list of the names of all the operational key parts loaded by the TKE for variable length key token preparation into the CEX*C to be returned.

This function is different from STATKPRL, which describes the list of 64-byte legacy style tokens in preparation.

Each name has a length of 64 bytes. The maximum number of key slots (and thus the count of labels returned) depends on the firmware loaded to the adapter; as of level CCA 4.2 this count was 100 key slots. If not enough space has been provided (using the verb_data_length field passed in by the application) to return the available list, an error is returned.

**STATVKPR**

This keyword cause non-secret information about a particular named operational key part loaded by the TKE to returned to the user.
Cryptographic Facility Query (CSUACFQ)

This is different from STATKPR in that a register for creating a key in a variable length key token is described. The structures for various key types are given under "OUTPUT DATA." An appropriate name for an existing operational key part is expected to be provided as "INPUT DATA." If not, the error return code of 8 and a reason code of 1026 will be returned, meaning 'key name not found'.

**INPUT DATA:**

A 64 byte key name must be provided in the *verb_data* field, while the *verb_data_length* must be set to 64.

The operational key name must match exactly the name returned by a call to STATVKP.

**OUTPUT DATA:**

The output data format for the operational key parts is provided in this topic.

See [Table 18] for the output data format.

**Note:**

1. The fields will be returned in the order given.
2. Output data will overwrite the input data in the *verb_data* field, and set the *verb_data_length* field to the output value.
3. The *verb_data_length* parameter will indicate the total size, at the bottom of the table describing the *verb_data*.
   Notice that the output data is smaller than the input data.
4. Multiple byte fields are stored in Big-Endian format, as is typical for CEX*C communication.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>1</td>
<td>Version of the structure</td>
</tr>
<tr>
<td>state</td>
<td>1</td>
<td>State of the key part register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'03'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'11'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'12'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'13'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'21'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'22'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'23'</td>
</tr>
<tr>
<td>key_length</td>
<td>1</td>
<td>Length of key in bytes. For DES keys, values are: 8, 16, 24. For AES keys, values are: 16, 24, 32.</td>
</tr>
</tbody>
</table>
Cryptographic Facility Query (CSUACFQ)

Table 18. Output data format for STATVKPR operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_completeness</td>
<td>1</td>
<td>Number of parts needed to complete key</td>
</tr>
<tr>
<td>X’C0’</td>
<td></td>
<td>Two parts are needed.</td>
</tr>
<tr>
<td>X’80’</td>
<td></td>
<td>One part is needed.</td>
</tr>
<tr>
<td>X’40’</td>
<td></td>
<td>No parts are needed. Key is complete.</td>
</tr>
<tr>
<td>ver_pattern</td>
<td>4</td>
<td>ENC-ZERO method calculated verification pattern of the key.</td>
</tr>
<tr>
<td>key_part_hash</td>
<td>8</td>
<td>Hash using the SHA-256 algorithm over the key that is currently stored, at</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the current level of completeness.</td>
</tr>
<tr>
<td>skel_length</td>
<td>2</td>
<td>Skeleton token length.</td>
</tr>
<tr>
<td>pad</td>
<td>2</td>
<td>Pad structure to 4-byte boundary.</td>
</tr>
<tr>
<td>skel</td>
<td>384</td>
<td>Stored key token skeleton, which will hold completed key when operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is complete. No key material is stored or returned here.</td>
</tr>
<tr>
<td>reserved2</td>
<td>108</td>
<td>Extra bytes.</td>
</tr>
<tr>
<td>Total byte count</td>
<td>512</td>
<td></td>
</tr>
</tbody>
</table>

Restrictions
You cannot limit the number of returned rule_array elements.

Tip: Allocate a minimum of 30 rule_array elements to allow for extensions of the returned information.

Required commands
The CSUACFQ required commands.
None

Usage notes
Usage notes for CSUACFQ.

This verb is not impacted by the AUTOSELECT option. See Table 1 on page 10 for more information.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSUACFQJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSUACFQJ are shown here.

Format

```java
public native void CSUACFQJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
```


Cryptographic Facility Query (CSUACFQ)

```c
byte[] rule_array,
hikmNativeInteger verb_data_length,
byte[] verb_data
);
```
Cryptographic Facility Version (CSUACFV)

The Cryptographic Facility Version verb is used to retrieve information about the Security Application Program Interface (SAPI) Version and the Security Application Program Interface build date.

In the same format as the Cryptographic Facility Query (CSUACFQ) verb returns for the CCA application with the STATCCA rule_array option.

This verb returns information elements in the version_data variable.

Format

The format of CSUACFV.

```plaintext
CSUACFV(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    version_data_length,
    version_data)
```

Parameters

The parameter definitions for CSUACFV.

Note that there is no rule_array keyword.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

version_data_length

Direction: Input/Output
Type: Integer

The version_data_length parameter is a pointer to an integer variable containing the number of bytes in the version_data variable. This value must be a minimum of 17 bytes. On input, the version_data_length variable must be set to the total size of the variable pointed to by the version_data parameter. On output, this variable contains the number of bytes of data returned by the verb in the version_data variable.

version_data

Direction: Output
Type: String

The version_data parameter is a pointer to a string variable containing data returned by the verb. An 8-byte character string identifies the version of the Security Application Program Interface (SAPI) library, followed by an 8-byte character string containing the build date for the SAPI library, followed by a null terminating character. The build date is in the format: yyyyymmdd, where yyyy is the year, mm is the month, and dd is the day of the month.

Restrictions

The restrictions for CSUACFV

None
Cryptographic Facility Version (CSUACFV)

**Required commands**
The CSUACFV required commands.

None

**Usage notes**
Usage notes for CSUACFV.

None

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSUACFVJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSUACFVJ are shown here.

**Format**
```java
public native void CSUACFVJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger version_data_length,
    byte[] version_data
);
```
Cryptographic Resource Allocate (CSUACRA)

The Cryptographic Resource Allocate verb is used to allocate a specific CCA coprocessor for use by the thread or process, depending on the scope of the verb.

This verb is scoped to a thread. When a thread or process, depending on the scope, allocates a cryptographic resource, requests are routed to that resource. When a cryptographic resource is not allocated, requests are routed to the default cryptographic resource.

You can set the default cryptographic resource. If you take no action, the default assignment is CRP01.

You cannot allocate a cryptographic resource while one is already allocated. Use the Cryptographic Resource Deallocate verb (see "Cryptographic Resource Deallocate (CSUACRD)" on page 113) to deallocate an allocated cryptographic resource.

Be sure to review "Multi-coprocessor capabilities" on page 9.

Format
The format of CSUACRA.

```c
CSUACRA(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    resource_name_length,
    resource_name)
```

Parameters
The parameter definitions for CSUACRA.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

```c
rule_array_count
    Direction: Input
    Type: Integer
```

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1.

```c
rule_array
    Direction: Input
    Type: Array
```

The rule_array parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keyword is described in Table 19 on page 111.
Table 19. Keywords for Cryptographic Resource Allocate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cryptographic resource</strong> <em>(Required)</em></td>
<td></td>
</tr>
<tr>
<td>DEVICE</td>
<td>Specifies a CEX2C, CEX3C or CEX4C coprocessor.</td>
</tr>
<tr>
<td>DEV-ANY</td>
<td>Specifies to enable the AUTOSELECT option, such that the operating system may select the CCA coprocessor to be used from the available resources according to its policy. This selection applies to most verbs, but not all. See &quot;Multi-coprocessor capabilities&quot; on page 9 for more information.</td>
</tr>
<tr>
<td>HCPUACLR</td>
<td>Specifies the use of host CPU assist for clear keys. This keyword enables clear key use of the CPACF, for clear key AES encryption and decryption with hash algorithms: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. This is the default state at the time of the first use of the CCA library by a PID or TID.</td>
</tr>
<tr>
<td>HCPUAPRT</td>
<td>Specifies the use of host CPU assist for protected keys. This keyword enables protected key use of the CPACF for protected key AES and DES, TDES, and MAC. This is not the default state at the time of the first use of the CCA library by a PID or TID.</td>
</tr>
</tbody>
</table>

There are environment variables that also impact default card, CSU_DEFAULT_ADAPTER (see "Multi-coprocessor capabilities" on page 9) and environment variables that influence CPACF support (see "Environment variables that affect CPACF usage" on page 13).

The actual hardware configuration determines what features are available, and CCA will use what exists if the user sets these values as desired, with respect to appropriate defaults.

**resource_name_length**

  **Direction:** Input  
  **Type:** Integer

The `resource_name_length` parameter is a pointer to an integer variable containing the number of bytes of data in the `resource-name` variable. The length must be 1 - 64.

**resource_name**

  **Direction:** Input  
  **Type:** String

The `resource_name` parameter is a pointer to a string variable containing the name of the coprocessor to be allocated.

**Restrictions**

Restrictions for CSUACRA.

None

**Required commands**

The CSUACRA required commands.

None.
Cryptographic Resource Allocate (CSUACRA)

Usage notes

Usage notes for CSUACRA.

For optimal performance, ensure that you have enabled CPACF in the thread doing
the processing, by making a quick call on the host side at thread startup time, to
Cryptographic Resource Allocate, specifying the correct HCPUACL and
HCPUAPRT keyword values for your operation. See the Cryptographic Resource
Allocate rule_array keyword definitions, and see "Access control points that affect
CPACF protected key operations" on page 14 for more affected verbs. Note that it
is the user's responsibility to ensure that all adapters accessible to the operating
system use the "DEV-ANY " state. Further use of "DEV-ANY " as a target adapter for
verbs that modify the state of an adapter (such as the generation of retained keys)
can lead to unexpected results. See "Multi-coprocessor capabilities" on page 9 for
more information.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUACRAJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSUACRAJ are shown here.

Format

public native void CSUACRAJ(
        hikmNativeInteger return_code,
        hikmNativeInteger reason_code,
        hikmNativeInteger exit_data_length,
        byte[] exit_data,
        hikmNativeInteger rule_array_count,
        byte[] rule_array,
        hikmNativeInteger resource_name_length,
        byte[] resource_name);
Cryptographic Resource Deallocate (CSUACRD)

The Cryptographic Resource Deallocate verb is used to deallocate a specific CCA coprocessor that is allocated by the thread or process, depending on the scope of the verb.

This verb is scoped to a thread. When a thread or process, depending on the scope, de-allocates a cryptographic resource, requests are routed to the default cryptographic resource.

You can set the default cryptographic resource. If you take no action, the default assignment is CRP01.

If a thread with an allocated coprocessor ends without first de-allocating the coprocessor, excess memory consumption results. It is not necessary to deallocate a cryptographic resource if the process itself is ending, only if individual threads end while the process continues to run.

Be sure to review “Multi-coprocessor capabilities” on page 9.

Format

The format of CSUACRD.

```plaintext
CSUACRD( return_code, reason_code, exit_data_length, exit_data, rule_array_count, rule_array, resource_name_length, resource_name)
```

Parameters

The parameter definitions for CSUACRD.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1.

`rule_array`

Direction: Input
Type: Array

The `rule_array` parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keyword is described in Table 20 on page 114.
Table 20. Keywords for Cryptographic Resource Deallocate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptographic resource (Required)</td>
<td></td>
</tr>
<tr>
<td>DEVICE</td>
<td>Specifies a CEX2C, CEX3C or CEX4C coprocessor.</td>
</tr>
<tr>
<td>DEV-ANY</td>
<td>Specifies to disable the AUTOSELECT option. Verbs will now all use the default adapter or a previously configured &quot;selected&quot; adapter as chosen via CSUACRA. See &quot;Multi-coprocessor capabilities&quot; on page 9 for more information.</td>
</tr>
<tr>
<td>HCPUACLR</td>
<td>Specifies the use of host CPU assist for clear keys. This keyword enables clear key use of the CPACF, for clear key AES encryption and decryption with hash algorithms: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. This is the default state at the time of the first use of the CCA library by a PID or TID.</td>
</tr>
<tr>
<td>HCPUAPRT</td>
<td>Specifies the use of host CPU assist for protected keys. This keyword disables protected key use of the CPACF for protected key AES and DES, TDES, and MAC. This is the default state at the time of the first use of the CCA library by a PID or TID.</td>
</tr>
</tbody>
</table>

There are environment variables that also impact default card, CSU_DEFAULT_ADAPTER (see "Multi-coprocessor capabilities" on page 9), and environment variables that influence CPACF support (see "Environment variables that affect CPACF usage" on page 13).

The actual hardware configuration determines what features are available, and CCA will use what exists if the user sets these values as desired, with respect to appropriate defaults.

resource_name_length

Direction: Input
Type: Integer

The `resource_name_length` parameter is a pointer to an integer variable containing the number of bytes of data in the `resource_name` variable. The length must be 1 - 64.

resource_name

Direction: Input
Type: String

The `resource_name` parameter is a pointer to a string variable containing the name of the coprocessor to be deallocated.

Restrictions

The restrictions for CSUACRD.

None

Required commands

The CSUACRD required commands.

None
Cryptographic Resource Deallocate (CSUACRD)

Usage notes

Usage notes for CSUACRD.

To disable CPACF usage in your processing thread, make a call to Cryptographic Resource Deallocate, specifying the correct HCPUACLR and HCPUAPRT keyword as appropriate. See the Cryptographic Resource Deallocate rule_array keyword definitions, and see “Access control points that affect CPACF protected key operations” on page 14 for more affected verbs.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSUACRDJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSUACRDJ are shown here.

Format

```java
public native void CSUACRDJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger resource_name_length,
    byte[] resource_name);
```
Key Storage Initialization (CSNBKSI)

The Key Storage Initialization verb initializes a key-storage file using the current symmetric or asymmetric master-key.

The initialized key storage file does not contain any preexisting key records. The key storage data and index files are in the /opt/IBM/CCA/keys directory.

The key storage functions do not work with HMAC keys.

Format

The format of CSNBKSI.

```
CSNBKSI(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_storage_file_name_length,
    key_storage_file_name,
    key_storage_description_length,
    key_storage_description,
    clear_master_key)
```

Parameters

The parameter definitions for CSNBKSI.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**rule_array_count**

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 2.

**rule_array**

Direction: Input
Type: Array

The `rule_array` parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords are described in Table 21.

**Table 21. Keywords for Key Storage Initialization control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master-key source (Required)</td>
<td></td>
</tr>
<tr>
<td>CURRENT</td>
<td>Specifies the current symmetric master-key of the default cryptographic facility is to be used for the initialization.</td>
</tr>
<tr>
<td>Key-storage selection (One required)</td>
<td></td>
</tr>
</tbody>
</table>
**Key Storage Initialization (CSNBKSI)**

Table 21. Keywords for Key Storage Initialization control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Initialize AES key storage.</td>
</tr>
<tr>
<td>DES</td>
<td>Initialize DES key storage.</td>
</tr>
<tr>
<td>PKA</td>
<td>Initialize PKA key storage (PKA and, beginning with Release 4.1.0, ECC key tokens).</td>
</tr>
</tbody>
</table>

**key_storage_file_name_length**

Direction: Input
Type: Integer

The `key_storage_file_name_length` parameter is a pointer to an integer variable containing the number of bytes of data in the `key_storage_file_name` variable. The length must be within the range of 1 - 64.

**key_storage_file_name**

Direction: Input
Type: String

The `key_storage_file_name` parameter is a pointer to a string variable containing the fully qualified file name of the key-storage file to be initialized. If the file does not exist, it is created. If the file does exist, it is overwritten and all existing keys are lost.

**key_storage_description_length**

Direction: Input
Type: Integer

The `key_storage_description_length` parameter is a pointer to an integer variable containing the number of bytes of data in the `key_storage_description` variable.

**key_storage_description**

Direction: Input
Type: String

The `key_storage_description` parameter is a pointer to a string variable containing the description string stored in the key-storage file when it is initialized.

**clear_master_key**

Direction: Input
Type: String

The `clear_master_key` parameter is unused, but it must be declared and point to 24 data bytes in application storage.

**Restrictions**

The restrictions for CSNBKSI.

ECC and variable-length symmetric key tokens are not supported in releases before Release 4.1.0.
Key Storage Initialization (CSNBKSI)

**Required commands**
The CSNBKSI required commands.

The Key Storage Initialization verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

**Usage notes**
Usage notes for CSNBKSI.

Using the Key Storage Initialization verb to initialize DES (symmetric) key storage currently requires both the symmetric master key and the asymmetric master key to be loaded, completed and set using the Master Key Process (CSNBMKP) verb or suitable utility, or an interface tool that uses the Master Key Process (CSNBMKP) verb.

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNBKSIJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKSIJ are shown here.

**Format**
```
public native void CSNBKSIJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger filename_length,
    byte[] filename,
    hikmNativeInteger key_storage_description_length,
    byte[] key_storage_description,
    byte[] clear_master_key);
```
The Master Key Process verb operates on the three master-key registers: new, current, and old.

Use the verb to perform the following services:
- Clear the new and clear the old master-key registers.
- Generate a random master-key value in the new master-key register.
- XOR a clear value as a key part into the new master-key register.
- Set the master key, which transfers the current master-key to the old master-key register, and the new master-key to the current master-key register. It then clears the new master-key register.

You can choose to process either the symmetric or asymmetric registers by specifying the SYM-MK and the ASYM-MK rule_array keywords.

Tip: Before starting to load new master-key information, ensure the new master-key register is cleared. Do this by using the CLEAR keyword in the rule_array.

To form a master key from key parts in the new master-key register, use the verb several times to complete the following tasks:
- Clear the register, if it is not already clear.
- Load the first key part.
- Load any middle key parts, calling the verb once for each middle key part.
- Load the last key part.
- SET or confirm a master key for which the last key part has been loaded into the new master-key register.

For the SYM-MK, the low-order bit in each byte of the key is used as parity for the remaining bits in the byte. Each byte of the key part must contain an odd number of one bits. If this is not the case, a warning is issued. The product maintains odd parity on the accumulated symmetric master-key value.

When the last master key part is entered, this additional processing is performed:
- If any two of the 8-byte parts of the new master-key have the same value, a warning is issued. Do not ignore this warning. Do not use a key with this property.
- If any of the 8-byte parts of the new master-key compares equal to one of the weak DES-keys, the verb fails with return code 8, reason code 703. See "Questionable DES keys" on page 127 for a list of these weak keys. A parity-adjusted version of the asymmetric master-key is used to look for weak keys.

If an AES, DES or PKA key storage exists, the header record of each key storage is updated with the verification pattern of the new, current master-key.
Master Key Process (CSNBMKP)

Format
The format of CSNBMKP.

```c
CSNBMKP(
    return_code,  
    reason_code,  
    exit_data_length,  
    exit_data,  
    rule_array_count,  
    rule_array,  
    key_part)
```

Parameters
The parameter definitions for CSNBMKP.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`
- Direction: Input
- Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1, 2, or 3.

`rule_array`
- Direction: Input
- Type: Array

The `rule_array` parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords are described in Table 22.

Table 22. Keywords for Master Key Process control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cryptographic component</strong> (Optional)</td>
<td></td>
</tr>
<tr>
<td>ADAPTER</td>
<td>Specifies the coprocessor. This is the default.</td>
</tr>
<tr>
<td><strong>Master key register class</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>AES-MK</td>
<td>Specifies operation with the AES master-key registers.</td>
</tr>
<tr>
<td>APKA-MK</td>
<td>Specifies operation with the APKA master-key registers. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>ASYM-MK</td>
<td>Specifies operation with the asymmetric master-key registers.</td>
</tr>
<tr>
<td>SYM-MK</td>
<td>Specifies operation with the symmetric master-key registers.</td>
</tr>
<tr>
<td><strong>Master-key process</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td>Specifies to clear the NMK register.</td>
</tr>
<tr>
<td>FIRST</td>
<td>Specifies to load the first <code>key_part</code>.</td>
</tr>
<tr>
<td>MIDDLE</td>
<td>Specifies to XOR the second, third, or other intermediate <code>key_part</code> into the NMK register.</td>
</tr>
<tr>
<td>LAST</td>
<td>Specifies to XOR the last <code>key_part</code> into the NMK register.</td>
</tr>
</tbody>
</table>
Master Key Process (CSNBMKP)

Table 22. Keywords for Master Key Process control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>Specifies to advance the CMK to the OMK register, to advance the NMK to the CMK register, and to clear the NMK register.</td>
</tr>
</tbody>
</table>

Note: The master-key register class is not optional for Linux on IBM System z. There is no default for this environment. If a suitable keyword is not specified, return code 8 with reason code 33 will be returned.

key_part

Direction: Input
Type: String

A pointer to a string variable containing a 168-bit or 192-bit clear key-part used when you specify one of the keywords FIRST, MIDDLE, or LAST. If you use the CLEAR or SET keywords, the information in the variable is ignored, but you must declare the variable.

Restrictions

The restrictions for CSNBMKP.

General restrictions:

- You must set up the groups for the users who will be loading the master keys to the cards. Each part of the load process is owned by a different Linux group created by the RPM install procedure, and verified in the host library implementing the API allowing master key processing. To complete a specific step, the user must have membership in the proper group. See Master key load (Step 7 on page 762).
- The AES-MK rule-array keyword is not supported in releases before Release 3.30.
- The APKA-MK rule-array keyword is not supported in releases before Release 4.1.0.

For applications that use this verb:
- When writing your own application, you must link it with the /usr/lib64/libcsulccamk.so library.

Required commands

The CSNBMKP required commands.

This verb requires the following commands to be enabled in the active role based on the master-key class and master-key operation:

<table>
<thead>
<tr>
<th>Master-key operation</th>
<th>Master-key class</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR</td>
<td>AES-MK</td>
<td>X'0124'</td>
<td>Clear New AES Master Key</td>
</tr>
<tr>
<td>SYM-MK</td>
<td>X'0032'</td>
<td></td>
<td>Clear New DES Master Key Register</td>
</tr>
<tr>
<td>ASYM-MK</td>
<td>X'0060'</td>
<td></td>
<td>Clear New RSA Master Key Register</td>
</tr>
<tr>
<td>APKA-MK</td>
<td>X'003F'</td>
<td></td>
<td>Clear New ECC Master Key</td>
</tr>
<tr>
<td>FIRST</td>
<td>AES-MK</td>
<td>X'0125'</td>
<td>Load First AES Master Key Part</td>
</tr>
<tr>
<td>SYM-MK</td>
<td>X'0018'</td>
<td></td>
<td>Load First DES Master Key Part</td>
</tr>
<tr>
<td>ASYM-MK</td>
<td>X'0053'</td>
<td></td>
<td>Load First RSA Master Key Part</td>
</tr>
<tr>
<td>APKA-MK</td>
<td>X'0320'</td>
<td></td>
<td>Load First ECC Master Key Part</td>
</tr>
</tbody>
</table>
Master Key Process (CSNBMKP)

<table>
<thead>
<tr>
<th>Master-key operation</th>
<th>Master-key class</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIDDLE or LAST</td>
<td>AES-MK</td>
<td>X'0126'</td>
<td>Combine AES Master Key Parts</td>
</tr>
<tr>
<td></td>
<td>SYM-MK</td>
<td>X'0019'</td>
<td>Combine DES Master Key Parts</td>
</tr>
<tr>
<td></td>
<td>ASYM-MK</td>
<td>X'0054'</td>
<td>Combine RSA Master Key Parts</td>
</tr>
<tr>
<td></td>
<td>APKA-MK</td>
<td>X'0321'</td>
<td>Combine ECC Master Key Parts</td>
</tr>
<tr>
<td>SET</td>
<td>AES-MK</td>
<td>X'0128'</td>
<td>Set AES Master Key</td>
</tr>
<tr>
<td></td>
<td>SYM-MK</td>
<td>X'001A'</td>
<td>Set DES Master Key</td>
</tr>
<tr>
<td></td>
<td>ASYM-MK</td>
<td>X'0057'</td>
<td>Set RSA Master Key</td>
</tr>
<tr>
<td></td>
<td>APKA-MK</td>
<td>X'0322'</td>
<td>Set ECC Master Key</td>
</tr>
</tbody>
</table>

Usage notes

Usage notes for CSNBMKP.

This verb is not impacted by the AUTOSELECT option. See Table 1 on page 10 for more information.

Questionable DES keys

These keys are considered questionable DES keys, and so should probably not be used when entering SYM-MK or ASYM-MK master keys.

01 01 01 01 01 01 01 01 /* weak */
FE FE FE FE FE FE FE FE /* weak */
1F 1F 1F 0E 0E 0E 0E 0E /* weak */
E0 E0 E0 E0 F1 F1 F1 F1 /* weak */
01 FE 01 FE 01 FE 01 FE 01 FE /* semi-weak */
FE 01 FE 01 FE 01 FE 01 FE 01 FE /* semi-weak */
1F 01 E0 01 F1 0E 01 F1 /* semi-weak */
E0 1F 01 F1 0E F1 0E 0E 0E /* semi-weak */
01 E0 01 E0 01 F1 0E 01 F1 /* semi-weak */
FE 01 1F 01 0E 0E 0E 0E 0E /* semi-weak */
1F 01 FE 1F FE 0E FE 0E FE /* semi-weak */
FE 01 FE 1F FE 0E FE 0E FE /* semi-weak */
01 1F 01 1F 01 0E 01 0E 01 /* semi-weak */
1F 01 1F 01 0E 01 0E 01 0E /* semi-weak */
E0 FE E0 FE F1 FE F1 FE /* semi-weak */
FE E0 FE E0 FE F1 FE F1 FE /* semi-weak */
1F 1F 01 0E 0E 01 01 01 /* possibly semi-weak */
01 1F 1F 01 0E 0E 01 01 01 /* possibly semi-weak */
1F 01 1F 0E 01 01 01 0E 01 0E /* possibly semi-weak */
01 01 1F 01 0E 01 01 0E 01 0E /* possibly semi-weak */
E0 E0 01 1F F1 F1 01 01 01 /* possibly semi-weak */
FE FE 01 01 FE FE 01 01 01 /* possibly semi-weak */
FE E0 1F 01 FE F1 FE 0E 01 01 01 /* possibly semi-weak */
FE E0 1F 01 FE F1 FE 0E 01 01 01 /* possibly semi-weak */
E0 E0 1F 1F 0E FE F1 FE 0E 01 /* possibly semi-weak */
FE FE 1F 1F FE F1 FE 0E 01 /* possibly semi-weak */
FE 1F 1F FE 0E FE F1 01 01 01 01 01 /* possibly semi-weak */
E0 1F FE 01 F1 FE 0E 01 01 01 01 /* possibly semi-weak */
FE 01 E0 1F FE 0E F1 FE 0E 01 01 01 /* possibly semi-weak */
E0 1F FE 01 F1 FE 0E 01 01 01 01 01 /* possibly semi-weak */
01 E0 01 01 F1 F1 01 01 01 01 /* possibly semi-weak */
1F FE E0 01 0E FE F1 01 01 01 01 /* possibly semi-weak */
1F E0 FE 01 0E F1 FE 01 01 01 01 01 /* possibly semi-weak */
01 FE FE 01 0E F1 FE 0E 01 01 01 01 /* possibly semi-weak */
01 E0 FE 01 F1 FE 0E 01 01 01 01 01 /* possibly semi-weak */
01 E0 FE 1F 0E FE F1 FE 0E 01 01 01 01 /* possibly semi-weak */
1F FE FE 1F 0E FE F1 FE 0E 01 01 01 01 01 /* possibly semi-weak */
JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMKPJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBMKPJ are shown here.

Format

```java
public native void CSNBMKPJ(
        hikmNativeInteger return_code,
        hikmNativeInteger reason_code,
        hikmNativeInteger exit_data_length,
        byte[] exit_data,
        hikmNativeInteger rule_array_count,
        byte[] rule_array,
        byte[] key_part);
```
Random Number Tests (CSUARNT)

Random Number Tests (CSUARNT)

The Random Number Tests verb invokes the USA NIST FIPS PUB 140-1 specified cryptographic operational tests.

These tests, selected by a `rule_array` keyword, consist of:
- For random numbers: a monobit test, poker test, runs test, and long-run test
- Known-answer tests of DES, RSA, and SHA-1 processes

The tests are performed three times. If there is any test failure, the verb returns return code 4 and reason code 1.

Format

The format of CSUARNT.

```c
CSUARNT(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array)
```

Parameters

The parameter definitions for CSUARNT.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

- **Direction**: Input
- **Type**: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1.

`rule_array`

- **Direction**: Input
- **Type**: Array

The `rule_array` parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords are described in Table 23.

Table 23. Keywords for Random Number Tests control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test selection (One required)</td>
<td>Perform the FIPS 140-1 specified test on the random number generation output.</td>
</tr>
<tr>
<td>FIPS-RNT</td>
<td>Perform the FIPS 140-1 specified known-answer tests on DES, RSA, and SHA-1.</td>
</tr>
</tbody>
</table>
Restrictions
The restrictions for CSUARNT.
None

Required commands
The CSUARNT required commands.
None

Usage notes
Usage notes for CSUARNT.
This verb is not impacted by the AUTOSELECT option. See Table 1 on page 10 for more information.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSUARNTJ.
See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSUARNTJ are shown here.

Format
public native void CSUARNTJ(
   hikmNativeInteger return_code,
   hikmNativeInteger reason_code,
   hikmNativeInteger exit_data_length,
   byte[] exit_data,
   hikmNativeInteger rule_array_count,
   byte[] rule_array);
Random Number Tests (CSUARNT)
Chapter 6. Managing AES and DES cryptographic keys

This topic describes the verbs that generate and maintain AES and DES cryptographic keys.

Using CCA, you can generate keys using the Key Generate verb. CCA provides a number of verbs to assist you in managing and distributing AES and DES keys, generating random numbers, and maintaining the key storage files.

This chapter describes the following verbs:

- “Clear Key Import (CSNBCKI)” on page 129
- “Control Vector Generate (CSNBCVG)” on page 134
- “Control Vector Translate (CSNBCVT)” on page 137
- “Cryptographic Variable Encipher (CSNBCVE)” on page 141
- “Data Key Export (CSNBDKX)” on page 144
- “Data Key Import (CSNBDKM)” on page 146
- “Diversified Key Generate (CSNBDKG)” on page 148
- “Key Export (CSNBKEX)” on page 164
- “Key Generate (CSNBKGN)” on page 167
- “Key Generate2 (CSNBKGN2)” on page 176
- “Key Import (CSNBKIM)” on page 184
- “Key Part Import (CSNBKPI)” on page 187
- “Key Part Import2 (CSNBKPI2)” on page 191
- “Key Test (CSNBKYT)” on page 195
- “Key Test2 (CSNBKYT2)” on page 200
- “Key Test Extended (CSNBKYTX)” on page 204
- “Key Token Build (CSNBKTB)” on page 209
- “Key Token Build2 (CSNBKTB2)” on page 214
- “Key Token Change (CSNBKTC)” on page 218
- “Key Token Change2 (CSNBKTC2)” on page 221
- “Key Token Parse (CSNBKTP)” on page 224
- “Key Translate (CSNBKTR)” on page 239
- “Key Translate2 (CSNBKTR2)” on page 241
- “Multiple Clear Key Import (CSNBCKM)” on page 131
- “PKA Decrypt (CSNDPKD)” on page 245
- “PKA Encrypt (CSNDPKE)” on page 248
- “Prohibit Export (CSNBPEX)” on page 252
- “Prohibit Export Extended (CSNBPEXX)” on page 254
- “Random Number Generate (CSNBRNG)” on page 260
- “Random Number Generate Long (CSNBRNGL)” on page 262
- “Restrict Key Attribute (CSNBRKA)” on page 256
- “Symmetric Key Export (CSNDSYX)” on page 265
- “Symmetric Key Generate (CSNDSYG)” on page 271
- “Symmetric Key Import (CSNDSY1)” on page 276
• “Symmetric Key Import2 (CSNDSYI2)” on page 280
Clear Key Import (CSNBCKI)

Use the Clear Key Import verb to import a clear DATA key that is to be used to encipher or decipher data.

This verb can import only DATA keys. The Clear Key Import verb accepts an 8-byte clear DATA key, enciphers it under the master key, and returns the encrypted DATA key in operational form in an internal key token.

If the clear key value does not have odd parity in the low-order bit of each byte, the verb returns a warning value in the reason_code parameter. This verb does not adjust the parity of the key.

Note: To import 16-byte or 24-byte DATA keys, use the Multiple Clear Key Import verb that is described in "Multiple Clear Key Import (CSNBCKM)" on page 131.

Format

The format of CSNBCKI.

```
CSNBCKI(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  clear_key,
  key_identifier)
```

Parameters

The parameter definitions for CSNBCKI.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

**clear_key**

Direction: Input
Type: String

The clear_key specifies the 8-byte clear key value to import.

**key_identifier**

Direction: Input/Output
Type: String

A 64-byte string that is to receive the internal key token. "Key tokens, key labels, and key identifiers" on page 21 describes the internal key token.

Restrictions

The restrictions for CSNBCKI.

None

Required commands

The CSNBCKI required commands.
Clear Key Import (CSNBCKI)

This verb requires the Clear Key Import/Multiple Clear Key Import - DES command (offset X'00C3') to be enabled in the active role.

Note: A role with offset X'00C3' enabled can also use the Multiple Clear Key Import verb with the DES algorithm.

Usage notes
Usage notes for CSNBCKI.

None

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBCKIJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBCKIJ are shown here.

Format
public native void CSNBCKIJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] clear_key,
    byte[] target_key_identifier );
Multiple Clear Key Import (CSNBCKM)

Use the Multiple Clear Key Import verb to import a clear single, double, or triple-length DATA key that is to be used to encipher or decipher data.

This verb can import only DATA keys. Multiple Clear Key Import accepts a clear DATA key, enciphers it under the master key, and returns the encrypted DATA key in operational form in an internal key token.

Format
The format of CSNBCKM.

```c
CSNBCKM( return_code, reason_code, exit_data_length, exit_data, rule_array_count, rule_array, clear_key_length, clear_key, target_key_identifier
```

Parameters
The parameter definitions for CSNBCKM.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**
- **Direction:** Input
- **Type:** Integer
- A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0, 1, 2, or 3.

**rule_array**
- **Direction:** Input
- **Type:** String
- Zero or one keyword that supplies control information to the verb. The keyword must be in eight bytes of contiguous storage, left-aligned and padded on the right with blanks. The `rule_array` keywords are described in Table 24.

**Table 24. Keywords for Multiple Clear Key Import control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>(On, optional)</td>
</tr>
<tr>
<td>AES</td>
<td>The key should be enciphered under the master key as an AES key.</td>
</tr>
<tr>
<td>DES</td>
<td>The key should be enciphered under the master key as a DES key. This is the default.</td>
</tr>
<tr>
<td>Key-wrapping method</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>USECONFIG</td>
<td>Specifies to wrap the key using the configuration setting for the default wrapping method. This keyword is ignored for AES keys. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
</tbody>
</table>
Multiple Clear Key Import (CSNBCKM)

Table 24. Keywords for Multiple Clear Key Import control information  (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAP-ENH</td>
<td>Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td></td>
<td><em>Translation control</em> (Optional). This is valid only with key-wrapping method WRAP-ENH or with USECONFIG when the default wrapping method is WRAP-ENH. This option cannot be used on a key with a control vector valued to binary zeros. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>ENH-ONLY</td>
<td>Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B’1’.</td>
</tr>
</tbody>
</table>

clear_key_length

Direction: Input

Type: Integer

The clear_key_length specifies the length of the clear key value to import. This length must be 8, 16, or 24.

clear_key

Direction: Input

Type: String

The clear_key specifies the clear key value to import.

target_key_identifier

Direction: Output

Type: String

A 64-byte string that is to receive the internal key token. Appendix B, “Key token formats,” on page 609 describes the key tokens.

Restrictions

The restrictions for CSNBCKM.

None

Required commands

The CSNBCKM required commands.

This verb requires the following commands to be enabled in the active role based on the algorithm or key-wrapping method:

<table>
<thead>
<tr>
<th>Algorithm or method</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>X’0129’</td>
<td>Multiple Clear Key Import/Multiple Secure Key Import - AES</td>
</tr>
<tr>
<td>DES</td>
<td>X’00C3’</td>
<td>Clear Key Import/Multiple Clear Key Import - DES</td>
</tr>
</tbody>
</table>
Multiple Clear Key Import (CSNBCKM)

<table>
<thead>
<tr>
<th>Algorithm or method</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAP-ECB or WRAP-ENH used, and default key-wrapping</td>
<td>X'0141'</td>
<td>Multiple Clear Key Import - Allow wrapping override keywords</td>
</tr>
<tr>
<td>method setting does not match keyword</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Note: A role with offset X'00C3' can also use the Clear Key Import verb.

Usage notes

Usage notes for CSNBCKM.

This verb produces an internal DATA token with a control vector which is usable on the Cryptographic Coprocessor Feature. If a valid internal token is supplied as input to the verb in the target_key_identifier field, that token's control vector will not be used in the encryption of the clear key value.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCKMJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBCKMJ are shown here.

Format

```java
public native void CSNBCKMJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger clear_key_length,
    byte[] clear_key,
    byte[] target_key_identifier);
```
Control Vector Generate (CSNCVG)

The Control Vector Generate verb builds a control vector from keywords specified by the key_type and rule_array parameters.

Format
The format of CSNCVG.

```c
CSNCVG(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_type,
    rule_array_count,
    rule_array,
    reserved,
    control_vector)
```

Parameters
The parameter definitions for CSNCVG.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

key_type
   Direction: Input
   Type: String
   A string variable containing a keyword for the key type. The keyword is eight bytes in length, left-aligned, and padded on the right with space characters. It is taken from the following list:

   CIPHER     DATAC     IKEYXLAT     OPINENC
   CVARDEC    DATAM     IMPORTER     PINENC
   CVARENC    DATMV     IPINENC      PINVER
   CVARPINE   DECIPHER  KEYGENKY     SECMSG
   CVARXCVL   DKYGENKY  MAC
   CVARXCVR   ENCPHIER  MACVER
   DATA       EXPORTER  OKEYXLAT

rule_array_count
   Direction: Input
   Type: Integer
   A pointer to an integer variable containing the number of elements in the rule_array variable.

rule_array
   Direction: Input
   Type: String
   Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields, and padded on the right with blanks. All keywords must be in contiguous storage. "Key Token Build (CSNBKTB)" on page 209 illustrates the key type and key usage keywords that can be combined in the Control Vector Generate and Key Token Build verbs to create a control vector.
Control Vector Generate (CSNBCVG)

See Figure 3 on page 43 for the key usage keywords that can be specified for a given key type. The rule_array keywords are shown here:

AMEX-CSC  DKYL0  EPINGEN  KEYLN16  TRANSLAT
ANSIX9.9  DKYL1  EPINGENA  LMTD-KEK  T31XPTOK
ANY      DKYL2  EPINVER  MIXED  UKPT
ANY-MAC  DKYL3  EXEX  NO-SPEC  VISA-PVV
CLR8-ENC  DKYL4  EXPORT  NO-XPORT  WRAP-ECB
CPINENC  DKYL5  GBP-PIN  NOOFFSET  WRAP-ENH
CPINGEN  DKYL6  GBP-PINO  NOT31XPT  XLATE
CPINGENX  DKYL7  IBM-PIN  NOT-KEK  XPORT-OK
Cvvkey-A  DMAC  IBM-PINO  OPEX
Cvvkey-B  Dmkey  IMEX  OPIM
DALL     DMPIN  IMIM  PIN
DATA     Dmv  IMPORT  REFORMAT
DDATA    DOUBLE  INBK-PIN  SINGLE
DEXP     DPVR  KEY-PART  SMKEY
DIMP     ENH-ONLY  KEYLN8  SMPIN

Note:
1. When the KEYGENKY key type is coded, either CLR8-ENC or UKPT must be specified in rule_array.
2. When the SECMMSG key_type is coded, either SMKEY or SMPIN must be specified in the rule_array.
3. Keywords ENH-ONLY, WRAP-ECB, and WRAP-ENH were introduced with CCA 4.1.0.

reserved
Direction: Input
Type: String
The reserved parameter must be a variable of eight bytes of X'00'.

ccontrol_vector
Direction: Output
Type: String
A 16-byte string variable in application storage where the verb returns the generated control vector.

Restrictions
The restrictions for CSNBCVG.
None

Required commands
The CSNBCVG required commands.
None

Usage notes
Usage notes for CSNBCVG.

See the key_type parameter on page 209 for an illustration of key type and key usage keywords that can be combined in the Control Vector Generate and Key Token Build verbs to create a control vector.
Control Vector Generate (CSNBCVG)

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCVGJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBCVGJ are shown here.

Format

```java
public native void CSNBCVGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_type,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] reserved_field_1,
    byte[] control_vector);
```
Control Vector Translate (CSNBCVT)

The Control Vector Translate verb changes the control vector used to encipher an external DES key.

Detailed information about control vectors and how to use this verb can be found in Appendix D, “Control vectors and changing control vectors with the Control Vector Translate verb,” on page 667.

Format

The format of .CSNBCVT

```c
CSNBCVT(  
  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  KEK_key_identifier,  
  source_key_token,  
  array_key_left_identifier,  
  mask_array_left,  
  array_key_right_identifier,  
  mask_array_right,  
  rule_array_count,  
  rule_array,  
  target_key_token)
```

Parameters

The parameter definitions for CSNBCVT.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**KEK_key_identifier**

Direction: Input
Type: String

A pointer to a string variable containing an operational key-token or the key label of an operational key-token record containing the key-encrypting key. The control vector in the key token must specify the key type IMPORTER, EXPORTER, IKEYXLAT, or OKEYXLAT.

**source_key_token**

Direction: Input
Type: String

A pointer to a string variable containing the external DES key-token with the key and control vector to be processed.

**array_key_left_identifier**

Direction: Input
Type: String

A pointer to a string variable containing an operational DES key-token or a key label of an operational DES key-token record that deciphers the left mask-array. The key token must contain a control vector specifying a CVARXCVL key-type. The CVARXCVL key must be single length.
Control Vector Translate (CSNBCVT)

mask_array_left
  Direction: Input  
  Type: String  
  A pointer to a string variable containing the mask array enciphered under the left-array key.

array_key_right_identifier
  Direction: Input  
  Type: String  
  A pointer to a string variable containing an operational DES key-token or the key label of an operational DES key-token record that deciphers the right mask-array. The key token must contain a control vector specifying a CVARXCVR key-type. The CVARXCVR key must be single length.

mask_array_right
  Direction: Input  
  Type: String  
  A pointer to a string variable containing the mask array enciphered under the right-array key.

rule_array_count
  Direction: Input  
  Type: Integer  
  A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0, 1, or 2.

rule_array
  Direction: Input  
  Type: Array  
  A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are described in Table 25.

Table 25. Keywords for Control Vector Translate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity adjustment</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>ADJUST</td>
<td>Ensures that all target-key bytes have odd parity. This is the default.</td>
</tr>
<tr>
<td>NOADJUST</td>
<td>Prevents the parity of the target key from being altered.</td>
</tr>
<tr>
<td>Key half processing mode</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>LEFT</td>
<td>Causes an 8-byte source key, or the left half of a 16-byte source key, to be processed with the result placed into both halves of the target key. This is the default.</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Causes the right half of a 16-byte source key to be processed with the result placed into only the right half of the target key. The left half of the target key is unchanged.</td>
</tr>
<tr>
<td>BOTH</td>
<td>Causes both halves of a 16-byte source key to be processed with the result placed into corresponding halves of the target key. When you use the BOTH keyword, the mask array must be able to validate the translation of both halves.</td>
</tr>
<tr>
<td>SINGLE</td>
<td>Causes the left half of the source key to be processed with the result placed into only the left half of the target. The right half of the target key is unchanged.</td>
</tr>
</tbody>
</table>

target_key_token
Control Vector Translate (CSNBCVT)

Direction: Input/Output
   Type: String
A pointer to a string variable containing an external DES key-token with the new control vector. This key token contains the key halves with the new control vector.

Restrictions
The restrictions for CSNBCVT.
None

Required commands
The CSNBCVT required commands.

This verb requires the Control Vector Translate command (offset X'00D6') to be enabled in the active role.

Usage notes
Usage notes for CSNBCVT.
Consider that Control Vector Translate represents the capability to translate, by definition, the limitations on the operations that a key can be used for, into a different set of limitations. The control vector is the heart of security against the misuse of keys that were defined for a specific purpose. The masks that control what the key can be translated into being able to do (the right and left masks) are themselves single-length (8-byte), and are encrypted with DES. Therefore, the protection against translating the key to have more power (or less power) than it did before are protected with single-DES. This reduces the security (somewhat) of a double-length DES key. You cannot decrypt the double-length key with this approach, or gain access to a key that you did not otherwise have the rights to use. But you can make a key which you already have access to, on a system you already have access to, more powerful than it was before if you can break single-DES.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBCVTJ.
See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBCVTJ are shown here.

Format
public native void CSNBCVTJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] kek_key_identifier,
    byte[] source_key_token,
    byte[] array_key_left,
    byte[] mask_array_left,
    byte[] array_key_right,
    byte[] mask_array_right,
Control Vector Translate (CSNBCVT)

```c
hikmNativeInteger rule_array_count,
byte[] rule_array,
byte[] target_key_token);
```
This verb is used to encrypt plaintext using a CVARENC key to produce ciphertext using the Cipher Block Chaining (CBC) method.

The plaintext must be a multiple of eight bytes in length.

Specify the following parameters to encrypt plaintext:
- An operational DES key-token or a key label of an operational DES key-token record that contains the key to be used to encrypt the plaintext with the c-variable_encrypting_key_identifier parameter. The control vector in the key token must specify the CVARENC key-type.
- The length of the plaintext, which is the same as the length of the returned ciphertext, with the text_length parameter. The plaintext must be a multiple of eight bytes in length.
- The plaintext with the plaintext parameter.
- The initialization vector with the initialization_vector parameter.
- A variable for the returned ciphertext with the ciphertext parameter. The length of this field is specified with the text_length variable.

This verb does the following:
- Uses the CVARENC key and the initialization value with the CBC method to encrypt the plaintext.
- Returns the encrypted plaintext in the variable pointed to by the ciphertext parameter.

Format
The format of CSNBCVE.

```c
CSNBCVE(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    c-variable_encrypting_key_identifier,
    text_length,
    plaintext,
    initialization_vector,
    ciphertext)
```

Parameters
The parameter definitions for CSNBCVE.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

c-variable_encrypting_key_identifier
- Direction: Input
- Type: String

A pointer to a string variable containing an operational DES key-token or a key label of an operational DES key-token record. The key token must contain a control vector that specifies a CVARENC key-type.

text_length
Cryptographic Variable Encipher (CSNBCVE)

**Direction: Input**
Type: Integer

A pointer to an integer variable containing the length of the plaintext variable and the ciphertext variable.

**plaintext**

**Direction: Input**
Type: String

A pointer to a string variable containing the plaintext to be encrypted.

**initialization_vector**

**Direction: Input**
Type: String

A pointer to a string variable containing the 8-byte initialization vector that the verb uses in encrypting the plaintext.

**ciphertext**

**Direction: Output**
Type: String

A pointer to a string variable containing the ciphertext returned by the verb.

**Restrictions**
The restrictions for CSNBCVE.

The text length must be a multiple of eight bytes.

The minimum length of text that the security server can process is eight bytes and the maximum is 256 bytes.

**Required commands**
The required commands for CSNBCVE.

This verb requires the Cryptographic Variable Encipher command (offset X'00DA') to be enabled in the active role.

**Usage notes**
Usage notes for CSNBCVE.

None

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNBCVEJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBCVEJ are shown here.

**Format**

```java
public native void CSNBCVEJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
```
Cryptographic Variable Encipher (CSNBCVE)

byte[] exit_data,
byte[] cvarenc_key_id,
hikmNativeInteger text_length,
byte[] plain_text,
byte[] init_vector,
byte[] cipher_text);
Data Key Export (CSNBDKX)

Use the Data Key Export verb to re-encipher a data-encrypting key (key type of DATA only) from encryption under the master key to encryption under an exporter key-encrypting key.

The re-enciphered key is in a form suitable for export to another system.

The Data Key Export verb generates a key token with the same key length as the input token’s key.

Format
The format of CSNBDKX.

```plaintext
CSNBDKX(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  source_key_identifier,
  exporter_key_identifier,
  target_key_identifier)
```

Parameters
The parameter definitions for CSNBDKX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`source_key_identifier`
- Direction: Input/Output
- Type: String
- A 64-byte string for an internal key token or label that contains a data-encrypting key to be re-enciphered. The data-encrypting key is encrypted under the master key.

`exporter_key_identifier`
- Direction: Input/Output
- Type: String
- A 64-byte string for an internal key token or key label that contains the exporter key-encrypting key. The data-encrypting key above will be encrypted under this exporter key-encrypting key.

`target_key_identifier`
- Direction: Input/Output
- Type: String
- A 64-byte field that is to receive the external key token, which contains the re-enciphered key that has been exported. The re-enciphered key can now be exchanged with another cryptographic system.

Restrictions
The restrictions for CSNBDKX.
Data Key Export (CSNBDKX)

For security reasons, requests will fail by default if they use an equal key halves exporter to export a key with unequal key halves. You must have access control point 'Data Key Export - Unrestricted' explicitly enabled if you want to export keys in this manner.

Required commands

The CSNBDKX required commands.

This verb requires the Data Key Export command (offset X'010A') to be enabled in the active role.

By also specifying the Data Key Export - Unrestricted command (offset X'0277'), you can permit a less secure mode of operation that enables an equal key-halves EXPORTER key-encrypting-key to export a key having unequal key-halves (key parity bits are ignored).

Usage notes

Usage notes for CSNBDKX.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDKXJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBDKXJ are shown here.

Format

```java
public native void CSNBDKXJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] source_key_identifier,
    byte[] exporter_key_identifier,
    byte[] target_key_token);
```
Data Key Import (CSNBDKM)

Use the Data Key Import verb to import an encrypted source DES single-length, double-length or triple-length DATA key and create or update a target internal key token with the master key enciphered source key.

Format

The format of CSNBDKM.

```
CSNBDKM(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    source_key_token,
    importer_key_identifier,
    target_key_identifier)
```

Parameters

The parameter definitions for CSNBDKM.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**source_key_token**

- **Direction:** Input
- **Type:** String

64-byte string variable containing the source key to be imported. The source key must be an external token or null token. The external key token must indicate that a control vector is present; however, the control vector is usually valued at zero. A double-length key that should result in a default DATA control vector must be specified in a version X'01' external key token. Otherwise, both single and double-length keys are presented in a version X'00' key token. For the null token, the verb will process this token format as a DATA key encrypted by the importer key and a null (all zero) control vector.

**importer_key_identifier**

- **Direction:** Input/Output
- **Type:** String

A 64-byte string variable containing the (IMPORTER) transport key or key label of the transport key used to decipher the source key.

**target_key_identifier**

- **Direction:** Output
- **Type:** String

A 64-byte string variable containing a null key token or an internal key token. The key token receives the imported key.

Restrictions

The restrictions for CSNBDKM.
Data Key Import (CSNBDKM)

For security reasons, requests will fail by default if they use an equal key halves importer to import a key with unequal key halves. You must have access control point 'Data Key Import - Unrestricted' explicitly enabled if you want to import keys in this manner.

Required commands

The CSNBDKM required commands.

This verb requires the Data Key Import command (offset X'0109') to be enabled in the active role.

By also specifying the Data Key Import - Unrestricted command (offset X'027C'), you can permit a less secure mode of operation that enables an equal key-halves IMPORTER key-encrypting key to import a key having unequal key-halves (key parity bits are ignored).

Usage notes

Usage notes for CSNBDKM.

This verb does not adjust the key parity of the source key.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDKMJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBDKMJ are shown here.

Format

```java
public native void CSNBDKMJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] source_key_token,
    byte[] importer_key_identifier,
    byte[] target_key_identifier);
```
Diversified Key Generate (CSNBDKG)

Diversified Key Generate (CSNBDKG)

Use the Diversified Key Generate verb to generate a key based on the
key-generating key, the processing method, and the parameter supplied.

The control vector of the key-generating key also determines the type of target key
that can be generated.

To use this verb, specify the following:

- The rule_array keyword to select the diversification process.
- The operational key-generating key from which the diversified keys are
generated. The control vector associated with this key restricts the use of this
key to the key generation process. This control vector also restricts the type of
key that can be generated.
- The data and length of data used in the diversification process.
- The generated-key could be an internal token or a skeleton token containing the
desired CV of the generated-key. The generated key CV must be one that is
permitted by the processing method and the key-generating key. The generated
key will be returned in this parameter.
- A key generation method keyword.

This verb generates diversified keys as follows:

- Determines if it can support the process specified in the rule_array.
- Recovers the key-generating key and checks the key-generating key class and
the specified usage of the key-generating key.
- Determines that the control vector in the generated-key token is permissible for
the specified processing method.
- Determines that the control vector in the generated-key token is permissible by
the control vector of the key-generating key.
- Determines the required data length from the processing method and the
generated-key CV. Validates the data_length.
- Generates the key appropriate to the specific processing method. Adjusts parity
of the key to odd. Creates the internal token and returns the generated
diversified key.

Format

The format of CSNBDKG.

```plaintext
CSNBDKG(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    generating_key_identifier,
    data_length,
    data,
    key_identifier,
    generated_key_identifier)
```

Parameters

The parameter definitions for CSNBDKG.
Diversified Key Generate (CSNBDKG)

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

rule_array_count
Direction: Input
Type: Integer
A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1, 2, or 3.

rule_array
Direction: Input
Type: String
The keyword that provides control information to the verb. The processing method is the algorithm used to create the generated key. The keyword is left-aligned and padded on the right with blanks. The rule_array keywords are described in Table 26.

Table 26. Keywords for Diversified Key Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Method for generating or updating diversified keys</strong> (One required)</td>
<td></td>
</tr>
</tbody>
</table>
| CLR8-ENC    | Specifies that eight bytes of clear data shall be multiply encrypted with the generating_key_identifier must be a KEYGENKY key type with bit 19 of the control vector set to B'1'. The control vector in generated_key_identifier must specify a single-length key. The key type can be DATA, MAC, or MACVER.  
  Note: CIPHER class keys are not supported. |
| TDES-DEC    | Data supplied could be 8 or 16 bytes of clear data. If the generated_key_identifier specifies a single length key, then 8-bytes of data is TDES decrypted under the generating_key_identifier. If the generated_key_identifier specifies a double length key, then 16-bytes of data is TDES ECB mode decrypted under the generating_key_identifier. No formatting of data is done before encryption. The generating_key_identifier must be a DKYGENKY key type, with appropriate usage bits for the desired generated key. |
| TDES-ENC    | Data supplied could be 8 or 16 bytes of clear data. If the generated_key_identifier specifies a single length key, then 8 bytes of data is TDES encrypted under the generating_key_identifier. If the generated_key_identifier specifies a double length key, then 16 bytes of data is TDES ECB mode encrypted under the generating_key_identifier. No formatting of data is done before encryption. The generating_key_identifier must be a DKYGENKY key type, with appropriate usage bits for the desired generated key. The generated_key_identifier can be a single or double length key, with a CV that is permitted by the generating_key_identifier. |
| TDES-XOR    | This option combines the function of the existing TDES-ENC and SESS-XOR into one step.  
  The generating key must be a level 0 DKYGENKY and cannot have replicated halves. The session key generated must be double length and the allowed key types are DATA, DATAC, DATAM, DATAMV, MAC, MACVER, SMPIN, and SMKEY. Key type must be allowed by the generating key control vector. |
| TDESEMV2    | This option supports generation of a session key by the EMV 2000 algorithm (This EMV2000 algorithm uses a branch factor of 2). The generating key must be a level 0 DKYGENKY and cannot have replicated halves. The session key generated must be double length and the allowed key types are DATA, DATAC, DATAM, DATAMV, MAC, MACVER, SMPIN, and SMKEY. Key type must be allowed by the generating key control vector. |
| TDESEMV4    | This option supports generation of a session key by the EMV 2000 algorithm (This EMV2000 algorithm uses a branch factor of 4). The generating key must be a level 0 DKYGENKY and cannot have replicated halves. The session key generated must be double length and the allowed key types are DATA, DATAC, DATAM, DATAMV, MAC, MACVER, SMPIN, and SMKEY. Key type must be allowed by the generating key control vector. |
### Diversified Key Generate (CSNBDKG)

#### Table 26. Keywords for Diversified Key Generate control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Method for updating a diversified key</strong> (optional)</td>
<td></td>
</tr>
<tr>
<td>SESS-XOR</td>
<td>Specifies the VISA method for session key generation. Data supplied can be 8 or 16 bytes of data depending on whether the generating_key_identifier is a single or double length key. The 8 or 16 bytes of data is XORed with the clear value of the generating_key_identifier. The generated_key_identifier has the same control vector as the generating_key_identifier. The generating_key_identifier can be DATA, DATAC, DATAM, DATAMV, MAC, MACVER key types.</td>
</tr>
<tr>
<td><strong>Key-wrapping method</strong></td>
<td>(One, optional)</td>
</tr>
<tr>
<td>USECONFIG</td>
<td>Specifies to wrap the key using the configuration setting for the default wrapping method. This keyword is ignored for AES keys. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td><strong>Translation control</strong></td>
<td>(Optional). This is valid only with key-wrapping method WRAP-ENH or with USECONFIG when the default wrapping method is WRAP-ENH. This option cannot be used on a key with a control vector valued to binary zeros.</td>
</tr>
<tr>
<td>ENH-ONLY</td>
<td>Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'. This keyword was introduced with CCA 4.1.0.</td>
</tr>
</tbody>
</table>

**generating_key_identifier**

- **Direction:** Input/Output
- **Type:** String

  The label or internal token of a key generating key. The type of key-generating key depends on the processing method.

**data_length**

- **Direction:** Input
- **Type:** Integer

  The length of the data parameter that follows. Length depends on the processing method and the generated key.

**data**

- **Direction:** Input
- **Type:** String

  Data input to the diversified key or session key generation process. Data depends on the processing method and the generated_key_identifier.

**key_identifier**

- **Direction:** Input/Output
- **Type:** String

  This parameter is currently not used. It must be a 64-byte null token.

**generated_key_identifier**

- **Direction:** Input/Output
- **Type:** String
Diversified Key Generate (CSNBDKG)

The internal token of an operational key, a skeleton token containing the control vector of the key to be generated, or a null token. A null token can be supplied if the generated_key_identifier will be a DKYGENKY with a CV derived from the generating_key_identifier. A skeleton token or internal token is required when generated_key_identifier will not be a DKYGENKY key type or the processing method is not SESS-XOR. For SESS-XOR, this must be a null token. On output, this parameter contains the generated key.

Restrictions

The restrictions for CSNBDKG.

None

Required commands

The CSNBDKG required commands.

This verb requires the following commands to be enabled in the active role based on the keyword specified for the process rule:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR8-ENC</td>
<td>X'0040'</td>
<td>Diversified Key Generate - CLR8-ENC</td>
</tr>
<tr>
<td>SESS-XOR</td>
<td>X'0043'</td>
<td>Diversified Key Generate - SESS-XOR</td>
</tr>
<tr>
<td>TDES-DEC</td>
<td>X'0042'</td>
<td>Diversified Key Generate - TDES-DEC</td>
</tr>
<tr>
<td>TDES-ENC</td>
<td>X'0041'</td>
<td>Diversified Key Generate - TDES-ENC</td>
</tr>
<tr>
<td>TDES-XOR</td>
<td>X'0045'</td>
<td>Diversified Key Generate - TDES-XOR</td>
</tr>
<tr>
<td>TDESEMV2 or TDESEMV4</td>
<td>X'0046'</td>
<td>Diversified Key Generate - TDESEMV2/TDESEMV4</td>
</tr>
<tr>
<td>WRAP-ECB or WRAP-ENH and default key-wrapping method setting does not match keyword</td>
<td>X'013D'</td>
<td>Diversified Key Generate - Allow wrapping override keywords</td>
</tr>
</tbody>
</table>

When a key-generating key of key type DKYGENKY is specified with control vector bits (19 - 22) of B'1111', the Diversified Key Generate - DKYGENKY - DALL command (offset X'0290') must also be enabled in the active role.

Note: A role with offset X'0290' enabled can also use the PIN Change/Unblock verb with a DALL key.

When using the TDES-ENC or TDES-DEC modes, you can specifically enable generation of a single-length key or a double-length key with equal key-halves (an effective single-length key) by enabling the Diversified Key Generate - Single length or same halves command (offset X'0044').

Usage notes

Usage notes for CSNBDKG.

Refer to Appendix D, “Control vectors and changing control vectors with the Control Vector Translate verb,” on page 667 for information on the control vector bits for the DKG key generating key.
Diversified Key Generate (CSNBDKG)

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBDKGJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBDKGJ are shown here.

Format

```java
public native void CSNBDKGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] generating_key_identifier,
    hikmNativeInteger data_length,
    byte[] data,
    byte[] data_decrypting_key_identifier,
    byte[] generated_key_identifier);
```
Use the EC Diffie-Hellman verb to create symmetric key material from a pair of Elliptic Curve Cryptography (ECC) keys using the Elliptic Curve Diffie-Hellman (ECDH) protocol and “Z” - The "secret" material output from EC Diffie-Hellman process.

For more information, see "EC Diffie-Hellman key agreement models" on page 44.

ECDH is a key-agreement protocol that allows two parties, each having an elliptic curve public-private key pair, to establish a shared secret over an insecure channel. This shared secret is used to derive another symmetric key. The ECDH protocol is a variant of the Diffie-Hellman protocol using elliptic curve cryptography. ECDH derives a shared secret value from a secret key owned by an Entity A and a public key owned by an Entity B, when the keys share the same elliptic curve domain parameters. Entity A can be either the initiator of a key-agreement transaction, or the responder in a scheme. If two entities both correctly perform the same operations with corresponding keys as inputs, the same shared secret value is produced.

Both parties must create an ECC public-private key pair. See "PKA Key Token Build (CSNDPKB)" on page 488 and "PKA Key Generate (CSNDPKG)" on page 478. A key can be internal or external, as well as encrypted or in the clear. Both keys must have the same elliptic curve domain parameters (curve type and key size):

- Brainpool (key size 160, 192, 224, 256, 320, 384, or 512)
- Prime (key size 192, 224, 256, 384, or 521)

In addition to having the same elliptic curve domain parameters, the keys must have their key-usage field set to permit key establishment (either KEY-MGMT or KM-ONLY). See "ECC key token" on page 636.

To use this verb, specify the following:

- One to five rule-array keywords:
  - A required key-agreement keyword
  - An optional transport key-type (required if output KEK key identifier is a label) that identifies which key-storage dataset contains the output KEK key-token
  - An optional output key-type (required if output key identifier is a label) that identifies which key-storage dataset contains the output key-token
  - When the output is a DES key-token, an optional key-wrapping method and an optional translation control keyword
- The internal or external ECC key-token containing the private key (public-private key pair).
  If the private key is in an external key-token and is not in the clear, specify the internal KEK that was used to wrap the private key. Otherwise, specify a private KEK key-length of 0.
- An internal or external ECC key-token containing the public key (public key only or public-private key pair) of the other party.
  If the public key is in a key token that contains a public-private key pair, only the public-key portion is used. No attempt is made to decrypt the private key.
- From 8 - 64 bytes of party information data of the initiator and the responder entities, according to NIST SP800-56A Section 5.8
EC Diffie-Hellman (CSNDEDH)

- The number of bits of key material, from 64 - 256, to derive and place in the provided output key-token
- The length in bytes of the buffer for the output key-identifier
- An internal or external skeleton key-token to be used as input for the output key-token

The skeleton key-token must be an AES key or a DES key, as shown in the following table:

Table 27. Skeleton key-tokens

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Token version number</th>
<th>Key type (see note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>X'04' (legacy fixed-length symmetric key-token)</td>
<td>DATA</td>
</tr>
</tbody>
</table>
|           | X'05' (variable-length symmetric key-token) | • CIPHER
  Both parties can provide any combination of encryption or decryption for key-usage field.
  • EXPORTER or IMPORTER
  Both parties can provide any combination of EXPORTER or IMPORTER. |
| DES       | X'00', X'01', X'03' (legacy fixed-length symmetric key-token) | • CIPHER, DECIPHER, ENCIPHER
  Both parties can provide any combination of Encipher or Decipher key-usage bits in the control vector.
  • EXPORTER or IMPORTER
  Both parties can provide any combination of EXPORT or IMPORT key-usage bits in the control vector. |

Note: Except as otherwise noted, both parties must provide identical skeleton key tokens for the output key in order to derive identical keys. For legacy skeletons, control vector parity bits are not used in the key derivation process.

If the skeleton key-token is an external key-token, specify the internal KEK to be used to wrap the output key-token. Otherwise, specify an output KEK length of 0.

If the output_key_identifier specifies a DES key-token, then the output_KEY_KEY_TOKEN must identify a legacy DES KEK. Otherwise it must identify a variable-length symmetric AES KEK key-token.

The output from this verb can be one of the following formats:
- Internal CCA Token (DES or AES): AES keys are in the variable-length symmetric key token format. DES keys are in the DES external key token format.
- External CCA key token (DES or AES): AES keys are in the variable-length symmetric key token format. DES keys are in the DES external key token format.
- "Z"—The "secret" material output from EC Diffie-Hellman process.

The PASSTHRU service is provided as a convenience to users who wish to implement their own key completion process in host application software. While the "Z" derivation process is not reversible (the ECC keys cannot be discovered by obtaining "Z") there is a level of security compromise associated with returning the clear "Z" to the application. Future derivations for CCA key tokens using ECC keys previously used in PASSTHRU must be considered to have lower security, and using the same ECC keys for PASSTHRU as for DERIV01 is strongly discouraged. It should also be noted that since "Z" is the secret material, returning it in the clear
EC Diffie-Hellman (CSNDEDH)

to the host application reduces security level of the ‘Z’ from the HSM level to the
host application level, and keys made from this material should not be regarded as
having any HSM protection.

For more information, see “EC Diffie-Hellman key agreement models” on page 44.

Format

The format of CSNDEDH.

```
CSNDEDH(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  rule_array_count,
  rule_array,
  private_key_identifier_length,
  private_key_identifier,
  private_KEK_key_identifier_length,
  private_KEK_key_identifier,
  public_key_identifier_length,
  public_key_identifier,
  chaining_vector_length,
  chaining_vector,
  party_info_length,
  party_info,
  key_bit_length,
  reserved_1_length,
  reserved_1,
  reserved_2_length,
  reserved_2,
  reserved_3_length,
  reserved_3,
  reserved_4_length,
  reserved_4,
  reserved_5_length,
  reserved_5,
  output_KEK_key_identifier_length,
  output_KEK_key_identifier,
  output_key_identifier_length,
  output_key_identifier
)
```

Parameters

The parameter definitions for CSNDEDH.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data`
parameters, see “Parameters common to all verbs” on page 20.

`rule_array_count`

  Direction: Input
  Type: Integer

  A pointer to an integer variable containing the number of elements in the
  `rule_array` variable. Valid values are 1 - 5.

`rule_array`

  Direction: Input
  Type: String array
A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords for this verb are shown below:

**Table 28. Keywords for EC Diffie-Hellman control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key agreement</strong> (one required)</td>
<td></td>
</tr>
<tr>
<td>DERIV01</td>
<td>Use input skeleton key-token and derive one element of any key pair. Denotes ANSI X9.63 protocol static unified model key-agreement scheme (see NIST SP800-56A). Initiator and responder must have a sufficient level of trust such that they each derive only one element of any key pair.</td>
</tr>
<tr>
<td>PASSTHRU</td>
<td>Skip key derivation step and return raw “Z” material. Note: This keyword is available only for Linux on System z.</td>
</tr>
<tr>
<td><strong>Transport key-type</strong> (one, optional; one required if output_KEK_key_identifier is a label)</td>
<td></td>
</tr>
<tr>
<td>OKEK-AES</td>
<td>The output_KEK_key_identifier represents an AES key-token.</td>
</tr>
<tr>
<td>OKEK-DES</td>
<td>The output_KEK_key_identifier represents a DES key-token.</td>
</tr>
<tr>
<td><strong>Output key-type</strong> (one, optional; required if output_key_identifier is a label)</td>
<td></td>
</tr>
<tr>
<td>KEY-AES</td>
<td>The outbound key-encrypting key represents an AES skeleton key-token.</td>
</tr>
<tr>
<td>KEY-DES</td>
<td>The outbound key-encrypting key represents a DES skeleton key-token.</td>
</tr>
<tr>
<td><strong>Key-wrapping method</strong> (one, optional). DES only.</td>
<td></td>
</tr>
<tr>
<td>USECONFIG</td>
<td>Specifies to wrap the key using the configuration setting for the default wrapping method. This is the default.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys.</td>
</tr>
<tr>
<td><strong>Translation control</strong> (optional). This is valid only with key-wrapping method WRAP-ENH or with USECONFIG when the default wrapping method is WRAP-ENH. This option cannot be used on a key with a control vector valued to binary zeros.</td>
<td></td>
</tr>
<tr>
<td>ENH-ONLY</td>
<td>Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'.</td>
</tr>
</tbody>
</table>

**private_key_identifier_length**

- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of bytes in the `private_key_identifier` variable.

**private_key_identifier**

- **Direction:** Input
- **Type:** String

A pointer to a string variable containing an internal or external ECC key-token, or a key label identifying a key-storage record for such a token. The ECC key-token will contain a public-private key pair. Clear keys are allowed.

The ECC curve type and size must be the same as the type (Prime or Brainpool) and size of the ECC key-token specified by the `public_key_identifier`
EC Diffie-Hellman (CSNDEDH)

parameter. The key-usage flag of the ECC key-token identified by the
private_key_identifier parameter must permit key establishment (either
KEY-MGMT or KM-ONLY).

private_KEY_key_identifier_length

Direction: Input
Type: Integer
A pointer to an integer variable containing the number of bytes in the
private_KEY_key_identifier variable. The maximum value is 900. If the
private_key_identifier contains an internal ECC token, this value must be a zero.

private_KEY_key_identifier

Direction: Input
Type: String
A pointer to a string variable containing an internal KEK key-token, a key label
identifying a key-storage record for such a key token, or a null token. The KEK
key-token must be present if the key token specified by the private_key_identifier
contains an external encrypted ECC key-token.

public_key_identifier_length

Direction: Input/Output
Type: Integer
A pointer to an integer variable containing the number of bytes in the
public_key_identifier variable.

Note that even though this variable is not currently updated on output, it is
reserved as an output field for future use.

public_key_identifier

Direction: Input/Output
Type: String
A pointer to a string variable containing an ECC key-token, or a key label
identifying a key-storage record for such a key token. The ECC curve type and
size must be the same as the type and size of the ECC key-token specified by
the private_key_identifier parameter. The public_key_identifier specifies the other
party’s ECC public key which is enabled for key management functions. If the
public_key_identifier parameter identifies a key token containing a public-private
key pair, no attempt to decrypt the private part is made.

Note that even though this variable is not currently updated on output, it is
reserved as an output field for future use.

chaining_vector_length

Direction: Input/Output
Type: Integer
A pointer to an integer variable containing the number of bytes in the
chaining_vector variable. This field is currently not used. The value must be 0.

chaining_vector

Direction: Input/Output
Type: String
A pointer to a string variable containing a buffer that is currently reserved.

party_info_length
EC Diffie-Hellman (CSNDEDH)

Direction: Input/Output  
Type: Integer

A pointer to an integer variable containing the number of bytes in the 
party_info variable. Valid values are 0, or 8 - 64. The party_info_length must be 0 
when the PASSTHRU rule-array keyword is specified.

party_info

Direction: Input/Output  
Type: String

A pointer to a string variable containing combined entity identifier 
information, including nonce. This information must contain data of both 
entities according to NIST SP800-56A Section 5.8, when the DERIV01 
rule-array keyword is specified.

key_bit_length

Direction: Input  
Type: Integer

A pointer to an integer variable containing the number of bits of key material 
to derive and place in the provided output key-token. The value must be 0 if 
the PASSTHRU rule-array keyword is specified. The value must be 64 - 256.

reserved_1_length

Direction: Input/Output  
Type: Integer

A pointer to an integer variable containing the number of bytes in the 
reserved_1 variable. The value must be 0.

reserved_1

Direction: Input/Output  
Type: String

A pointer to a string variable that is currently not used.

reserved_2_length

Direction: Input/Output  
Type: Integer

A pointer to an integer variable containing the number of bytes in the 
reserved_2 variable. The value must be 0.

reserved_2

Direction: Input/Output  
Type: String

A pointer to a string variable that is currently not used.

reserved_3_length

Direction: Input/Output  
Type: Integer

A pointer to an integer variable containing the number of bytes in the 
reserved_3 variable. The value must be 0.

reserved_3

Direction: Input/Output  
Type: String
EC Diffie-Hellman (CSNDEDH)

A pointer to a string variable that is currently not used.

**reserved_4_length**
- Direction: Input/Output
- Type: Integer
- A pointer to an integer variable containing the number of bytes in the `reserved_4` variable. The value must be 0.

**reserved_4**
- Direction: Input/Output
- Type: String
- A pointer to a string variable that is currently not used.

**reserved_5_length**
- Direction: Input/Output
- Type: Integer
- A pointer to an integer variable containing the number of bytes in the `reserved_5` variable. The value must be 0.

**reserved_5**
- Direction: Input/Output
- Type: String
- A pointer to a string variable that is currently not used.

**output_KEK_key_identifier_length**
- Direction: Input
- Type: Integer
- A pointer to an integer variable containing the number of bytes in the `output_KEK_key_identifier` variable. The maximum value is 900. The `output_KEK_key_identifier_length` must be zero if `output_key_identifier` will contain an internal token or if the PASSTHRU rule-array keyword was specified.

**output_KEK_key_identifier**
- Direction: Input
- Type: String
- A pointer to a string variable containing an internal KEK key-token, or a key label identifying a key-storage record for such a token. This parameter must identify a KEK key-token whenever the `output_key_identifier` specifies an external key-token. If the `output_key_identifier` specifies a DES key-token, then the `output_KEK_key_identifier` must identify a legacy DES KEK, otherwise it must identify a variable-length symmetric AES KEK key-token.

If this variable contains a key label, specify a transport key-type rule-array keyword (OKEK-DES or OKEK-AES) to identify which key-storage dataset contains the key token. If a transport key-type keyword is specified, it must match the type of key identified by this parameter, whether the key is in key storage or not.

If the `output_KEK_key_identifier` specifies a legacy DES KEK, then the key token must contain either an EXPORTER control vector with bit 21 on (EXPORT) or an IMPORTER control vector with bit 21 set to B’1’ (IMPORT). The XLATE bit (bit 22) is not checked. Similarly, if the `output_KEK_key_identifier` identifies a variable-length symmetric AES KEK, then the KEK must be have a key type of
EC Diffie-Hellman (CSNDEDH)

EXPORTER or IMPORTER. Key-usage field 1 of the KEK must be set so that the key can be used for EXPORT or IMPORT. In addition, key-usage field 4 must be set so that the key can wrap DERIVATION class keys.

output_key_identifier_length
Direction: Input/Output
Type: Integer
A pointer to an integer variable containing the number of bytes in the output_key_identifier variable. On input, the output_key_identifier_length variable must be set to the total size of the buffer pointed to by the output_key_identifier parameter. On output, this variable contains the number of bytes of data returned by the verb in the output_key_identifier variable. The maximum allowed value is 900 bytes.

output_key_identifier
Direction: Input/Output
Type: String
A pointer to a string variable. On input, it must contain an internal or an external skeleton key-token, or a key label identifying a key-storage record for such a token. The skeleton key-token must be one of the following:

DES (legacy DES key-token)
- CIPHER, DECIPHER, or ENCIPHER
- EXPORTER or IMPORTER

AES
- DATA (legacy AES key-token)
- CIPHER (variable-length symmetric key-token) with key-usage field set so that the key can be used for decryption, encryption, or both
- EXPORTER or IMPORTER (variable-length symmetric key-token)

On successful completion, this variable contains either:
- An updated key-token that contains the generated symmetric key material, or the key label of the key-token that has been updated in key storage.
- "Z" data (in the clear) if the PASSTHRU rule-array keyword was specified.

If this variable contains an external key-token on input, then the output_kek_key_identifier is used to securely encrypt it for output. If this variable contains a key label, specify an output key-type rule-array keyword (KEY-DES or KEY-AES) to identify which key-storage dataset contains the key token. If an output key-type keyword is specified, it must match the type of key identified by this parameter, whether the key is in key storage or not.

If this variable identifies an external DES key-token, then the output_kek_key_identifier must identify a DES KEK key-token. If this variable is present and identifies an external key-token other than a DES key-token, then the output_kek_key_identifier must identify an AES KEK key-token.

Restrictions
The restrictions for CSNDEDH.

The NIST security strength requirements are enforced, with respect to ECC curve type (input) and derived key-length. See "Required commands" on page 161 about how you can override this enforcement.
This table lists the valid key bit lengths and the minimum curve size required for each of the supported output key types:

<table>
<thead>
<tr>
<th>Output key ID type</th>
<th>Valid key bit lengths</th>
<th>Minimum curve required</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>64</td>
<td>P160</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>P160</td>
</tr>
<tr>
<td></td>
<td>192</td>
<td>P224</td>
</tr>
<tr>
<td>AES</td>
<td>128</td>
<td>P256</td>
</tr>
<tr>
<td></td>
<td>192</td>
<td>P384</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>P512</td>
</tr>
</tbody>
</table>

**Required commands**

The CSNDEDH required commands.

This table describes access control points that the EC Diffie-Hellman verb must have enabled in the active role under certain circumstances.

<table>
<thead>
<tr>
<th>Command</th>
<th>Offset</th>
<th>When required</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECC Diffie-Hellmann Callabe Services</td>
<td>X'0360'</td>
<td>When using the EC Diffie-Hellman verb</td>
</tr>
<tr>
<td>ECC Diffie-Hellmann - Allow Configuration Override with Keyword in EDH command</td>
<td>X'0362'</td>
<td>If the output_key_identifier parameter identifies a DES key-token, and the wrapping method specified is not the default method</td>
</tr>
<tr>
<td>Disallow Weak Key Wrap With Key of a Different Curve Type</td>
<td>X'0328'</td>
<td>To disable the wrapping of a stronger key with a weaker key</td>
</tr>
<tr>
<td>Warn When Wrapping Weak Keys</td>
<td>X'032C'</td>
<td>To receive a warning against the wrapping of a stronger key with a weaker key</td>
</tr>
<tr>
<td>Prevent Weaker Keys from being Used to Generate Stronger Keys</td>
<td>X'036F'</td>
<td>To disable a weaker key from being used to generate a stronger key</td>
</tr>
<tr>
<td>EC Diffie-Hellman - Allow PASSTHRU</td>
<td>X'0361'</td>
<td>When specifying the PASSTHRU rule-array keyword.</td>
</tr>
</tbody>
</table>

Depending on curve type, each length of $p$ in bits contained in the ECC private-key section and the ECC public-key section must have the following command enabled in the active role:
EC Diffie-Hellman (CSNDEDH)

<table>
<thead>
<tr>
<th>Curve type</th>
<th>Length of prime ( p ) in bits</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brainpool</td>
<td>160 (X’00A0’)</td>
<td>X’0368’</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 160</td>
</tr>
<tr>
<td>Brainpool</td>
<td>192 (X’00C0’)</td>
<td>X’0369’</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 192</td>
</tr>
<tr>
<td>Brainpool</td>
<td>224 (X’00E0’)</td>
<td>X’036A’</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 224</td>
</tr>
<tr>
<td>Brainpool</td>
<td>256 (X’0100’)</td>
<td>X’036B’</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 256</td>
</tr>
<tr>
<td>Brainpool</td>
<td>320 (X’0140’)</td>
<td>X’036C’</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 320</td>
</tr>
<tr>
<td>Brainpool</td>
<td>384 (X’0180’)</td>
<td>X’036D’</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 384</td>
</tr>
<tr>
<td>Brainpool</td>
<td>512 (X’0200’)</td>
<td>X’036E’</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 512</td>
</tr>
<tr>
<td>Prime</td>
<td>192 (X’00C0’)</td>
<td>X’0363’</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 192</td>
</tr>
<tr>
<td>Prime</td>
<td>224 (X’00E0’)</td>
<td>X’0364’</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 224</td>
</tr>
<tr>
<td>Prime</td>
<td>256 (X’0100’)</td>
<td>X’0365’</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 256</td>
</tr>
<tr>
<td>Prime</td>
<td>384 (X’0180’)</td>
<td>X’0366’</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 384</td>
</tr>
<tr>
<td>Prime</td>
<td>521 (X’0209’)</td>
<td>X’0367’</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 521</td>
</tr>
</tbody>
</table>

Usage notes

Usage notes for CSNDEDH.

The PASSTHRU service is provided as a convenience to users who wish to implement their own key completion process in host application software. While the "Z" derivation process is not reversible (the ECC keys cannot be discovered by obtaining "Z") there is a level of security compromise associated with returning the clear "Z" to the application. Future derivations for CCA key tokens using ECC keys previously used in PASSTHRU must be considered to have lower security, and using the same ECC keys for PASSTHRU as for DERIV01 is strongly discouraged. It should also be noted that since "Z" is the secret material, returning it in the clear to the host application reduces security level of the "Z" from the HSM level to the host application level, and keys made from this material should not be regarded as having any HSM protection.

For more information, see “EC Diffie-Hellman key agreement models” on page 44.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDEDHJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDEDHJ are shown here.
Format

public native void CSNDEDHJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger private_key_identifier_length,
    byte[] private_key_identifier,
    hikmNativeInteger private_KEK_key_identifier_length,
    byte[] private_KEK_key_identifier,
    hikmNativeInteger public_key_identifier_length,
    byte[] public_key_identifier,
    hikmNativeInteger chaining_vector_length,
    byte[] chaining_vector,
    hikmNativeInteger party_info_length,
    byte[] party_info,
    hikmNativeInteger key_bit_length,
    hikmNativeInteger reserved_1_length,
    byte[] reserved_1,
    hikmNativeInteger reserved_2_length,
    byte[] reserved_2,
    hikmNativeInteger reserved_3_length,
    byte[] reserved_3,
    hikmNativeInteger reserved_4_length,
    byte[] reserved_4,
    hikmNativeInteger reserved_5_length,
    byte[] reserved_5,
    hikmNativeInteger output_KEK_key_identifier_length,
    byte[] output_KEK_key_identifier,
    hikmNativeInteger output_key_identifier_length,
    byte[] output_key_identifier);
Key Export (CSNBKEX)

Use the Key Export verb to re-encipher any type of key (except an IMP-PKA) from encryption under a master key variant to encryption under the same variant of an exporter key-encrypting key. The format of.

The re-enciphered key can be exported to another system.

If the key to be exported is a DATA key, the Key Export verb generates a key token with the same key length as the input token’s key.

This verb supports the no-export bit that the Prohibit Export verb sets in the internal token.

Format

The format of CSNBKEX.

```
CSNBKEX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_type,
    source_key_identifier,
    exporter_key_identifier,
    target_key_identifier)
```

Parameters

The parameter definitions for CSNBKEX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`key_type`

Direction: Input
Type: String

The parameter is an 8-byte field that contains either a key type value or the keyword TOKEN. The keyword is left-aligned and padded on the right with blanks.

If the key type is TOKEN, CCA determines the key type from the control vector (CV) field in the internal key token provided in the `source_key_identifier` parameter.

Key type values for the Key Export verb are:

<table>
<thead>
<tr>
<th>CIPHER</th>
<th>EXPORTER</th>
<th>OPINENC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>IMPORTER</td>
<td>PINGEN</td>
</tr>
<tr>
<td>DATAC</td>
<td>IKEYXLAT</td>
<td>PINVER</td>
</tr>
<tr>
<td>DATAM</td>
<td>IPINENC</td>
<td>TOKEN</td>
</tr>
<tr>
<td>DATAMV</td>
<td>MAC</td>
<td></td>
</tr>
<tr>
<td>DECIPHER</td>
<td>MACVER</td>
<td></td>
</tr>
<tr>
<td>ENCIPLAYER</td>
<td>OKEYXLAT</td>
<td></td>
</tr>
</tbody>
</table>

For information about the meaning of the key types, see Table 2 on page 40.
Key Export (CSNBKEX)

Direction: Input
Type: String

A 64-byte string of the internal key token that contains the key to be re-enciphered. This parameter must identify an internal key token in application storage, or a label of an existing key in the DES key storage file.

If you supply TOKEN for the key_type parameter, CCA looks at the control vector in the internal key token and determines the key type from this information. If you supply TOKEN for the key_type parameter and supply a label for this parameter, the label must be unique in the DES key storage file.

exporter_key_identifier

Direction: Input/Output
Type: String

A 64-byte string of the internal key token or key label that contains the exporter key-encrypting key. This parameter must identify an internal key token in application storage, or a label of an existing key in the key storage file.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key.

Control vectors are explained in "Control vector" on page 36 and the NOCV bit is shown in Table 153 on page 612.

target_key_identifier

Direction: Input/Output
Type: String

The 64-byte field external key token that contains the re-enciphered key. The re-enciphered key can be exchanged with another cryptographic system.

Restrictions

The restrictions for CSNBKEX.

For security reasons, requests will fail by default if they use an equal key halves exporter to export a key with unequal key halves. You must have access control point 'Key Export - Unrestricted' explicitly enabled if you want to export keys in this manner.

Required commands

The CSNBKEX required commands.

This verb requires the Key Export command (offset X'0013') to be enabled in the active role.

By also specifying the Key Export - Unrestricted command (offset X'0276'), you can permit a less secure mode of operation that enables an equal key-halves EXPORTER key-encrypting-key to export a key having unequal key-halves (key parity bits are ignored).
**Key Export (CSNBKEX)**

**Usage notes**

Usage notes for CSNBKEX.

For Key Export, you can use the following combinations of parameters:

- A valid key type in the `key_type` parameter and an internal key token in the `source_key_identifier` parameter. The key type must be equivalent to the control vector specified in the internal key token.

- A `key_type` parameter of `TOKEN` and an internal key token in the `source_key_identifier` parameter. The `source_key_identifier` can be a label with `TOKEN` only if the label name is unique in the key storage. The key type is extracted from the control vector contained in the internal key token.

- A valid key type in the `key_type` parameter, and a label in the `source_key_identifier` parameter.

If internal key tokens are supplied in the `source_key_identifier` or `exporter_key_identifier` parameters, the key in one or both tokens can be re-enciphered. This occurs if the master key was changed since the internal key token was last used. The return and reason codes that indicate this do not indicate which key was re-enciphered. Therefore, assume both keys have been re-enciphered.

Existing internal tokens created with key type `MACD` must be exported with either a `TOKEN` or `DATAM` key type. The external CV will be `DATAM` CV. The `MACD` key type is not supported.

To export a double-length MAC generation or MAC verification key, it is recommended that a key type of `TOKEN` be used.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBKEXJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKEXJ are shown here.

**Format**

```java
public native void CSNBKEXJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_type,
    byte[] source_key_identifier,
    byte[] exporter_key_identifier,
    byte[] target_key_token);
```
**Key Generate (CSNBKGN)**

Use the Key Generate verb to generate an AES key of type **DATA**, or either one or two odd parity DES keys of **any** type. The format of.

The DES keys can be single-length (8-byte), double-length (16-byte), or, in the case of **DATA** keys, triple-length (24-byte). The AES keys can be 16, 24 or 32 bytes in length. The Key Generate verb does not produce keys in clear form; all keys are returned in encrypted form. When two keys are generated (DES only), each key has the same clear value, although this clear value is not exposed outside the secure cryptographic feature.

For AES, the verb returns only one copy of the key, enciphered under the AES master key. For DES, the verb selectively returns one copy of the key or two, with each copy enciphered under a user-specified DES key-encrypting key.

This verb returns the key to the application program that called it and the application program can then use the CCA key storage verbs to store the key in the key storage file.

**Format**

The format of CSNBKGN.

```plaintext
CSNBKGN(    return_code, 
            reason_code,  
            exit_data_length,  
            exit_data,  
            key_form, 
            key_length, 
            key_type_1, 
            key_type_2, 
            kek_key_identifier_1, 
            kek_key_identifier_2,  
            generated_key_identifier_1, 
            generated_key_identifier_2)
```

**Parameters**

The parameter definitions for CSNBKGN.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**key_form**

**Direction: Input**

**Type: String**

A 4-byte keyword that defines the type of key you want generated. This parameter also specifies if each key should be returned for either operational, importable, or exportable use. The keyword must be in a 4-byte field, left-aligned, and padded with blanks.

The possible key forms are:

**Operational (OP)**

The key is used for cryptographic operations on the local system.
Key Generate (CSNBKGN)

Operational keys are protected by master key variants and can be stored in the CCA key storage file or held by applications in internal key tokens.

Importable (IM)
The key is stored with a file or sent to another system. Importable keys are protected by importer key-encrypting keys.

Exportable (EX)
The key is transported or exported to another system and imported there for use. Exportable keys are protected by exporter key-encrypting keys and cannot be used by CCA verb.

Importable and exportable keys are contained in external key tokens. For more information on key tokens, refer to "Key token" on page 31.

The first two characters refer to key_type_1. The next two characters refer to key_type_2.

The following keywords are allowed: OP, IM, EX, OPIM, OPEX, IMEX, EXEX, OPOP, and IMIM. See Table 29 for their meanings.

Table 29. Keywords for the Key Generate verb key_form parameter

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>One key that can be sent to another system.</td>
</tr>
<tr>
<td>EXEX</td>
<td>A key pair; both keys to be sent elsewhere, possibly for exporting to two different systems. The key pair has the same clear value.</td>
</tr>
<tr>
<td>IM</td>
<td>One key that can be locally imported. The key can later be imported onto this system to make it operational.</td>
</tr>
<tr>
<td>IMEX</td>
<td>A key pair to be imported; one key to be imported locally and one key to be sent elsewhere. Both keys have the same clear value.</td>
</tr>
<tr>
<td>IMIM</td>
<td>A key pair to be imported; both keys to be imported locally at a later time.</td>
</tr>
<tr>
<td>OP</td>
<td>One operational key. The key is returned to the caller in the key token format.</td>
</tr>
<tr>
<td>OPEX</td>
<td>A key pair; one key that is operational and one key to be sent from this system. Both keys have the same clear value.</td>
</tr>
<tr>
<td>OPIM</td>
<td>A key pair; one key that is operational and one key to be imported to the local system. Both keys have the same clear value. On the other system, the external key token can be imported to make it operational.</td>
</tr>
<tr>
<td>OPOP</td>
<td>A key pair; normally with different control vector values.</td>
</tr>
</tbody>
</table>

The key forms are defined as follows:

**Operational (OP)**
The key value is enciphered under a master key. The result is placed into an internal key token. The key is then operational at the local system.

**Importable (IM)**
The key value is enciphered under an importer key-encrypting key. The result is placed into an external key token.

**Exportable (EX)**
The key value is enciphered under an exporter key-encrypting key. The result is placed into an external key token. The key can then be transported or exported to another system and imported there for use. This key form cannot be used by any CCA verb.
The keys are placed into tokens that the generated_key_identifier_1 and generated_key_identifier_2 parameters identify.

Valid key type combinations depend on the key form. See Table 33 on page 174 for valid key combinations.

**key_length**

**Direction:** Input  
**Type:** String

An 8-byte value that defines the length of the key as being 8, 16, 24 or 32 bytes. The keyword must be left-aligned and padded on the right with blanks. You must supply one of the key length values in the key_length parameter.

Table 30 lists the key lengths used for various key types.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINGLE, SINGLE-R, or KEYLN8</td>
<td>Single length (8-byte or 64-bit) key</td>
<td>DES</td>
</tr>
<tr>
<td>DOUBLE or KEYLN16</td>
<td>Double length (16-byte or 128-bit) key</td>
<td>AES or DES</td>
</tr>
<tr>
<td>KEYLN24</td>
<td>Triple length (24-byte or 192-bit) key</td>
<td>AES or DES</td>
</tr>
<tr>
<td>KEYLN32</td>
<td>32-byte (256-bit) key</td>
<td>AES</td>
</tr>
</tbody>
</table>

AES keys allow only KEYLN16, KEYLN24, and KEYLN32. To generate a 128-bit AES key, specify key_length as KEYLN16. For 192-bit AES keys specify key_length as KEYLN24. A 256-bit AES key requires a key_length of KEYLN32. All AES keys are DATA keys.

Keys with a length of 32 bytes have four 8-byte key parts. This key length is valid only for AES keys. To generate a 32-byte AES key with four different values to be the basis of each key part, specify key_length as KEYLN32.

To generate a single-length key, specify key_length as SINGLE or KEYLN8.

Double-length (16-byte) keys have an 8-byte left half and an 8-byte right half. Both halves can have identical clear values or not. If you want the same value to be used in both key halves (called replicated key values), specify a key_length of SINGLE, SINGLE-R, or KEYLN8. If you want different values to be the basis of each key half, specify a key_length of DOUBLE or KEYLN16.

Triple-length (24-byte) keys have three 8-byte key parts. This key length is valid for DATA keys only. To generate a triple-length DATA key with three different values to be the basis of each key part, specify a key_length of KEYLN24.

Use SINGLE or SINGLE-R if you want to create a DES transport key that you would use to exchange DATA keys with a PCF system. Because PCF does not use double-length transport keys, specify SINGLE so that the effects of multiple encipherment are nullified.

When generating an AKEK, the key_length parameter is ignored. The AKEK key length (8-byte or 16-byte) is determined by the skeleton token created by the Key Token Build verb and provided in the generated_key_identifier_1 parameter.

The key length specified must be consistent with the key length indicated by the token you supply. For DES keys, this length is a field in the control vector. For AES keys, the length is an explicit field in the token. Table 31 on page 170
Key Generate (CSNBKGN)

shows the valid key lengths for each key type. An X indicates that a key length is permitted for a key type. A Y indicates that the key generated will be a double-length key with replicated key values. It is preferred that SINGLE-R be used for this result.

Table 31. Key Generate - key lengths for each key type

<table>
<thead>
<tr>
<th>Key Type</th>
<th>SINGLE (KEYLN8)</th>
<th>SINGLE-R</th>
<th>DOUBLE (KEYLN16)</th>
<th>Triple (KEYLN24)</th>
<th>(KEYLN32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AESTOKEN</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAC</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MACVER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DATAM</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DATAMV</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EXPORTER</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPORTER</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIPHER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECIPHER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCRYPTER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPINENC</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPINENC</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PINGEN</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PINVER</td>
<td>Y</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARDEC*</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARENC*</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARPINE*</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARXCVL*</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARXCVR*</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DKGGENKY*</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYGENKY*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Key types marked with an asterisk (*) are requested through the use of the TOKEN keyword and specifying a proper control vector in a key token.

key_type_1

Direction: Input
Type: String

An 8-byte keyword from the following group:

AEDATA DATAXLAT IPINENC PINVER
AESTOKEN DECIPHER MAC
CIPHER ENCIPHER MACVER
DATA EXPORTER OKEYXLAT
DATAM IKEYXLAT OPINENC
DATAMV IMPORTER PINGEN

or the keyword TOKEN.

For information on the meaning of the key types, see Table 2 on page 40.
Key Generate (CSNBKGN)

Use the `key_type_1` parameter for the first, or only key, that you want generated. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form.

If `key_type_1` is `TOKEN`, CCA examines the control vector (CV) field in the `generated_key_identifier_1` parameter to derive the key type. When `key_type_1` is `TOKEN`, CCA does not check for the length of the key for DATA keys. Instead, it uses the `key_length` parameter to determine the length of the key.

Use the `AESTOKEN` keyword for AES keys, or the `TOKEN` keyword for DES keys to indicate that the verb should determine the key type from the key token that you supply. For AES, all keys are type `AESDATA`. For DES, the key type is determined from the control vector in the key tokens. Alternatively, you can specify the key type using keywords shown in Table 32 on page 174 and Table 33 on page 174.

**Key types can have mandatory key forms.** For example, CVARENC keys must be generated in pairs with CVARDEC keys. The reason is that a CVARENC key can only be used for encryption, and without a CVARDEC key you cannot decrypt the data. See Table 32 on page 174 and Table 33 on page 174 for valid key type and key form combinations.

**key_type_2**

- **Direction:** Input
- **Type:** String

An 8-byte keyword from the following group:

```
AESTOKEN  DATAXLAT  IPINENC  PINVER
AESTOKEN  DECIPHER  MAC
CIPHER    ENCIPHER  MACVER
DATA      EXPORTER  OKEYXLAT
DATAM     IKEYXLAT  OPINENC
DATAMV    IMPORTER  PINGEN
```

or the keyword `TOKEN`.

For information on the meaning of the key types, see Table 2 on page 40.

Use the `key_type_2` parameter for a key pair, which is shown in Table 33 on page 174. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form.

If `key_type_2` is `TOKEN`, CCA examines the control vector (CV) field in the `generated_key_identifier_2` parameter to derive the key type. When `key_type_2` is `TOKEN`, CCA does not check for the length of the key for DATA keys. Instead, it uses the `key_length` parameter to determine the length of the key.

If you want only one key to be generated, specify the `key_type_2` and `KEK_key_identifier_2` as binary zeros.

See Table 32 on page 174 and Table 33 on page 174 for valid key type and key form combinations.

**KEK_key_identifier_1**

- **Direction:** Input/Output
- **Type:** String

A 64-byte string of an internal key token containing the importer or exporter key-encrypting key, or a key label. If you supply a key label that is less than 64-bytes, it must be left-aligned and padded with blanks. `KEK_key_identifier_1` is required for a `key_form` of IM, EX, IMEX, EXEX, or IMIM.
If the key_form is OP, OPEX, OPIM, or OPOP, the KEK_key_identifier_1 is null.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key.

Control vectors are explained in "Control vector" on page 36 and the NOCV bit is shown in Table 153 on page 612.

This parameter is not used when generating AES keys, and should point to null key-tokens.

**KEK_key_identifier_2**
- Direction: Input/Output
- Type: String

A 64-byte string of an internal key token containing the importer or exporter key-encrypting key, or a key label of an internal token. If you supply a key label that is less than 64-bytes, it must be left-aligned and padded with blanks. **KEK_key_identifier_2** is required for a key_form of OPIM, OPEX, IMEX, IMIM, or EXEX. This field is ignored for key_form keywords OP, IM and EX.

If the NOCV bit is on in the internal key token containing the key-encrypting key, the key-encrypting key itself (not the key-encrypting key variant) is used to encipher the generated key.

Control vectors are explained in "Control vector" on page 36 and the NOCV bit is shown in Table 153 on page 612.

This parameter is not used when generating AES keys, and should point to null key-tokens.

**generated_key_identifier_1**
- Direction: Input/Output
- Type: String

This parameter specifies either a generated:

- Internal key token for an operational key form, or
- External key token containing a key enciphered under the kek_key_identifier_1 parameter.

When key_type_1 parameter is AESDATA, the generated_key_identifier_1 parameter is ignored. In this case, it is recommended that the parameter be initialized to 64-bytes of X'00'.

If you specify a key_type_1 of TOKEN, then this field contains a valid token of the key type you want to generate. Otherwise, on input, this parameter must be binary zeros. See key_type_1 for a list of valid key types.

If you specify a key_type_1 of IMPORTER or EXPORTER and a key_form of OPEX, and if the generated_key_identifier_1 parameter contains a valid internal token of the same type, the NOCV bit, if on, is propagated to the generated key token.

Using the AESTOKEN or TOKEN keyword in the key type parameters requires that the key tokens already exist when the verb is called, so the information in those tokens can be used to determine the key type:

- The key_type_1 parameter overrides the type in the token.
- The key_length parameter overrides the length value in the generated key token.
Key Generate (CSNBKGN)

In general, unless you are using the AESTOKEN or TOKEN keyword, you must identify a null key token in the generated key identifier parameters on input.

**generated_key_identifier_2**

Direction: Input/Output  
Type: String

This parameter specifies a generated external key token containing a key enciphered under the kek_key_identifier_2 parameter.

If you specify a key_type_2 of TOKEN, then this field contains a valid token of the key type you want to generate. Otherwise, on input, this parameter must be binary zeros. See key_type_1 for a list of valid key types.

The token can be an internal or external token.

Using the AESTOKEN or TOKEN keyword in the key type parameters requires that the key tokens already exist when the verb is called, so the information in those tokens can be used to determine the key type. In general, unless you are using the AESTOKEN or TOKEN keyword, you must identify a null key token in the generated key identifier parameters on input.

**Restrictions**

The restrictions for CSNBKGN.

None

**Required commands**

The CSNBKGN required commands.

Depending on the key_type and key_form parameters selected, the verb could require one or more of these commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'008C'</td>
<td>Key Generate - OPIM_OPEX_IMEX, etc.</td>
</tr>
<tr>
<td>X'008E'</td>
<td>Key Generate - OP_IM_EX</td>
</tr>
<tr>
<td>X'00D7'</td>
<td>Key Generate - OPIM_OPEX_IMEX, etc. extended</td>
</tr>
<tr>
<td>X'00DB'</td>
<td>Key Generate - SINGLE-R</td>
</tr>
</tbody>
</table>

Note: A role with offset X'00DB' enabled can also use the Remote Key Export verb.

**Usage notes**

The usage notes for CSNBKGN.

Table 32 on page 174 shows the valid key type and key form combinations for a single key. Key types marked with an "*" must be requested through the specification of a proper control vector in a key token and through the use of the TOKEN keyword. See also Appendix C, “Key forms and types used in the Key Generate verb,” on page 663.

Note: Not all key types are valid on all hardware. See Table 2 on page 40.
Key Generate (CSNBKGN)

For AES keys, only key form OP is supported. AES keys cannot be generated in pairs.

Table 32. Keywords for Key Generate, valid key types and key forms for a single key

<table>
<thead>
<tr>
<th>Key Type 1</th>
<th>Key Type 2</th>
<th>OP</th>
<th>IM</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESDATA</td>
<td>Not applicable</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AESTOKEN</td>
<td>Not applicable</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>Not applicable</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAC*</td>
<td>Not applicable</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAM</td>
<td>Not applicable</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DKYGENKY*</td>
<td>Not applicable</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>KEYGENKY*</td>
<td>Not applicable</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAC</td>
<td>Not applicable</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PINGEN</td>
<td>Not applicable</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 33 shows the valid key type and key form combinations for a key pair.

Table 33. Keywords for Key Generate, valid key types and key forms for a key pair

<table>
<thead>
<tr>
<th>Key Type 1</th>
<th>Key Type 2</th>
<th>OPEX</th>
<th>EXEX</th>
<th>OPIM, OPOP, IMIM</th>
<th>IMEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPHER</td>
<td>CIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CIPHER</td>
<td>DECIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CIPHER</td>
<td>ENCIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CVARDEC*</td>
<td>CVARENC*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>CVARDECC*</td>
<td>CVARPINE*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>CVARENC*</td>
<td>CVARDEC*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>CVARENC*</td>
<td>CVARXCVL*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>CVARENC*</td>
<td>CVARXCVR*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>CVARXCVL*</td>
<td>CVARENC*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>CVARXCVR*</td>
<td>CVARENC*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>CVARPINE*</td>
<td>CVARDEC*</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>DATA</td>
<td>DATA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATA</td>
<td>DATAXLAT</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DATAC*</td>
<td>DATAC*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAM</td>
<td>DATAM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAM</td>
<td>DATAMV</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAXLAT</td>
<td>DATAXLAT</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DECIPHER</td>
<td>CIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DECIPHER</td>
<td>ENCIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DKYGENKY*</td>
<td>DKYGENKY*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ENCIPHER</td>
<td>CIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ENCIPHER</td>
<td>DECIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>IKETXLAT</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
### Table 33. Keywords for Key Generate, valid key types and key forms for a key pair (continued)

<table>
<thead>
<tr>
<th>Key Type 1</th>
<th>Key Type 2</th>
<th>OPEX</th>
<th>EXEX</th>
<th>OPIM, OPOP, IMIM</th>
<th>IMEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPORTER</td>
<td>IMPORTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>EXPORTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>OKEYXLAT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>EXPORTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>OKEYXLAT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IPINENC</td>
<td>OPINENC</td>
<td>X</td>
<td>X</td>
<td>E</td>
<td>X</td>
</tr>
<tr>
<td>KEYGENKY*</td>
<td>KEYGENKY*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAC</td>
<td>MAC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAC</td>
<td>MACVER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>IKEYXLAT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>IMPORTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OPINENC</td>
<td>IPINENC</td>
<td>X</td>
<td>X</td>
<td>E</td>
<td>X</td>
</tr>
<tr>
<td>OPINENC</td>
<td>OPINENC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PINVER</td>
<td>PINGEN</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PINGEN</td>
<td>PINVER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Note:**
1. AES keys cannot be generated in pairs.
2. An 'X' indicates a permissible key type combination for a given key form. An 'E' indicates that a special (Extended) command is required as those keys require special handling.
3. The key types marked with an '*' must be requested through the specification of a proper control-vector in a key token and the use of the **TOKEN** keyword.

### JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKGNJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBKGNJ are shown here.

**Format**

```java
public native void CSNBKGNJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_form,
    byte[] key_length,
    byte[] key_type_1,
    byte[] key_type_2,
    byte[] KEK_key_identifier_1,
    byte[] KEK_key_identifier_2,
    byte[] generated_key_identifier_1,
    byte[] generated_key_identifier_2);
```
Key Generate2 (CSNBKGN2)

Use the Key Generate2 verb to generate either one or two keys of any type.

This verb does not produce keys in clear form and all keys are returned in encrypted form. When two keys are generated, each key has the same clear value, although this clear value is not exposed outside the secure cryptographic feature.

This verb returns variable-length CCA key tokens and uses the AESKW wrapping method.

This verb supports AES and HMAC keys. Operational keys will be encrypted under the AES master key.

Format

The format of CSNBKGN2.

```
CSNBKGN2(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    clear_key_bit_length,
    key_type_1,
    key_type_2,
    key_name_1_length,
    key_name_1,
    key_name_2_length,
    key_name_2,
    user_associated_data_1_length,
    user_associated_data_1,
    user_associated_data_2_length,
    user_associated_data_2,
    key_encrypting_key_identifier_1_length,
    key_encrypting_key_identifier_1,
    key_encrypting_key_identifier_2_length,
    key_encrypting_key_identifier_2,
    generated_key_identifier_1_length,
    generated_key_identifier_1,
    generated_key_identifier_2_length,
    generated_key_identifier_2)
```

Parameters

The parameter definitions for CSNBKGN2.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

- **Direction:** Input
- **Type:** Integer

The number of keywords you supplied in the `rule_array` parameter. This value must be 2.

`rule_array`
The *rule_array* contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The *rule_array* keywords are described in Table 34.

**Table 34. Keywords for Key Generate2 control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token algorithm</strong> (Required)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Specifies to generate an AES key token.</td>
</tr>
<tr>
<td>HMAC</td>
<td>Specifies to generate an HMAC key token.</td>
</tr>
<tr>
<td><strong>Key form</strong> (One, required) The first two characters refer to <em>key_type_1</em>. The next two characters refer to <em>key_type_2</em>. See &quot;Usage notes&quot; on page 181 for details.</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>One key that can be exported to another system.</td>
</tr>
<tr>
<td>EXEX</td>
<td>A key pair. Both keys are to be exported, possibly to two different systems. Both keys have the same clear value.</td>
</tr>
<tr>
<td>IM</td>
<td>One key that can be locally imported. The key can be imported onto this system to make it operational at another time.</td>
</tr>
<tr>
<td>IMEX</td>
<td>A key pair to be imported. One key is to be imported locally, and one key to be exported. Both keys have the same clear value.</td>
</tr>
<tr>
<td>IMIM</td>
<td>A key pair to be imported. Both keys are to be imported locally at another time. Both keys have the same clear value.</td>
</tr>
<tr>
<td>OP</td>
<td>One operational key. The key is returned to the caller in operational form to be used locally.</td>
</tr>
<tr>
<td>OPEX</td>
<td>A key pair. One key that is operational, and one key to be exported. Both keys have the same clear value.</td>
</tr>
<tr>
<td>OPIM</td>
<td>A key pair. One key that is operational, and one key to be imported locally at another time. Both keys have the same clear value.</td>
</tr>
<tr>
<td>OPOP</td>
<td>A key pair. Either with the same key type with different associated data, or complementary key types. Both keys have the same clear value.</td>
</tr>
</tbody>
</table>

**clear_key_bit_length**

**Direction: Input**

**Type: Integer**

A pointer to an integer variable containing the number of clear-key bits to randomly generate and return encrypted in the generated key or keys. If a generated key token has a key type of **TOKEN**, this value overrides any key length contained in the key token. The value can be 128, 192, and 256 for AES keys, and 80 - 2048 for HMAC keys.

**key_type_1**

**Direction: Input**

**Type: String**

Use the *key_type_1* parameter for the first, or only, key that you want generated. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form, and are documented in Table 37 on page 182 and Table 38 on page 182.

The 8-byte keyword for the *key_type_1* parameter can be one of the following:
Key Generate2 (CSNBKGN2)

Table 35. Keywords and associated algorithms for key_type_1 parameter

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPHER</td>
<td>AES</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>AES</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>AES</td>
</tr>
<tr>
<td>MAC</td>
<td>HMAC</td>
</tr>
<tr>
<td>MACVER</td>
<td>HMAC</td>
</tr>
</tbody>
</table>

Specify the keyword TOKEN when supplying a key token in the generated_key_identifier_1 parameter.

If key_type_1 is TOKEN, the associated data in the generated_key_identifier_1 parameter is used to derive the key type.

key_type_2

Direction: Input
Type: String

Use the key_type_2 parameter for a key pair, which is shown in Table 38 on page 182. The keyword must be left-aligned and padded with blanks. Valid type combinations depend on the key form.

The 8-byte keyword for the key_type_2 parameter can be one of the following:

Table 36. Keywords and associated algorithms for key_type_2 parameter

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPHER</td>
<td>AES</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>AES</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>AES</td>
</tr>
<tr>
<td>MAC</td>
<td>HMAC</td>
</tr>
<tr>
<td>MACVER</td>
<td>HMAC</td>
</tr>
</tbody>
</table>

Specify the keyword TOKEN when supplying a key token in the generated_key_identifier_2 parameter.

If key_type_2 is TOKEN, the associated data in the generated_key_identifier_2 parameter is used to derive the key type.

When only one key is being generated, this parameter is ignored.

key_name_1_length

Direction: Input
Type: Integer

The length of the key_name parameter for generated_key_identifier_1. Valid values are 0 and 64.

key_name_1

Direction: Input
Type: String

A 64-byte key store label to be stored in the associated data structure of generated_key_identifier_1.
Key Generate2 (CSNBKGN2)

Direction: Input
Type: Integer

The length of the key_name parameter for generated_key_identifier_2. Valid values are 0 and 64. When only one key is being generated, this parameter is ignored.

key_name_2

Direction: Input
Type: String

A 64-byte key store label to be stored in the associated data structure of generated_key_identifier_2.

When only one key is being generated, this parameter is ignored.

user_associated_data_1_length

Direction: Input
Type: Integer

The length of the user-associated data parameter for generated_key_identifier_1. The valid values are 0 - 255 bytes.

user_associated_data_1

Direction: Input
Type: String

User-associated data to be stored in the associated data structure for generated_key_identifier_1.

user_associated_data_2_length

Direction: Input
Type: Integer

The length of the user-associated data parameter for generated_key_identifier_2. The valid values are 0 - 255 bytes. When only one key is being generated, this parameter is ignored.

user_associated_data_2

Direction: Input
Type: String

User associated data to be stored in the associated data structure for generated_key_identifier_2.

When only one key is being generated, this parameter is ignored.

key_encrypting_key_identifier_1_length

Direction: Input
Type: Integer

The length of the buffer for key_encrypting_key_identifier_1 in bytes. When the key form rule is OP, OPOP, OPIM, or OPEX, this length must be zero. When the key form rule is EX, EXEX, IM, IMEX, or IMIM, the value must be between the actual length of the token and 725 bytes when key_encrypting_key_identifier_1 is a token.

The value must be 64 bytes when key_encrypting_key_identifier_1 is a label.

key_encrypting_key_identifier_1

Direction: Input
Type: String
Key Generate2 (CSNBKGN2)

When `key_encrypting_key_identifier_1_length` is zero, this parameter is ignored. Otherwise, `key_encrypting_key_identifier_1` contains an internal key token containing the AES importer or exporter key-encrypting key, or a key label.

If the token supplied was encrypted under the old master key, the token will be returned encrypted under the current master key.

`key_encrypting_key_identifier_2_length`

Direction: Input
Type: Integer

The length of the buffer for `key_encrypting_key_identifier_2` in bytes. When the key form rule is OPOP, this length must be zero. When the key form rule is EXEX, IMEX, IMIM, OPIM, or OPEX, the value must be between the actual length of the token and 725 when `key_encrypting_key_identifier_2` is a token. The value must be 64 when `key_encrypting_key_identifier_2` is a label.

When only one key is being generated, this parameter is ignored.

`key_encrypting_key_identifier_2`

Direction: Input/Output
Type: String

When `key_encrypting_key_identifier_2_length` is zero, this parameter is ignored. Otherwise, `key_encrypting_key_identifier_2` contains an internal key token containing the AES importer or exporter key-encrypting key, or a key label.

If the token supplied was encrypted under the old master key, the token will be returned encrypted under the current master key.

When only one key is being generated, this parameter is ignored.

`generated_key_identifier_1_length`

Direction: Input/Output
Type: Integer

On input, the length of the buffer for the `generated_key_identifier_1` parameter in bytes. The maximum value is 900 bytes.

On output, the parameter will hold the actual length of the `generated_key_identifier_1`.

`generated_key_identifier_1`

Direction: Input/Output
Type: String

The buffer for the first generated key token.

On input, if you specify a `key_type_1` of TOKEN, then the buffer contains a valid key token of the key type you want to generate. The key token must be left-aligned in the buffer. Otherwise, this parameter must be binary zeros. See `key_type_1` on page 177 for a list of valid key types.

On output, the buffer contains the generated key token.

`generated_key_identifier_2_length`

Direction: Input/Output
Type: Integer

On input, the length of the buffer for the `generated_key_identifier_2` in bytes. The minimum value is 120 bytes and the maximum value is 725 bytes. The maximum value is 900 bytes.
Key Generate2 (CSNBKGN2)

On output, the parameter will hold the actual length of the generated_key_identifier_2.

When only one key is being generated, this parameter is ignored.

generated_key_identifier_2

Direction: Input/Output
Type: String

The buffer for the second generated key token.

On input, if you specify a key_type_2 of TOKEN, then the buffer contains a valid key token of the key type you want to generate. The key token must be left-aligned in the buffer. Otherwise, this parameter must be binary zeros. See key_type_2 on page 178 for a list of valid key types.

On output, the buffer contains the generated key token.

When only one key is being generated, this parameter is ignored

Restrictions
The restrictions for CSNBKGN2.

This verb was introduced with CCA 4.1.0.

Required commands
The CSNBKGN2 required commands.

Depending on the key_type and key_form parameters selected, the verb could require one or more of these commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'00EA'</td>
<td>Key Generate2 - OP_EX_IM</td>
</tr>
<tr>
<td>X'00EB'</td>
<td>Key Generate2 - OPOP_OPIM_OPEX_ etc.</td>
</tr>
</tbody>
</table>

To disallow the wrapping of a key with a weaker key-encrypting key, enable the Disallow Weak Key Wrap command (offset X'0328') in the active role. This command affects multiple verbs. See Appendix G, “Access control points and verbs,” on page 723.

To receive a warning when wrapping a key with a weaker key-encrypting key, enable the Warn when Wrapping Weak Keys command (offset X'032C') in the active role. The Disallow Weak Key Wrap command (offset X'0328') overrides this command.

Usage notes
Usage notes for CSNBKGN2.

The key forms are defined as follows:

Operational (OP)
The key value is enciphered under a master key. The result is placed into an internal key token. The key is then operational at the local system.

Importable (IM)
The key value is enciphered under an importer key-encrypting key. The
Key Generate2 (CSNBKGN2)

result is placed into an external key token. The corresponding
key_encrypting_key_identifier_n parameter must contain an AES IMPORTER
key token or label.

Exportable (EX)
The key value is enciphered under an exporter key-encrypting key. The
result is placed into an external key token. The corresponding
key_encrypting_key_identifier_n parameter must contain an AES EXPORTER
key token or label.

These tables list the valid key type and key form combinations.

Table 37. Key Generate2 valid key type and key form for one key

<table>
<thead>
<tr>
<th>key_type_1</th>
<th>Key Form OP, IM, EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPHER</td>
<td>X</td>
</tr>
<tr>
<td>MAC</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 38. Key Generate2 valid key type and key forms for two keys

<table>
<thead>
<tr>
<th>key_type_1</th>
<th>key_type_2</th>
<th>Key form OPOP, OPIM, or IMIM</th>
<th>Key Form OPEX, EXEX, or IMEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPHER</td>
<td>CIPHER</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAC</td>
<td>MAC</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAC</td>
<td>MACVER</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MACVER</td>
<td>MAC</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>IMPORTER</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IMPORTER</td>
<td>EXPORTER</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

For AES keys, the AES KEK must be at least as strong as the key being generated
to be considered sufficient strength.

For HMAC keys, the AES KEK must be sufficient strength as described in the
following table:

Table 39. AES KEK strength required for generating an HMAC key under an AES KEK

<table>
<thead>
<tr>
<th>Key-usage field 2 in the HMAC key contains</th>
<th>Minimum strength of AES KEK to adequately protect the HMAC key</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-256, SHA-384, or SHA-512</td>
<td>256 bits</td>
</tr>
<tr>
<td>SHA-224</td>
<td>192 bits</td>
</tr>
<tr>
<td>SHA-1</td>
<td>128 bits</td>
</tr>
</tbody>
</table>

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKGN2J.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKGN2J are shown here.
Key Generate2 (CSNBKGN2)

Format

public native void CSNBKGN2J{
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] clear_key_bit_length,
    byte[] key_type_1,
    byte[] key_type_2,
    byte[] key_name_1_length,
    byte[] key_name_1,
    byte[] key_name_2_length,
    byte[] key_name_2,
    byte[] user_associated_data_1_length,
    byte[] user_associated_data_1,
    byte[] user_associated_data_2_length,
    byte[] user_associated_data_2,
    byte[] key_encrypting_key_identifier_1_length,
    byte[] key_encrypting_key_identifier_1,
    byte[] key_encrypting_key_identifier_2_length,
    byte[] key_encrypting_key_identifier_2,
    byte[] generated_key_identifier_1_length,
    byte[] generated_key_identifier_1,
    byte[] generated_key_identifier_2_length,
    byte[] generated_key_identifier_2);

Key Import (CSNBKIM)

Use the Key Import verb to re-encipher a key from encryption under an importer key-encrypting key to encryption under the master key.

The re-enciphered key is in operational form.

Choose one of the following options:

- Specify the key_type parameter as TOKEN and specify the external key token in the source_key_identifier parameter. The key type information is determined from the control vector in the external key token.
- Specify a key type in the key_type parameter and specify an external key token in the source_key_identifier parameter. The specified key type must be compatible with the control vector in the external key token.
- Specify a valid key type in the key_type parameter and a null key token in the source_key_identifier parameter. The default control vector for the key_type specified will be used to process the key.

For DATA keys, this verb generates a key of the same length as that contained in the input token.

Format

The format of CSNBKIM.

```
CSNBKIM(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  key_type,
  source_key_identifier,
  importer_key_identifier,
  target_key_identifier)
```

Parameters

The parameter definitions for CSNBKIM.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

key_type

Direction: Input
Type: String

The type of key you want to re-encipher under the master key. Specify an 8-byte keyword or the keyword TOKEN. The keyword must be left-aligned and padded on the right with blanks.

If the key type is TOKEN, CCA determines the key type from the control vector (CV) field in the external key token provided in the source_key_identifier parameter.

The key type of TOKEN is not allowed when the importer_key_identifier parameter is NOCV.

Key type values for the Key Import verb are:
For information on the meaning of the key types, see Table 2 on page 40.

We recommend using key type of **TOKEN** when importing double-length **MAC** and **MACVER** keys.

**source_key_identifier**

**Direction:** Input

**Type:** String

The key you want to re-encipher under the master key. The parameter is a 64-byte field for the enciphered key to be imported containing either an external key token or a null key token. If you specify a null token, the token is all binary zeros, except for a key in bytes 16-23 or 16-31, or in bytes 16-31 and 48-55 for triple-length **DATA** keys. Refer to Table 156 on page 615.

If key type is **TOKEN**, this field might not specify a null token.

This verb supports the no-export function in the CV.

**importer_key_identifier**

**Direction:** Input/Output

**Type:** String

The importer key-encrypting key that the key is currently encrypted under. The parameter is a 64-byte area containing either the key label of the key in the cryptographic key data set or the internal key token for the key. If you supply a key label that is less than 64-bytes, it must be left-aligned and padded with blanks.

**Note:** If you specify a NOCV importer in the importer_key_identifier parameter, the key to be imported must be enciphered under the importer key itself.

**target_key_identifier**

**Direction:** Input/Output

**Type:** String

This parameter is the generated re-enciphered key. The parameter is a 64-byte area that receives the internal key token for the imported key.

If the imported key type is **IMPORTER** or **EXPORTER** and the token key type is the same, the target_key_identifier parameter changes direction to both input and output. If the application passes a valid internal key token for an **IMPORTER** or **EXPORTER** key in this parameter, the NOCV bit is propagated to the imported key token.

**Restrictions**

The restrictions for CSNBKIM.

For security reasons, requests will fail by default if they use an equal key halves importer to import a key with unequal key halves. You must have access control point ‘Key Import - Unrestricted’ explicitly enabled if you want to import keys in this manner.
Key Import (CSNBKIM)

**Required commands**

The required commands for CSNBKIM.

This verb requires the Key Import command (offset X'0012') to be enabled in the active role.

By also enabling the Key Import - Unrestricted command (offset X'027B'), you can permit a less secure mode of operation that enables an equal key-halves **IMPORTER** key-encrypting key to import a key having unequal key-halves (key parity bits are ignored).

**Usage notes**

Usage notes for CSNBKIM.

Use of NOCV keys are controlled by an access control point in the CEX*C. Creation of NOCV key-encrypting keys is available only for standard IMPORTERs and EXPORTERs.

This verb will mark an imported KEK as a NOCV-KEK KEK:

- If a token is supplied in the target token field, it must be a valid importer or exporter token. If the token fails token validation, processing continues, but the NOCV flag will not be copied
- The source token (key to be imported) must be a importer or exporter with the default control vector.
- If the target token is valid and the NOCV flag is on and the source token is valid and the control vector of the target token is exactly the same as the source token, the imported token will have the NOCV flag set on.
- If the target token is valid and the NOCV flag is on and the source token is valid and the control vector of the target token is NOT exactly the same as the source token, a return code will be given.
- All other scenarios will complete successfully, but the NOCV flag will not be copied

The software bit used to mark the imported token with export prohibited is not supported on a CEX*C. The internal token for an export prohibited key will have the appropriate control vector that prohibits export.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBKIMJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKIMJ are shown here.

**Format**

```java
public native void CSNBKIMJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_type,
    byte[] source_key_token,
    byte[] importer_key_identifier,
    byte[] target_key_identifier );
```
Use the Key Part Import verb to combine, by XORing, the clear key parts of any key type and return the combined key value either in an internal token or as an update to the key storage file.

Before you use the Key Part Import verb for the first key part, you must use the Key Token Build or Key Token Build2 verb to create the internal key token into which the key will be imported. Subsequent key parts are combined with the first part in internal token form or as a label from the key storage file.

The preferred way to specify key parts is FIRST, ADD-PART, and COMPLETE in the rule_array. Only when the combined key parts have been marked as complete can the key token be used in any cryptographic operation. The partial key can be passed to the Key Token Change or Key Token Change2 verb for re-encipherment, in case building the key was started during a master key change operation. The partial key can be passed to the Key Token Parse verb, in order to discover how the key token was originally specified, if researching an old partial key. Partial keys can also be passed to the Key Test, Key Test2, and Key Test Extended verbs.

Key parts can also be specified as FIRST, MIDDLE, or LAST in the rule_array. ADD-PART or MIDDLE can be executed multiple times for as many key parts as necessary. Only when the LAST part has been combined can the key token be used in any other service.

New applications should employ the ADD-PART and COMPLETE keywords in lieu of the MIDDLE and LAST keywords in order to ensure a separation of responsibilities between someone who can add key-part information and someone who can declare that appropriate information has been accumulated in a key.

The Key Part Import verb can also be used to import a key without using key parts. Call the Key Part Import verb FIRST with key part value X’0000...’ then call the Key Part Import verb LAST with the complete value.

Keys created using this service have odd parity. The FIRST key part is adjusted to odd parity. All subsequent key parts are adjusted to even parity before being combined.

**Format**

The format of CSNBKPI.

```
CSNBKPI(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_part,
    key_identifier)
```

**Parameters**

The parameter definitions for CSNBKPI.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.
Key Part Import (CSNBKPI)

**rule_array_count**
- **Direction:** Input
- **Type:** Integer
  
  A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1 or 2.

**rule_array**
- **Direction:** Input
- **Type:** String

  The keyword that provides control information to the verb. The keywords must be eight bytes of contiguous storage with the keyword left-aligned in its 8-byte location and padded on the right with blanks. The `rule_array` keywords are described in Table 40.

### Table 40. Keywords for Key Part Import control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key part</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>FIRST</td>
<td>This keyword specifies that an initial key part is being entered. This verb returns this key-part encrypted by the master key in the key token that you supplied.</td>
</tr>
<tr>
<td>ADD-PART</td>
<td>This keyword specifies that additional key-part information is provided.</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>This keyword specifies that the key-part bit shall be turned off in the control vector of the key rendering the key fully operational. Note that no key-part information is added to the key with this keyword.</td>
</tr>
<tr>
<td>MIDDLE</td>
<td>This keyword specifies that an intermediate key part, which is neither the first key part nor the last key part, is being entered. Note that the command control point for this keyword is the same as that for the LAST keyword and different from that for the ADD-PART keyword.</td>
</tr>
<tr>
<td>LAST</td>
<td>This keyword specifies that the last key part is being entered. The key-part bit is turned off in the control vector.</td>
</tr>
<tr>
<td>RETRKPR</td>
<td>A key label must be passed as the <code>key_identifier</code>. This key label corresponds to a key stored in a KPIT register inside the crypto-card (not in host key storage). The key in that register has been loaded by label and key part using the KPIT verb by the TKE. This keyword for KPI allows the user to tell the card to wrap that key (it must be in the complete state) using the master key, place it in an internal token, and return that token to the user. This keyword applies only when using IBM System z.</td>
</tr>
</tbody>
</table>

**Key-wrapping method** (One, optional)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USECONFIG</td>
<td>Specifies to wrap the key using the configuration setting for the default wrapping method. This keyword is ignored for AES keys. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys. This keyword was introduced with CCA 4.1.0.</td>
</tr>
</tbody>
</table>

**key_part**
- **Direction:** Input
- **Type:** String

  A 16-byte field containing the clear key part to be entered. If the key is a single-length key, the key part must be left-aligned and padded on the right with zeros. This field is ignored if COMPLETE is specified.
Key Part Import (CSNBKPI)

**key_identifier**

**Direction:** Input/Output  
**Type:** String

A 64-byte field containing an internal token or a label of an existing key in the key storage file. If `rule_array` is **FIRST**, this field is the skeleton of an internal token of a single- or double-length key with the **KEY-PART** marking. If `rule_array` is **MIDDLE** or **LAST**, this is an internal token or key label of a partially combined key. Depending on the input format, the accumulated partial or complete key is returned as an internal token or as an updated key storage file record. The returned `key_identifier` will be encrypted under the current master key.

**Restrictions**

The restrictions for CSNBKPI.

If a label is specified on `key_identifier`, the label must be unique. If more than one record is found, the verb fails.

You must have access control point 'Key Part Import - Unrestricted' explicitly enabled. Otherwise, current applications will fail with either of the following conditions:

- The first eight bytes of `key_identifier` is different than the second eight bytes AND the first eight bytes of the combined key are the same as the last second eight bytes
- The first eight bytes of `key_identifier` is the same as the second eight bytes AND the first eight bytes of the combined key are different than the second eight bytes.

**Required commands**

The required commands for CSNBKPI.

This verb requires the following commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>X'001B'</td>
<td>Key Part Import - first key part</td>
</tr>
<tr>
<td>ADD-PART</td>
<td>X'0278'</td>
<td>Key Part Import - ADD-PART</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>X'0279'</td>
<td>Key Part Import - COMPLETE</td>
</tr>
<tr>
<td>MIDDLE or LAST</td>
<td>X'001C'</td>
<td>Key Part Import - middle and last</td>
</tr>
<tr>
<td>MIDDLE or LAST</td>
<td>X'027A'</td>
<td>Key Part Import - Unrestricted</td>
</tr>
<tr>
<td>WRAP-ECB or WRAP-ENH used, and default key-wrapping method setting does not match keyword</td>
<td>X'0140'</td>
<td>Key Part Import - Allow wrapping override keywords</td>
</tr>
</tbody>
</table>

**Usage notes**

The usage notes for CSNBKPI.

None
Key Part Import (CSNBKPI)

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKPIJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKPIJ are shown here.

Format

```java
public native void CSNBKPIJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array, 
    byte[] key_part,
    byte[] key_identifier);
```
Key Part Import2 (CSNBKPI2)

Use the Key Part Import2 verb to combine, by XORing, the clear key parts of any key type and return the combined key value either in a variable-length internal key token or as an update to the key storage file.

Before you use the Key Part Import2 verb for the first key part, you must use the Key Token Build2 verb to create the variable-length internal key token into which the key will be imported. Subsequent key parts are combined with the first part in variable-length internal key token form, or as a label from the key storage file.

The preferred way to specify key parts is FIRST, ADD-PART, and COMPLETE in the rule_array. Only when the combined key parts have been marked as complete can the key token be used in any cryptographic operation. The partial key can be passed to the Key Token Change2 verb for re-encipherment, in case building the key was started during a master key change operation. The partial key can be passed to the Key Token Parse verb, in order to discover how the key token was originally specified, if researching an old partial key. Partial keys can also be passed to the Key Test, Key Test2, and Key Test Extended verbs.

Key parts can also be specified as FIRST, MIDDLE, or LAST in the rule_array. ADD-PART or MIDDLE can be executed multiple times for as many key parts as necessary. Only when the LAST part has been combined can the key token be used by any other verb.

New applications should employ the ADD-PART and COMPLETE keywords in lieu of the MIDDLE and LAST keywords in order to ensure a separation of responsibilities between someone who can add key-part information and someone who can declare that appropriate information has been accumulated in a key.

On each call to Key Part Import2 (except with the COMPLETE keyword), specify the number of bits to use for the clear key part. Place the clear key part in the key_part parameter, and specify the number of bits using the key_part_length variable. Any extraneous bits of key_part data will be ignored.

Consider using the Key Test2 verb to ensure a correct key value has been accumulated prior to using the COMPLETE option to mark the key as fully operational.

Format

The format of CSNBKPI2.

```plaintext
CSNBKPI2(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_part_bit_length,
    key_part,
    key_identifier_length,
    key_identifier)
```

Parameters

The parameters for CSNBKPI2.
Key Part Import2 (CSNBKPI2)

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

**rule_array_count**

*Direction: Input*

*Type: Integer*

The number of keywords you supplied in the rule_array parameter. This value must be 2 or 3.

**rule_array**

*Direction: Input*

*Type: Integer*

The rule_array contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The rule_array keywords are described in Table 41.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token algorithm</strong> (Required)</td>
<td></td>
</tr>
<tr>
<td>HMAC</td>
<td>Specifies to import an HMAC key token.</td>
</tr>
<tr>
<td>AES</td>
<td>Specifies to import an AES key token.</td>
</tr>
<tr>
<td><strong>Key part</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td>FIRST</td>
<td>This keyword specifies that an initial key part is being entered. This verb returns this key-part encrypted by the master key in the key token that you supplied.</td>
</tr>
<tr>
<td>ADD-PART</td>
<td>This keyword specifies that additional key-part information is provided.</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>This keyword specifies that the key-part bit shall be turned off in the control vector of the key rendering the key fully operational. Note that no key-part information is added to the key with this keyword.</td>
</tr>
<tr>
<td><strong>Split knowledge</strong> (Optional, required when keyword FIRST is used)</td>
<td></td>
</tr>
<tr>
<td>MIN3PART</td>
<td>Specifies that the key must be entered in at least three parts.</td>
</tr>
<tr>
<td>MIN2PART</td>
<td>Specifies that the key must be entered in at least two parts.</td>
</tr>
<tr>
<td>MIN1PART</td>
<td>Specifies that the key must be entered in at least one part.</td>
</tr>
</tbody>
</table>

**key_part_bit_length**

*Direction: Input*

*Type: Integer*

The length of the clear key in bits. This indicates the bit length of the key supplied in the key_part field. For FIRST and ADD-PART keywords, valid values are 80 - 2048 for HMAC keys, or 128, 192, or 256 for AES keys. The value must be 0 for the COMPLETE keyword.

**key_part**

*Direction: Input*

*Type: String*

This parameter is the clear key value to be applied. The key part must be left-aligned. This parameter is ignored if COMPLETE is specified.

**key_identifier_length**
Direction: Input/Output
Type: Integer

On input, the length of the buffer for the key_identifier parameter. For labels, the value is 64. The key_identifier must be left-aligned in the buffer. The buffer must be large enough to receive the updated token. The maximum value is 725. The output token will be longer when the first key part is imported.

On output, the actual length of the token returned to the caller. For labels, the value will be 64.

key_identifier

Direction: Input/Output
Type: String

The parameter containing an internal token or a 64-byte label of an existing key storage file record. If rule_array is FIRST, the key is a skeleton token. If rule_array is ADD-PART, this is an internal token or the label of a key storage file record of a partially combined key. Depending on the input format, the accumulated partial or complete key is returned as an internal token or as an updated record in a key storage file. The returned key_identifier will be encrypted under the current master key.

Restrictions
The restrictions for CSNBKPI2.

This verb was introduced with CCA 4.1.0.

Required commands
The required commands for CSNBKPI2.

This verb requires the following commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST and MIN3PART</td>
<td>X'0297'</td>
<td>Key Part Import2 - Load first key part, require 3 key parts</td>
</tr>
<tr>
<td>FIRST and MIN2PART</td>
<td>X'0298'</td>
<td>Key Part Import2 - Load first key part, require 2 key parts</td>
</tr>
<tr>
<td>FIRST and MIN1PART</td>
<td>X'0299'</td>
<td>Key Part Import2 - Load first key part, require 1 key parts</td>
</tr>
<tr>
<td>ADD-PART</td>
<td>X'029A'</td>
<td>Key Part Import2 - Add second of 3 or more key parts</td>
</tr>
<tr>
<td></td>
<td>X'029B'</td>
<td>Key Part Import2 - Add last required key part</td>
</tr>
<tr>
<td></td>
<td>X'029C'</td>
<td>Key Part Import2 - Add optional key part</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>X'029D'</td>
<td>Key Part Import2 - Complete key</td>
</tr>
</tbody>
</table>

Usage notes
The usage notes for CSNBKPI2.

On each call to Key Part Import2, also specify a rule-array keyword to define the service action: FIRST, ADD-PART, or COMPLETE.
Key Part Import2 (CSNBKPI2)

- With the **FIRST** keyword, the input key-token must be a skeleton token (no key material). Use of the **FIRST** keyword requires that the Load First Key Part2 access control point be enabled in the default role.
- With the **ADD-PART** keyword, the service XORs the clear key-part with the key value in the input key-token. Use of the **ADD-PART** keyword requires that an Add Key Part2 access control point be enabled in the default role. The key remains incomplete in the updated key token returned from the service.
- With the **COMPLETE** keyword, the KEY-PART bit is set off in the updated key token that is returned from the service. Use of the **COMPLETE** keyword requires that the Complete Key Part2 access control point be enabled in the default role. The `key_part_bit_length` parameter must be set to zero.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKPI2J.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKPI2J are shown here.

**Format**

```java
public native void CSNBKPI2J(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_part_bit_length,
    hikmNativeInteger key_part,
    hikmNativeInteger key_identifier_length,
    hikmNativeInteger key_identifier);
```
Key Test (CSNBKYT)

Use the Key Test verb to generate or verify the value of either a master key, an internal AES key or key-part, or an internal DES key or key-part.

A key to test can be in the clear or encrypted under the master key. Keywords in the rule_array parameter specify whether the verb generates or verifies a verification pattern.

This algorithm is supported for clear and encrypted single and double length keys. Single, double and triple length keys are also supported with the ENC-ZERO algorithm. Clear triple length keys are not supported. See “Cryptographic key-verification techniques” on page 697.

With the default method, the verb generates a verification pattern and it creates and cryptographically processes a random number. This verb returns the random number with the verification pattern.

For historical reasons, the verification information is passed in two 8-byte variables pointed to by the value_1 and value_2 parameters. The GENERATE option uses these variables for output, and the VERIFY option uses these variables as input. For VERIFY, the verb returns a warning of return code 4, reason code 1 if the information provided in these variables does not match the calculated values.

Table 43 describes the use of the value_1 and value_2 variables for each of the available verification-process rule keywords.

This document uses new names for two of the parameters. The former names were misleading because they no longer reflected the use of these parameters. The header file, csulincl.h, continues to use the former names. See Table 42.

Table 42. Key Test parameter changes

<table>
<thead>
<tr>
<th>Current name (used in this document)</th>
<th>Former name (used in header file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_1</td>
<td>random_number</td>
</tr>
<tr>
<td>value_2</td>
<td>verification_pattern</td>
</tr>
</tbody>
</table>

Table 43. Key Test GENERATE outputs and VERIFY inputs

<table>
<thead>
<tr>
<th>Verification-process rule</th>
<th>GENERATE outputs and VERIFY inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>value_1 variable</td>
</tr>
<tr>
<td>ENC-ZERO</td>
<td>Unused</td>
</tr>
<tr>
<td>MDC-4</td>
<td>Contains the 8-byte KVP taken from the high-order 8 bytes of the MDC-4 hash value.</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Contains the 8-byte KVP taken from the high-order 8 bytes of the SHA-1 hash value.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Unused</td>
</tr>
</tbody>
</table>
**Key Test (CSNBKYT)**

Table 43. Key Test GENERATE outputs and VERIFY inputs (continued)

<table>
<thead>
<tr>
<th>Verification-process rule</th>
<th>GENERATE outputs and VERIFY inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>value_1 variable</td>
</tr>
<tr>
<td>No keyword, and first and third parts of the master key have different values</td>
<td>Same as SHA-1</td>
</tr>
<tr>
<td></td>
<td>value_2 variable</td>
</tr>
<tr>
<td>No keyword, and first and third parts of the master key have the same value</td>
<td>Contains the 8-byte KVP taken from the result of the z/OS-based master-key verification method.</td>
</tr>
<tr>
<td></td>
<td>Unused</td>
</tr>
</tbody>
</table>

**Format**

The format of CSNBKYT.

```c
CSNBKYT(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier,
    value_1,
    value_2)
```

This document uses new names for two of the parameters. The former names were misleading because they no longer reflected the use of these parameters. The header file, csullincl.h, continues to use the former names. See Table 42 on page 195.

**Parameters**

The parameters for CSNBKYT.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 2, 3, 4, or 5.

**rule_array**

- **Direction:** Input
- **Type:** String

Two to five keywords provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The `rule_array` keywords are described in Table 44.

Table 44. Keywords for Key Test control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key rule</td>
<td>(One, required)</td>
</tr>
</tbody>
</table>
### Table 44. Keywords for Key Test control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY-CLR</td>
<td>Specifies the key supplied in key_identifier is a single-length clear key.</td>
</tr>
<tr>
<td>KEY-CLRD</td>
<td>Specifies the key supplied in key_identifier is a double-length clear key.</td>
</tr>
<tr>
<td>KEY-ENC</td>
<td>Specifies the key supplied in key_identifier is a single-length encrypted key.</td>
</tr>
<tr>
<td>KEY-ENCD</td>
<td>Specifies the key supplied in key_identifier is a double-length encrypted key.</td>
</tr>
<tr>
<td>KEY-KM</td>
<td>Specifies that the target is the master key register.</td>
</tr>
<tr>
<td>KEY-NKM</td>
<td>Specifies that the target is the new master-key register.</td>
</tr>
<tr>
<td>KEY-OKM</td>
<td>Specifies that the target is the old master-key register.</td>
</tr>
<tr>
<td>CLR-A128</td>
<td>Process a 128-bit AES clear-key or clear-key part.</td>
</tr>
<tr>
<td>CLR-A192</td>
<td>Process a 192-bit AES clear-key or clear-key part.</td>
</tr>
<tr>
<td>CLR-A256</td>
<td>Process a 256-bit AES clear-key or clear-key part.</td>
</tr>
<tr>
<td>TOKEN</td>
<td>Process an AES clear or encrypted key contained in an AES key-token.</td>
</tr>
<tr>
<td>AES-MK</td>
<td>Process one of the AES master-key registers.</td>
</tr>
<tr>
<td>APKA-MK</td>
<td>Process one of the APKA master-key registers. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>ASYM-MK</td>
<td>Specifies use of only the asymmetric master-key registers.</td>
</tr>
<tr>
<td>SYM-MK</td>
<td>Specifies use of only the symmetric master-key registers.</td>
</tr>
<tr>
<td>GENERATE</td>
<td>Generate a verification pattern for the key supplied in key_identifier.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Verify a verification pattern for the key supplied in key_identifier.</td>
</tr>
<tr>
<td>ADJUST</td>
<td>Adjust the parity of test key to odd before generating or verifying the verification pattern. The key_identifier field itself is not adjusted.</td>
</tr>
<tr>
<td>NOADJUST</td>
<td>Do not adjust the parity of test key to odd before generating or verifying the verification pattern. This is the default.</td>
</tr>
<tr>
<td>ENC-ZERO</td>
<td>Specifies use of the &quot;encrypted zeros&quot; method. Use only with KEY-CLR, KEY-CLRD, KEY-ENC, or KEY-ENCD keywords.</td>
</tr>
<tr>
<td>MDC-4</td>
<td>Specifies use of the MDC-4 master key verification method. Use only with the KEY-KM, KEY-NKM, or KEY-OKM keywords. You must specify one master-key selector keyword to use this keyword.</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Specifies use of the SHA-1 master-key-verification method. Use only with KEY-KM, KEY-NKM, or KEY-OKM keywords. You must specify one master-key selector keyword to use this keyword.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Specifies use of the SHA-256 master-key-verification method.</td>
</tr>
</tbody>
</table>

**Master-key selector** (One, optional). Use only with KEY-KM, KEY-NKM, or KEY-OKM keywords.

<table>
<thead>
<tr>
<th>Process rule (One, required)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATE</td>
<td>Generate a verification pattern for the key supplied in key_identifier.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Verify a verification pattern for the key supplied in key_identifier.</td>
</tr>
</tbody>
</table>

**Parity adjustment** (One, optional)

<table>
<thead>
<tr>
<th>ADJUST</th>
<th>Adjust the parity of test key to odd before generating or verifying the verification pattern. The key_identifier field itself is not adjusted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOADJUST</td>
<td>Do not adjust the parity of test key to odd before generating or verifying the verification pattern. This is the default.</td>
</tr>
</tbody>
</table>

**Verification process rule** (One, optional). See "Cryptographic key-verification techniques" on page 697.

<table>
<thead>
<tr>
<th>ENC-ZERO</th>
<th>Specifies use of the &quot;encrypted zeros&quot; method. Use only with KEY-CLR, KEY-CLRD, KEY-ENC, or KEY-ENCD keywords.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDC-4</td>
<td>Specifies use of the MDC-4 master key verification method. Use only with the KEY-KM, KEY-NKM, or KEY-OKM keywords. You must specify one master-key selector keyword to use this keyword.</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Specifies use of the SHA-1 master-key-verification method. Use only with KEY-KM, KEY-NKM, or KEY-OKM keywords. You must specify one master-key selector keyword to use this keyword.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Specifies use of the SHA-256 master-key-verification method.</td>
</tr>
</tbody>
</table>

No keyword, and first and third parts of the master key have different values.

No keyword, and first and third parts of the master key have the same value.

`key_identifier`
Key Test (CSNBKYT)

Direction: Input/Output  
Type: String

The key for which to generate or verify the verification pattern. The parameter is a 64-byte string of an internal token, key label, or a clear key value left-aligned.

Note: If you supply a key label for this parameter, it must be unique in the key storage file.

value_1  
Direction: Input/Output  
Type: String

A pointer to a string variable. See Table 43 on page 195 for how this variable is used. For process rule GENERATE this parameter is output only, and for process rule VERIFY it is input only. This variable must be specified, even if it is not used. With the ENC-ZERO method, this parameter is not used.

value_2  
Direction: Input/Output  
Type: String

A pointer to a string variable. See Table 43 on page 195 for how this variable is used. For process rule GENERATE this parameter is output only, and for process rule VERIFY it is input only. This variable must be specified, even if it is not used. With the ENC-ZERO method, the high-order four bytes contain the verification data. For more detail, see "Cryptographic key-verification techniques" on page 697.

Restrictions
The restrictions for CSNBKYT.

None

Required commands
The required commands for CSNBKYT.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNBKYT.

You can generate the verification pattern for a key when you generate the key. You can distribute the pattern with the key and it can be verified at the receiving node. In this way, users can ensure using the same key at the sending and receiving locations. You can generate and verify keys of any combination of key forms, that is, clear, operational or external.

The parity of the key is not tested.

For triple-length keys, use KEY-ENC or KEY-ENCD with ENC-ZERO. Clear triple-length keys are not supported.
In the Transaction Security System, **KEY-ENC** or **KEY-ENCD** both support enciphered single-length and double-length keys. They use the key-form bits in byte 5 of CV to determine the length of the key. To be consistent, in this implementation of CCA, both **KEY-ENC** and **KEY-ENCD** handle single- and double-length keys. Both products effectively ignore the keywords, which are supplied only for compatibility reasons.

This document uses new names for two of the parameters. The former names were misleading because they no longer reflected the use of these parameters. The header file, `csuincl.h`, continues to use the former names. See Table 42 on page 195.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named **CSNBKYTJ**.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for **CSNBKYTJ** are shown here.

**Format**

```java
public native void CSNBKYTJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_identifier,
    byte[] value_1,
    byte[] value_2);
```
Key Test2 (CSNBKYT2)

Use the Key Test2 verb to generate or verify a secure, cryptographic verification pattern for keys contained in a variable-length symmetric key-token.

A key to test can be in the clear or encrypted under the master key. In addition, the verb permits you to test the CCA master keys. Keywords in the rule_array parameter specify whether the verb generates or verifies a verification pattern. See “Cryptographic key-verification techniques” on page 697.

When the verb tests a verification pattern against a key, you must supply the verification pattern from a previous call to Key Test2. This verb returns the verification result in the return code and reason code.

Format

The format of CSNBKYT2.

```
CSNBKYT2(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   rule_array_count,
   rule_array,
   key_identifier_length,
   key_identifier,
   key_encrypting_key_identifier_length,
   key_encrypting_key_identifier,
   reserved_length,
   reserved,
   verification_pattern_length,
   verification_pattern)
```

Parameters

The parameters for CSNBKYT2.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

**rule_array_count**

Direction: Input
Type: Integer

The number of keywords you supplied in the rule_array parameter. This value must be 2, 3, 4, or 5.

**rule_array**

Direction: Input
Type: String

The rule_array contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The rule_array keywords are described in Table 45 on page 201.
Table 45. Keywords for Key Test2 control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token algorithm</strong> (Required)</td>
<td></td>
</tr>
<tr>
<td><strong>AES</strong></td>
<td>Specifies that the key token is an AES key token.</td>
</tr>
<tr>
<td><strong>DES</strong></td>
<td>Specifies that the key token is a DES token. CCA internal, CCA external, and TR-31 token types are supported. Clear keys are not supported for this rule.</td>
</tr>
<tr>
<td><strong>HMAC</strong></td>
<td>Specifies that the key token is an HMAC key token.</td>
</tr>
<tr>
<td><strong>Process rule</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td><strong>GENERATE</strong></td>
<td>Generate a verification pattern and an associated random number for the input key or key part for the specified key.</td>
</tr>
<tr>
<td><strong>VERIFY</strong></td>
<td>Verify that a verification pattern matches the specified key.</td>
</tr>
<tr>
<td><strong>Verification pattern calculation algorithm</strong> (One optional)</td>
<td></td>
</tr>
</tbody>
</table>
| **ENC-ZERO**        | Verification pattern for AES and DES keys calculated by encrypting a data block filled with X'00' bytes.  
This is the default and only method available for DES. Not valid with HMAC. This method is only available for AES if the Key Test2 - AES, ENC-ZERO (offset X'0021)' access control point is enabled. |
| **SHA-256**         | Verification pattern will be calculated for an AES token using the same method as the Key Test verb, with the SHA-256 rule.  
This rule can be used to verify that the same key value is present in a version X'04' DATA token and version X'05' AES CIPHER token or to verify that the same key value is present in a version X'05' AES IMPORTER/EXPORTER pair. |
| **SHA2VP1**         | Specifies to use the SHA-256 based verification pattern calculation algorithm. Valid only with HMAC. This is the default for HMAC. For more information, see "SHAVP1 algorithm" on page 700. |
| **Token type rule** (Required if TR-31 token passed and token algorithm DES is specified. Not valid otherwise.) |                                                                                                   |
| **TR-31**           | Specifies that key_identifier contains a TR-31 key block.                                          |
| **KEK identifier rules** (Optional - see defaults) |                                                                                                   |
| **IKEK-AES**        | The wrapping KEK for the key to test is an AES KEK. This is the default for AES and HMAC token algorithms, and is not allowed with DES. |
| **IKEK-DES**        | The wrapping KEK for the key to test is a DES KEK. This is the default for DES token algorithm, and is only allowed with DES token algorithm. |
| **IKEK-PKA**        | The wrapping KEK for the key to test is an RSA or (other key stored in PKA key storage.) This is not the default for any token algorithm, and must be specified if an RSA KEK is used. This rule is not allowed with DES token algorithm. |

**key_identifier_length**

**Direction:** Input  
**Type:** Integer

The length of the key_identifier in bytes. The maximum value is 9992.
key_identifier
Direction: Input
Type: String
A pointer to the key for which to generate or verify the verification pattern. The parameter is a variable length string of an internal token or the 64-byte label of a key in key storage. This token may be a DES internal or external token, AES internal version X’04’ token, internal or external variable-length symmetric token, or a TR-31 key block. Clear DES tokens are not supported. If an internal token was supplied and was encrypted under the old master key, the token will be returned encrypted under the current master key.

key_encrypting_key_identifier_length
Direction: Input
Type: Integer
The byte length of the key_encrypting_key_identifier parameter. When key_identifier is an internal token, the value must be zero.
If key_encrypting_key_identifier is a label for a record in key storage, the value must be 64. If the key_encrypting_key_identifier is an AES KEK, the value must be between the actual length of the token and 725. If the key_encrypting_key_identifier is a DES KEK, the value must be 64. If key_encrypting_key_identifier is an RSA KEK, the maximum length is 3500.

key_encrypting_key_identifier
Direction: Input/Output
Type: String
When key_encrypting_key_identifier_length is non-zero, the key_encrypting_key_identifier contains an internal key token containing the key-encrypting key, or a key label. If the key identifier supplied was an AES or DES token encrypted under the old master key, the token will be returned encrypted under the current master key.

reserved_length
Direction: Input
Type: Integer
The byte length of the reserved parameter. This value must be 0.

reserved
Direction: Input/Output
Type: String
This parameter is ignored.

verification_pattern_length
Direction: Input/Output
Type: Integer
The length in bytes of the verification_pattern parameter.
On input: for GENERATE the length must be at least 8 bytes; for VERIFY the length must be 8 bytes.
On output for GENERATE the length of the verification pattern returned.

verification_pattern
Key Test2 (CSNBKYT2)

Direction: Input/Output
Type: String

For **GENERATE**, the verification pattern generated for the key. For **VERIFY**, the supplied verification pattern to be verified.

**Restrictions**

The restrictions for CSNBKYT2.

The `key_identifier` parameter must not identify a key label when the input key is in a TR-31 key block.

**Required commands**

The required commands for CSNBKYT2.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

For verification rule keyword **ENC-ZERO** together with algorithm keyword **AES**, the Compute ENC-ZERO Verification Pattern for AES command (offset X'0021') must be enabled in the active rule. This command is not required for algorithm keyword **DES**.

**Usage notes**

The usage notes for CSNBKYT2.

You can generate the verification pattern for a key when you generate the key. You can distribute the pattern with the key and it can be verified at the receiving node. In this way, users can ensure using the same key at the sending and receiving locations. You can generate and verify keys of any combination of key forms, that is, clear, operational or external.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBKYT2J.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKYT2J are shown here.

**Format**

```java
public native void CSNBKYT2J(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_identifier_length,
    byte[] key_identifier,
    hikmNativeInteger key_encrypting_key_identifier_length,
    byte[] key_encrypting_key_identifier,
    hikmNativeInteger reserved_length,
    byte[] reserved,
    hikmNativeInteger verification_pattern_length,
    byte[] verification_pattern);
```
This verb is essentially the same as Key Test (CSNBKYT).

For further information, see "Key Test (CSNBKYT)" on page 195. The differences are:
- In addition to operating on internal keys and key parts, this verb also operates on external keys and key parts.
- This verb does not operate on clear keys, and does not accept rule_array keywords CLR-A128, CLR-A192, CLR-A256, KEY-CLR, and KEY-CLRD.

See also "Key Test (CSNBKYT)" on page 195 for operating only on internal keys.

Use this verb to verify the value of a key or key part in an external or internal key token. This verb supports two options:

**GENERATE**
To compute and return a verification pattern for a specified key.

**VERIFY**
To verify that a passed verification pattern is correct for the specified key.

The verification pattern and the verification process do not reveal any information about the value of the tested key, other than equivalency of two key values. Several verification algorithms are supported.

This verb supports testing of AES (Release 3.30 or later), DES, and PKA master keys, and enciphered keys or key parts. rule_array keywords are used to specify information about the target key that is not implicit from other verb parameters.

When testing the master keys, there are two sets of rule_array keywords to indicate what key to test:
1. The **SYM-MK**, **ASYM-MK**, and **AES-MK** (Release 3.30 or later) master-key selector keywords indicate whether to test the DES (symmetric) master key, the PKA (asymmetric) master key, or the AES master key.
2. The **KEY-KM**, **KEY-NKM**, and **KEY-OKM** key or key-part rule_array keywords choose among the current-master-key register, the new-master-key register, and the old-master-key register.

Not specifying a master-key selector keyword (**SYM-MK**, **ASYM-MK**, or **AES-MK**) means that the DES (symmetric) and PKA (asymmetric) master keys have the same value, and that you want to test that value.

Several key test algorithms are supported by the verb. See "Cryptographic key-verification techniques" on page 697. Some are implicitly selected based on the type of key you are testing, while others are optional and selected by specifying a verification process rule keyword. You can specify one of the following:

1. The **ENC-ZERO** keyword to encrypt a block of binary zeros with the specified key. This verb returns the leftmost 32 bits of the encryption result as the verification pattern. The encrypted block consists of 16 bytes of binary zeros for AES, and eight bytes for DES and Triple-DES keys. This method is valid only with the **TOKEN** keyword for AES, and **KEY-ENC** and **KEY-ENCD** keywords for DES.
2. The **MDC-4** keyword to compute a 16-byte verification pattern using the MDC-4 algorithm. This keyword is valid only when computing the verification pattern for a DES (symmetric) or PKA (asymmetric) master key.
Key Test Extended (CSNBKYTX)

3. The SHA-1 keyword to compute the verification pattern using the SHA-1 hashing method. This keyword is valid only when computing the verification pattern for the DES (symmetric) or PKA (asymmetric) master key.

4. The SHA-256 keyword to compute the verification pattern using the SHA-256 hashing method. This keyword is valid only when computing the verification pattern for an AES key.

Table 43 on page 195 describes the use of the random_number and verification_pattern fields for each of the available verification methods.

Note: For historical reasons, the verification information is passed in two 8-byte variables pointed to by the random_number and verification_pattern parameters. The GENERATE option returns information in these two variables, and the VERIFY option uses the information provided in these two variables. If the verb cannot verify the information provided, it returns a return code of 4 and a reason code of 1. For simplicity, these two variables can be two 8-byte elements of a 16-byte array, which is processed by your application program as a single quantity. Both parameters must be coded when calling the API.

DES and Triple-DES keys reserve the low-order bit of each byte for parity. If parity is used, the low-order bit is set so that the total number of '1' bits in the byte is odd. These parity adjustment keywords allow you to control how the Key Test Extended verb handles the parity bits:

NOADJUST
   Specifies not to alter the parity bit values in any way. This is the default.

ADJUST
   Specifies to modify the low-order bit of each byte as necessary for odd parity.

   This is done on the cleartext value of the key before the verification pattern is computed. The parity adjustment is performed only on a temporary copy of the key within the card, and does not affect the key value in the key_identifier parameter.

Format
The format of CSNBKYTX.

C

```c
CSNBKYTX(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   rule_array_count,
   rule_array,
   key_identifier,
   random_number,
   verification_pattern)
```

Parameters
The parameters for CSNBKYTX.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

rule_array_count
Key Test Extended (CSNBKYTX)

**Direction: Input**
Type: Integer
A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 2, 3, 4, or 5.

**rule_array**
Direction: Input
Type: Array
Between two and five keywords provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The `rule_array` keywords are described in Table 46.

Table 46. Keywords for Key Test Extended control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process rule</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td>GENERATE</td>
<td>Generate a verification pattern for the key supplied in <code>key_identifier</code>.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Verify a verification pattern for the key supplied in <code>key_identifier</code>.</td>
</tr>
<tr>
<td><strong>Key or key-part rule</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td>KEY-ENC</td>
<td>Specifies that the key supplied in <code>key_identifier</code> is a single-length encrypted key.</td>
</tr>
<tr>
<td>KEY-ENCD</td>
<td>Specifies that the key supplied in <code>key_identifier</code> is a double-length encrypted key.</td>
</tr>
<tr>
<td>KEY-KM</td>
<td>Specifies that the target is the master key register.</td>
</tr>
<tr>
<td>KEY-NKM</td>
<td>Specifies that the target is the new master-key register.</td>
</tr>
<tr>
<td>KEY-OKM</td>
<td>Specifies that the target is the old master-key register.</td>
</tr>
<tr>
<td>TOKEN</td>
<td>Process an AES clear or encrypted key contained in an AES key-token.</td>
</tr>
<tr>
<td><strong>Master-key selector</strong> (One, optional). Use only with KEY-KM, KEY-NKM, or KEY-OKM keywords. The default is to process the ASYM-MK and SYM-MK key registers, which must have the same key for the default to be valid.</td>
<td></td>
</tr>
<tr>
<td>AES-MK</td>
<td>Process one of the AES master-key registers.</td>
</tr>
<tr>
<td>APKA-MK</td>
<td>Process one of the APKA master-key registers. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>ASYM-MK</td>
<td>Specifies use of only the asymmetric master-key registers.</td>
</tr>
<tr>
<td>SYM-MK</td>
<td>Specifies use of only the symmetric master-key registers.</td>
</tr>
<tr>
<td><strong>Parity adjustment</strong> (One, optional) Not valid with the AES-MK Master-key selector keyword.</td>
<td></td>
</tr>
<tr>
<td>ADJUST</td>
<td>Adjust the parity of test key to odd before generating or verifying the verification pattern. The <code>key_identifier</code> field itself is not adjusted.</td>
</tr>
<tr>
<td>NOADJUST</td>
<td>Do not adjust the parity of test key to odd before generating or verifying the verification pattern. This is the default.</td>
</tr>
<tr>
<td><strong>Verification process rule</strong> (One, optional) For the AES master key, SHA-256 is the default. For the DES or PKA master keys, the default is SHA-1 if the first and third parts of the key are different, or the IBM z/OS method if the first and third parts of the key are the same.</td>
<td></td>
</tr>
<tr>
<td>ENC-ZERO</td>
<td>Specifies use of the &quot;encrypted zeros&quot; method. Use only with the KEY-CLR, KEY-CLRD, KEY-ENC, or KEY-ENCD keywords.</td>
</tr>
<tr>
<td>MDC-4</td>
<td>Specifies use of the MDC-4 master key verification method. Use only with the KEY-KM, KEY-NKM, or KEY-OKM keywords. You must specify one master-key selector keyword to use this keyword.</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Specifies use of the SHA-1 master-key-verification method. Use only with the KEY-KM, KEY-NKM, or KEY-OKM keywords. You must specify one master-key selector keyword to use this keyword.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Specifies use of the SHA-256 master-key-verification method.</td>
</tr>
</tbody>
</table>

**key_identifier**

206  Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Key Test Extended (CSNBKYTX)

**A pointer to a string variable containing an internal or external key-token, a key label that identifies an internal or external key-token record, or a clear key.**

The key token contains the key or the key part used to generate or verify the verification pattern.

**random_number**

**Direction: Input/Output**

**Type: String**

A pointer to a string variable containing a number the verb might use in the verification process. When you specify the GENERATE keyword, the verb returns the random number. When you specify the VERIFY keyword, you must supply the number. With the ENC-ZERO method, the random_number variable is not used but must be specified.

**verification_pattern**

**Direction: Input/Output**

**Type: String**

A pointer to a string variable containing the binary verification pattern. When you specify the GENERATE keyword, the verb returns the verification pattern. When you specify the VERIFY keyword, you must supply the verification pattern. With the ENC-ZERO method, the verification data occupies the high-order four bytes, while the low-order four bytes are unspecified (the data is passed between your application and the cryptographic engine but is otherwise unused). For more detail, see “Cryptographic key-verification techniques” on page 697.

**kek_key_identifier**

**Direction: Input**

**Type: String**

A pointer to a string variable containing an operational key-token or the key label of an operational key-token record containing an IMPORTER or EXPORTER key-encrypting key. If the key_identifier parameter does not identify an external key-token, the contents of the kek_key_identifier variable should contain a null DES key-token.

**Restrictions**

The restrictions for CSNBKYTX.

1. Releases earlier than Release 3.20 do not support the ADJUST and NOADJUST parity adjustment keywords.

2. AES keys and keywords are not supported in releases before Release 3.30.

**Required commands**

The required commands for CSNBKYTX.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBKYTX.
Key Test Extended (CSNBKYTX)

You can generate the verification pattern for a key when you generate the key. You can distribute the pattern with the key and it can be verified at the receiving node. In this way, users can ensure using the same key at the sending and receiving locations. You can generate and verify keys of any combination of key forms: clear, operational, or external.

The parity of the key is not tested.

For triple-length keys, use KEY-ENC or KEY-ENCD with ENC-ZERO. Clear triple-length keys are not supported.

In the Transaction Security System, KEY-ENC and KEY-ENCD both support enciphered single-length and double-length keys. They use the key-form bits in byte 5 of the control vector (CV) to determine the length of the key. To be consistent, in this implementation of CCA, both KEY-ENC and KEY-ENCD handle single- and double-length keys. Both products effectively ignore the keywords, which are supplied only for compatibility reasons.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKYTXJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKYTXJ are shown here.

Format

```java
public native void CSNBKYTXJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_identifier,
    byte[] random_number,
    byte[] verification_pattern,
    byte[] kek_key_identifier);
```
Key Token Build (CSNBKTB)

The Key Token Build verb assembles a fixed-length symmetric key-token in application storage from information you supply, either as an internal fixed-length AES or DES key-token, or as an external fixed-length DES token. CCA does not support fixed-length external AES key tokens.

This verb can include a control vector that you supply or can build a control vector based on the key type and the control vector related keywords in the rule_array. The Key Token Build verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Format

The format of CSNBKTB.

```plaintext
CSNBKTB(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_token,
    key_type,
    rule_array_count,
    rule_array,
    key_value,
    reserved_1,
    reserved_2,
    token_data,
    control_vector,
    reserved_4,
    reserved_5,
    reserved_6,
    masterkey_verification_pattern)
```

Note: Previous implementations used the reserved_1 parameter to point to a four-byte integer or string that represented the master key verification pattern. In current versions, CCA requires this parameter to point to a four-byte value equal to binary zero.

Parameters

The parameters for CSNBKTB.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

key_token

**Direction:** Input/Output

**Type:** String

The `key_token` parameter is a pointer to a string variable containing the assembled `key_token`.

**Note:** This variable cannot contain a key label.

key_type
Key Token Build (CSNBKTbw)

**Direction:** Input
**Type:** String

The `key_type` parameter is a pointer to a string variable containing a keyword that defines the key type. The keyword is eight bytes in length and must be left-aligned and padded on the right with space characters.

Valid AES key type keywords are:
- CLRAES
- DATA

Valid DES key type keywords are:
- CIPHER
- DATAC
- DECIPHER
- DATA
- CIPHER
- DATAM
- ENCIPHER
- DATA
- DATAMV
- IPINENC
- MAC
- DATA
- IMPORTER
- PINENC
- MACVER
- DATA
- KEYGENKY
- OKEYXLAT
- DATA
- SECMSG
- USE-CV

Specify the `USE-CV` keyword to indicate that the key type should be obtained from the `control_vector` variable.

**rule_array_count**

**Direction:** Input/Output
**Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1, 2, 3, 4, 5, or 6.

**rule_array**

**Direction:** Output
**Type:** String

One to four keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. For any key type, there are no more than four valid `rule_array` values. The `rule_array` keywords are described in Table 47.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token type</td>
<td>(One required) An external key token.</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>An external key token.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>An internal key token.</td>
</tr>
</tbody>
</table>

**Token algorithm (One, optional)**

| AES          | An AES key.                                      |
| DES          | A DES key.                                       |

**Key status (One, optional). Not valid for CLRDES.**

| KEY          | The key token to build will contain an encrypted key. The `key_value` parameter identifies the field that contains the key. |
| NO-KEY       | The key token to build will not contain a key. This is the default key status.                                    |

**CV source (One, optional). Not valid for CLRDES.**

| CV           | The verb is to obtain the control vector from the variable identified by the `control_vector` parameter. |

---

`Table 47. Keywords for Key Token Build control information`

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token type</td>
<td>(One required) An external key token.</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>An external key token.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>An internal key token.</td>
</tr>
</tbody>
</table>

**Token algorithm (One, optional)**

| AES          | An AES key.                                      |
| DES          | A DES key.                                       |

**Key status (One, optional). Not valid for CLRDES.**

| KEY          | The key token to build will contain an encrypted key. The `key_value` parameter identifies the field that contains the key. |
| NO-KEY       | The key token to build will not contain a key. This is the default key status.                                    |

**CV source (One, optional). Not valid for CLRDES.**

| CV           | The verb is to obtain the control vector from the variable identified by the `control_vector` parameter. |

---
Table 47. Keywords for Key Token Build control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO-CV</td>
<td>The control vector is to be supplied based on the key type and the control vector related keywords. This is the default.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>Use enhanced key wrapping method, which is compliant with the ANSI X9.24 standard. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Use original key wrapping method, which uses ECB wrapping for DES key tokens and CBC wrapping for AES key tokens. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>ENH-ONLY</td>
<td>Restrict rewrapping of the output_key_token. After the token has been wrapped with the enhanced method, it cannot be rewrapped using the original method. This keyword was introduced with CCA 4.1.0.</td>
</tr>
</tbody>
</table>

See Figure 3 on page 43 for the key usage keywords that can be specified for a given key type.

The difference between Key Token Parse (CSNBKTP) and Control Vector Generate (CSNBCVG) is that Key Token Parse returns the rule_array keywords that apply to a parsed token, such as EXTERNAL, INTERNAL, and so forth. These rule_array parameters are returned in addition to the key_type parameter.

Keywords ENH-ONLY, WRAP-ECB, and WRAP-ENH were introduced with CCA 4.1.0.

**key_value**

**Direction:** Output  
**Type:** String

This parameter is a pointer to a string variable containing the enciphered key or AES clear-key value which is placed into the key field of the key token when you use the KEY rule_array keyword. If the KEY keyword is not specified, this parameter is ignored.

The length of this variable depends on the type of key that is provided. The length is 16 bytes for DES keys. A single-length DES key must be left-aligned and padded on the right with eight bytes of X'00'. For a clear AES key, the length is 16 bytes for KEYLN16, 24 bytes for KEYLN24, and 32 bytes for KEYLN32. An enciphered AES key is 32 bytes.
Key Token Build (CSNBKTB)

Direction: Output  
Type: Integer

This parameter is a pointer to an integer variable or a 4-byte string variable.  
The value must be equal to an integer valued 0.

reserved_2  
Direction: Output  
Type: Integer

This parameters is a pointer to an integer variable. The value must be 0 or a null pointer.

token_data_1  
Direction: Input  
Type: String

This parameter is unused for DES keys and cleartext AES keys. In either of those cases it must be a null pointer or point to a string variable containing eight bytes of binary zeros. For encrypted AES keys, this parameter is a pointer to a one-byte string variable containing the LRC value for the key passed in the key_value parameter. For more information on LRC values, see IBM CCA Basic Services Reference and Guide for the IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors.

control_vector  
Direction: Output  
Type: String

A parameter is a pointer to a string variable. If you specify the CV keyword in the rule_array, the contents of this variable are copied to the control vector field of the fixed-length DES key token. If the CV keyword is not specified, this keyword is ignored.

reserved_4  
Direction: Output  
Type: String

This parameter is a pointer to a string variable. The value must be binary zeros or a null pointer.

reserved_5  
Direction: Output  
Type: Integer

This parameter is a pointer to an integer variable. The value must be 0 or a null pointer.

reserved_6  
Direction: Output  
Type: String

This parameter is a pointer to an 8-byte string variable. The value must eight space characters or a null pointer.

master_key_verification_pattern  
Direction: Output  
Type: String
Key Token Build (CSNBKTB)

This parameter is a pointer to a string variable containing the master-key verification pattern of the master key used to encipher the key in the internal key-token. The contents of the variable are copied into the MKVP field of the key token when keywords INTERNAL and KEY are specified, and key_type keyword CLRAES is not specified.

Restrictions
The restrictions for CSNBKTB.

None

Required commands
The required commands for CSNBKTB.

None

Usage notes
The usage notes for CSNBKTB.

Because 24-byte (TRIPLE) DES keys can only be generated as DATA keys, capability to create 24-byte DES tokens (with keywords TRIPLE or KEYLN24 has not been added to Key Token Build (CSNBKTB). Instead, call Key Generate (CSNBKGN) directly.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBKTBJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBKTBJ are shown here.

Format
public native void CSNBKTBJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_token,
    byte[] key_type,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_value,
    byte[] reserved_1,
    hikmNativeInteger reserved_2,
    byte[] reserved_3,
    byte[] control_vector,
    hikmNativeInteger reserved_4,
    byte[] reserved_5,
    byte[] reserved_6,
    byte[] masterkey_verify_parm);
Key Token Build2 (CSNBKTB2)

Use the Key Token Build2 verb to assemble an internal variable-length symmetric key-token in application storage from information that you supply.

This verb assembles the information as a skeleton keyed hash Message Authentication Code (HMAC) internal key token. This skeleton token can be supplied to the Key Generate2 verb, which then provides a completed key token with the attributes of the skeleton along with a randomly generated key. These attributes become cryptographically bound to the key when it is enciphered.

The Key Token Build2 verb cannot assemble a usable key-token that contains an enciphered key. It can assemble an internal HMAC key-token that has either a clear key, usable for a limited number of services, or no key, which is only usable for passing to the Key Generate2 verb in order to receive an enciphered key.

The Key Token Build2 verb is a host-only verb and it does not use the cryptographic coprocessor. This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Format

The format of CSNBKTB2.

```
CSNBKTB2( return_code,
          reason_code,
          exit_data_length,
          exit_data,
          rule_array_count,
          rule_array,
          clear_key_bit_length,
          clear_key_value,
          key_name_length,
          key_name,
          user_associated_data_length,
          user_associated_data,
          token_data_length,
          token_data,
          reserved_length,
          reserved,
          target_key_token_length,
          target_key_token)
```

Parameters

The parameters for CSNBKTB2.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`rule_array_count`

- **Direction:** Input
- **Type:** Integer

  The number of keywords you supplied in the `rule_array` parameter. The minimum value is 4.

`rule_array`
### Key Token Build2 (CSNBKTB2)

**Direction:** Input

**Type:** String

The `rule_array` contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The `rule_array` keywords are described in Table 48.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token type</strong></td>
<td>(Required)</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>Specifies to build an internal key token.</td>
</tr>
<tr>
<td><strong>Token algorithm</strong></td>
<td>(Required)</td>
</tr>
<tr>
<td>HMAC</td>
<td>Specifies to build an HMAC key token.</td>
</tr>
<tr>
<td><strong>Key status</strong></td>
<td>(One, optional)</td>
</tr>
<tr>
<td>NO-KEY</td>
<td>Specifies to build the key token without a key value. This creates a skeleton key token that can later be supplied to the Key Generate2 verb. This is the default.</td>
</tr>
<tr>
<td>KEY-CLR</td>
<td>Specifies to build the key token with a clear key value. This creates a key token that can be used with the Key Test2 verb to generate a verification pattern for the key value.</td>
</tr>
<tr>
<td><strong>Key type</strong></td>
<td>(Required)</td>
</tr>
<tr>
<td>MAC</td>
<td>Specifies that this key is for Message Authentication Code operations.</td>
</tr>
<tr>
<td><strong>Key management related keywords (applies to all key types)</strong></td>
<td></td>
</tr>
<tr>
<td>Symmetric-key export key-management control</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>NOEX-SYM</td>
<td>Prohibits the export of the key with a symmetric key.</td>
</tr>
<tr>
<td>XPRT-SYM</td>
<td>Permits the export of the key with a symmetric key. This is the default.</td>
</tr>
<tr>
<td><strong>Unauthenticated asymmetric-key export key-management control</strong></td>
<td>(One, optional)</td>
</tr>
<tr>
<td>NOEXUASY</td>
<td>Prohibits the export of the key with an unauthenticated asymmetric key.</td>
</tr>
<tr>
<td>XPRTUASY</td>
<td>Permits the export of the key with an unauthenticated asymmetric key. This is the default.</td>
</tr>
<tr>
<td><strong>Authenticated asymmetric-key export key-management control</strong></td>
<td>(One, optional)</td>
</tr>
<tr>
<td>NOEXAASY</td>
<td>Prohibits the export of the key with an authenticated asymmetric key.</td>
</tr>
<tr>
<td>XPRTAASY</td>
<td>Permits the export of the key with an authenticated asymmetric key. This is the default.</td>
</tr>
<tr>
<td><strong>Key-usage keywords (these are specific to the key type specified)</strong></td>
<td></td>
</tr>
<tr>
<td>MAC key usage</td>
<td></td>
</tr>
<tr>
<td>GENERATE</td>
<td>Specifies that this key can be used to generate a MAC. A key that can generate a MAC can also verify a MAC.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Specifies that this key cannot be used to generate a MAC. It can only be used to verify a MAC.</td>
</tr>
<tr>
<td><strong>Hash method key-usage control</strong></td>
<td>(any combination, optional)</td>
</tr>
<tr>
<td>Note: All keywords in the list below are defaults unless one or more keywords in the list are specified.</td>
<td></td>
</tr>
<tr>
<td>SHA-1</td>
<td>Specifies that the SHA-1 hash method is allowed for the key.</td>
</tr>
<tr>
<td>SHA-224</td>
<td>Specifies that the SHA-224 hash method is allowed for the key.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Specifies that the SHA-256 hash method is allowed for the key.</td>
</tr>
<tr>
<td>SHA-384</td>
<td>Specifies that the SHA-384 hash method is allowed for the key.</td>
</tr>
<tr>
<td>SHA-512</td>
<td>Specifies that the SHA-512 hash method is allowed for the key.</td>
</tr>
</tbody>
</table>

| clear_key_bit_length |                                      |

Chapter 6. Managing AES and DES cryptographic keys 215
Key Token Build2 (CSNBKTB2)

Direction: Input  
Type: Integer

The length of the clear key in bits. Specify 0 when no key value is supplied or a valid HMAC key bit length, between 80 and 2048.

clear_key_value

Direction: Input  
Type: String

This parameter is used when the KEY-CLR keyword is specified. This parameter is the clear key value to be put into the token being built.

key_name_length

Direction: Input  
Type: Integer

The length of the key_name parameter. Valid values are 0 and 64.

key_name

Direction: Input  
Type: String

A 64-byte key store label to be stored in the associated data structure of the token.

user_associated_data_length

Direction: Input  
Type: Integer

The length of the user-associated data. The valid values are 0 - 255 bytes.

user_associated_data

Direction: Input  
Type: String

User-associated data to be stored in the associated data structure.

token_data_length

Direction: Input  
Type: Integer

This parameter is reserved. This value must be 0.

token_data

Direction: Ignored  
Type: String

This parameter is ignored.

reserved_length

Direction: Input  
Type: Integer

This parameter is reserved. This value must be 0.

reserved

Direction: Ignored  
Type: String

This parameter is ignored.
target_key_token_length
Direction: Input/Output
Type: Integer

On input, the length of the target_key_token parameter supplied to receive the token. On output, the actual length of the token returned to the caller. Maximum length is 725 bytes.

target_key_token
Direction: Output
Type: String

The key token built by this verb.

Restrictions
The restrictions for CSNBKTB2.

This verb was introduced with CCA 4.1.0.

Required commands
The required commands for CSNBKTB2.

None

Usage notes
The usage notes for CSNBKTB2.

None.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBKTB2J.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBKTB2J are shown here.

Format
public native void CSNBKTB2J(
   hikmNativeInteger return_code,
   hikmNativeInteger reason_code,
   hikmNativeInteger exit_data_length,
   byte[] exit_data,
   hikmNativeInteger rule_array_count,
   byte[] rule_array,
   hikmNativeInteger clear_key_bit_length,
   byte[] clear_key_value,
   hikmNativeInteger key_name_length,
   byte[] key_name,
   hikmNativeInteger user_associated_data_length,
   byte[] user_associated_data,
   hikmNativeInteger token_data_length,
   byte[] token_data,
   hikmNativeInteger reserved_length,
   byte[] reserved,
   hikmNativeInteger target_key_token_length,
   byte[] target_key_token);
Key Token Change (CSNBKTC)

Use the Key Token Change verb to re-encipher a DES or AES key from encryption under the old master-key to encryption under the current master-key and to update the keys in internal DES or AES key-tokens.

Note:
1. An application system is responsible for keeping all of its keys in a usable form. When the master key is changed, the CEXC implementations can use an internal key that is enciphered by either the current or the old master-key. Before the master key is changed a second time, it is important to have a key re-enciphered under the current master-key for continued use of the key. Use the Key Token Change verb to re-encipher such a key.
2. Previous implementations of IBM CCA products had additional capabilities with this verb such as deleting key records and key tokens in key storage. Also, use of a wild card (*) was supported in those implementations.

Format
The format of CSNBKTC.

```
CSNBKTC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier)
```

Parameters
The parameters for CSNBKTC.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1, 2, 3, or 4.

`rule_array`

Direction: Input
Type: Array

The `rule_array` parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords are described in Table 49:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-encipherment method</td>
<td>(Required)</td>
</tr>
</tbody>
</table>
Table 49. Keywords for Key Token Change control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCMK</td>
<td>Re-enciphers a DES or AES key to the current master-key in an internal key-token in application storage or in key storage. If the supplied key is already enciphered under the current master-key the verb returns a positive response (return code 0, reason code 0). If the supplied key is enciphered under the old master-key, the key is updated to encipherment by the current master-key and the verb returns a positive response (return code 0, reason code 0). Other cases return some form of abnormal response.</td>
</tr>
<tr>
<td>RTNMK</td>
<td>Re-enciphers an internal DES or AES key to the new master-key. A key enciphered under the new master key is not usable. It is expected that the user will use this keyword (RTNMK) to take a preparatory step in re-enciphering an external key store that they manage themselves to a new master-key, before the set operation has occurred. Note also that the new master-key register must be full; it must have had the last key part loaded and therefore not be empty or partially full (partially full means that one or more key parts have been loaded but not the last key part). The SET operation makes the new master-key operational, moving it to the current master-key register, and the current master-key is displaced into the old master-key register. When this happens, all the keys that were re-enciphered to the new master-key are now usable, because the new master-key is not ‘new’ any more, it is ‘current’. Because the RTNMK keyword is added primarily for support of externally managed key storage (see “Key Storage on z/OS (RTNMK-focused)” on page 349, it is not valid to pass a key_identifier when the RTNMK keyword is used. Only a full internal key token (encrypted under the current master-key) can be passed for re-encipherment with the RTNMK keyword. When a key label is passed along with the RTNMK keyword, the error return code 8 with reason code 181 will be returned. For more information, see “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.</td>
</tr>
</tbody>
</table>
| REFORMAT| Rewrap the input_key_token with the key wrapping method specified. Only the input KEK_identifier will be used. The output KEK_identifier is ignored. This keyword was introduced with CCA 4.1.0.

**Algorithm (Optional)**

<table>
<thead>
<tr>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
</tr>
<tr>
<td>DES</td>
</tr>
</tbody>
</table>

**Key wrapping method (Optional)**

<table>
<thead>
<tr>
<th>Key wrapping method</th>
</tr>
</thead>
<tbody>
<tr>
<td>USECONFIG</td>
</tr>
<tr>
<td>WRAP-ENH</td>
</tr>
<tr>
<td>WRAP-ECB</td>
</tr>
</tbody>
</table>

**Translation control (Optional)**

<table>
<thead>
<tr>
<th>Translation control</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENH-ONLY</td>
</tr>
</tbody>
</table>

**key_identifier**

**Direction:** Input/Output  
**Type:** String

The key_identifier parameter is a pointer to a string variable containing the DES internal key-token or the key label of an internal key-token record in key storage.
Key Token Change (CSNBKTC)

**Restrictions**
The restrictions for CSNBKTC.

None

**Required commands**
The required commands for CSNBKTC.

If you specify the **RTCMK** keyword, the Key Token Change verb requires the DES Key Token Change command (offset X'0090') to be enabled in the active role.

If you specify the **REFORMAT** keyword, the Key Token Change verb requires the CKDS Conversion2 - Allow use of REFORMAT command (offset X'014C') to be enabled in the active role.

If you specify the **WRAP-ECB** or **WRAP-ENH** key wrapping method, and the default key-wrapping method setting does not match this keyword, the Allow Configuration Override with Keyword in KTC command (offset X'0146') must be enabled in the active role.

**Usage notes**
The usage notes for CSNBKTC.

None

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNBKTCJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBKTCJ are shown here.

**Format**

```java
public native void CSNBKTCJ(  
    hikmNativeInteger return_code,  
    hikmNativeInteger reason_code,  
    hikmNativeInteger exit_data_length,  
    byte[] exit_data,  
    hikmNativeInteger rule_array_count,  
    byte[] rule_array,  
    byte[] key_label);
```
Key Token Change2 (CSNBKTC2)

Key Token Change2 (CSNBKTC2)

Use the Key Token Change2 verb to re-encipher a variable-length HMAC key from encryption under the old master-key to encryption under the current master-key and to update the keys in internal HMAC key-tokens.

Note:
1. An application system is responsible for keeping all of its keys in a usable form. When the master key is changed, the CEX*C implementations can use an internal key that is enciphered by either the current or the old master-key. Before the master key is changed a second time, it is important to have a key re-enciphered under the current master-key for continued use of the key. Use the Key Token Change2 verb to re-encipher such a keys.
2. Previous implementations of IBM CCA products had additional capabilities with this verb such as deleting key records and key tokens in key storage. Also, use of a wild card (*) was supported in those implementations.

Format

The format of CSNBKTC2.

```
CSNBKTC2(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier_length,
    key_identifier)
```

Parameters

The parameters for CSNBKTC2.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 2.

rule_array

Direction: Input
Type: Array

The rule_array parameter is a pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are described in Table 50.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>(One, required)</td>
</tr>
</tbody>
</table>

Table 50. Keywords for Key Token Change2 control information
Key Token Change2 (CSNBKTC2)

Table 50. Keywords for Key Token Change2 control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMAC</td>
<td>Specifies that the key token is for an HMAC key.</td>
</tr>
<tr>
<td>AES</td>
<td>Specifies that the key token is for an AES key in a variable-length symmetric key token.</td>
</tr>
</tbody>
</table>

Re-encipherment method (Required)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCMK</td>
<td>Re-enciphers a variable-length HMAC key to the current master-key in an internal key-token in application storage or in key storage. If the supplied key is already enciphered under the current master-key the verb returns a positive response (return code 0, reason code 0). If the supplied key is enciphered under the old master-key, the key is updated to encipherment by the current master-key and the verb returns a positive response (return code 0, reason code 0). Other cases return some form of abnormal response.</td>
</tr>
</tbody>
</table>
| RTNMK   | Re-enciphers an internal variable-length HMAC key to the new master-key. A key enciphered under the new master key is not usable. It is expected that the user will use this keyword (RTNMK) to take a preparatory step in re-enciphering an external key store that they manage themselves to a new master-key, before the set operation has occurred. Note also that the new master-key register must be full; it must have had the last key part loaded and therefore not be empty or partially full (partially full means that one or more key parts have been loaded but not the last key part).

The SET operation makes the new master-key operational, moving it to the current master-key register, and the current master-key is displaced into the old master-key register. When this happens, all the keys that were re-enciphered to the new master-key are now usable, because the new master-key is not new anymore, it is current.

Because the RTNMK keyword is added primarily for support of externally managed key storage (see "Key Storage on z/OS (RTNMK-focused)" on page 349), it is not valid to pass a key_identifier when the RTNMK keyword is used. Only a full internal key token (encrypted under the current master-key) can be passed for re-encipherment with the RTNMK keyword. When a key label is passed along with the RTNMK keyword, the error return code 8 with reason code 181 will be returned.

For more information, see "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

key_identifier_length

Direction: Input/Output
Type: Integer

The key_identifier_length parameter is a pointer to a string variable containing the length in bytes of the key_identifier parameter. On input, this variable contains the number of bytes for the key_identifier buffer, and must be large enough to hold the key token or key label. On output, this variable contains the number of bytes of data returned in the key_identifier variable.

key_identifier

Direction: Input/Output
Type: String

A pointer to a string variable containing an internal variable-length symmetric key-token, or a key label of such a key in AES key-storage. The enciphered data within the token is securely reenciphered under the current master-key.

Restrictions

The restrictions for CSNBKTC2.

This verb was introduced with CCA 4.1.0.
**Required commands**

The required commands for CSNKTC2.

If you specify the **RTNMK** keyword, this verb requires the Symmetric Key Token Change2 command (offset X'00F0') to be enabled in the active role.

If you specify the **RTCMK** keyword, this verb requires the Symmetric Key Token Change2 - RTCMK command (offset X'00F1') to be enabled in the active role.

**Usage notes**

The usage notes for CSNKTC2.

None

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNKTC2J.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNKTC2J are shown here.

**Format**

```java
public native void CSNKTC2J(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_identifier_length,
    byte[] key_identifier);
```
Key Token Parse (CSNBKTP)

The Key Token Parse verb disassembles a key token into separate pieces of information.

This verb can disassemble an external key token or an internal key token in application storage.

Use the key_token parameter to specify the key token to disassemble.

This verb returns some of the key token information in a set of variables identified by individual parameters and the remaining key token information as keywords in the rule_array.

Control vector information is returned in keywords found in the rule_array when the verb can fully parse the control vector. Otherwise, the verb returns return code 4, reason code 2039.

The Key Token Parse verb performs no cryptographic services.

Format

The format of CSNBKTP.

```
CSNBKTP(
    return_code,
    reason_code,
    exit_data_length,
    edit_data,
    key_token,
    key_type,
    rule_array_count,
    rule_array,
    key_value,
    MKVP,
    reserved_2,
    reserved_3,
    control_vector,
    reserved_4,
    reserved_5,
    reserved_6,
    master_key_verification_pattern)
```

Parameters

The parameters for CSNBKTP.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

key_token

Direction: Input
Type: String

The key_token parameter is a pointer to a string variable in application storage containing an external or internal key-token to be disassembled.

Note: You cannot use a key label for a key-token record in key storage. The key token must be in application storage.
Key Token Parse (CSNBKTP)

**key_type**

**Direction:** Output  
**Type:** String

The `key_type` parameter is a pointer to a string variable containing a keyword defining the key type. The keyword is eight bytes in length and must be left-aligned and padded on the right with space characters. Valid `key_type` keywords are shown here:

```
CIPHER       DATAC       1KEYXLAT      OPINENC
CVARDEC      DATAM       IMPORTER      PINGEN
CVARENC      DATAMV      IPINENC       PINVER
CVARPINE     DECIPHER    KEYGENKY     SECMSG
CVARKCVL     DKEYGENKY   MAC
CVARXCVR     ENCIPHER    MACVER
DATA         EXPORTER    OKEYXLAT
```

**rule_array_count**

**Direction:** Input/Output  
**Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be a minimum of 3.

On input, specify the maximum number of usable array elements that are allocated. On output, the verb sets the value to the number of keywords returned to the application.

**rule_array**

**Direction:** Output  
**Type:** Array

The `rule_array` parameter is a pointer to a string variable containing an array of keywords that expresses the contents of the key token. The keywords are eight bytes in length and are left-aligned and padded on the right with space characters. The `rule_array` keywords are described in Table 51.

---

### Table 51. Keywords for Key Token Parse control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token type</strong> (One returned)</td>
<td></td>
</tr>
<tr>
<td>INTERNAL</td>
<td>Specifies an internal key-token.</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>Specifies an external key-token.</td>
</tr>
<tr>
<td><strong>Key status</strong> (One returned)</td>
<td></td>
</tr>
<tr>
<td>KEY</td>
<td>Indicates the key token contains a key. The <code>key_value</code> parameter contains the key.</td>
</tr>
<tr>
<td>NO-KEY</td>
<td>Indicates the key token does not contain a key.</td>
</tr>
<tr>
<td><strong>Key-wrapping method</strong> (One returned)</td>
<td></td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>The wrapping method for this key is legacy. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>The wrapping method for this key is enhanced. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td><strong>Control-vector (CV) status</strong> (One returned)</td>
<td></td>
</tr>
<tr>
<td>CV</td>
<td>The key token specifies that a control vector is present. The verb sets the control vector variable with the value of the control vector found in the key token.</td>
</tr>
<tr>
<td>NO-CV</td>
<td>The key token does not specify the presence of a control vector. The verb sets the control vector variable with the value of the control vector variable found in the key token.</td>
</tr>
</tbody>
</table>
Key Token Parse (CSNBKTP)

The difference between Key Token Parse (CSNBKTP) and Control Vector
Generate (CSNBCVG) is that Key Token Parse returns the rule_array keywords
that apply to a parsed token, such as EXTERNAL, INTERNAL, and so forth.
These rule_array parameters are returned in addition to key_type parameter.

AMEX-CSC DKYL0 EPINGEN KEYN16 UKPT
ANSIX9.9 DKYL1 EPINGENA LMTD-KEK VISA-PVV
ANY DKYL2 EPINVER MIXED WRAP-ECB
ANY-MAC DKYL3 EXEX NO-SPEC WRAP-ENH
CLR8-ENC DKYL4 EXPORT NO-XPORT Xlate
CPINENC DKYL5 GBP-PIN NOOFFSET XPORT-OK
CPINGEN DKYL6 GBP-PINO NOT-KEK
CPINGENA DKYL7 IBM-PIN OPEX
CVVKEY-A DMAC IBM-PINO OPIM
CVVKEY-B DMKEY IMEX PIN
DALL DMPIN IMIM REFORMAT
DATA DMV IMPORT SINGLE
DDATA DOUBLE INBK-PIN SMKEY
DEXP DPVR KEY-PART SMPIN
DIMP ENH-ONLY KEYNB TRANSLAT

key_value

Direction: Output
Type: String
The key_value parameter is a pointer to a string variable. If the verb returns the
KEY keyword in the rule_array, the key_value parameter contains the 16-byte
enciphered key.

MKVP

Direction: Output
Type: Integer
The MKVP parameter is a pointer to an integer variable. The verb writes zero
into the variable except when parsing a version X'03' internal key-token.

reserved_2/5

Direction: Output
Type: Integer
The reserved_2 and reserved_5 parameters are either null pointers or pointers to
integer variables. If the parameter is not a null pointer, the verb writes zero
into the reserved variable.

reserved_3/4

Direction: Output
Type: String
The reserved_3 and reserved_4 parameters are either null pointers or pointers to
string variables. If the parameter is not a null pointer, the verb writes eight
bytes of X'00' into the reserved variable.

reserved_6

Direction: Output
Type: String
The reserved_6 parameter is either a null pointer or a pointer to a string
variable. If the parameter is not a null pointer, the verb writes eight space
characters into the reserved variable.
**Key Token Parse (CSNBKTP)**

**control_vector**
- **Direction:** Output
- **Type:** String

The `control_vector` parameter is a pointer to a string variable in application storage. If the verb returns the **NO-CV** keyword in the `rule_array`, the key token did not contain a control-vector value and the control vector variable is filled with 16 space characters.

**master_key_verification_pattern**
- **Direction:** Output
- **Type:** String

The `master_key_verification_pattern` parameter is a pointer to a string variable in application storage. For version 0 key-tokens that contain a key, the 8-byte master key version number will be copied to the variable. Otherwise the variable is filled with eight space characters.

**Restrictions**
The restrictions for CSNBKTP.

None

**Required commands**
The required commands for CSNBKTP.

None

**Usage notes**
The usage notes for CSNBKTP.

Be aware that Key Token Parse (CSNBKTP) will fail (return code 8, reason code 49) when given a DES INTERNAL key token that is version X'01'. These tokens are DOUBLE and TRIPLE length DES INTERNAL DATA key tokens.

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNBKTPJ.

See ["Building Java applications to use with the CCA JNI" on page 26](#).

The parameters for CSNBKTPJ are shown here.

**Format**
```
public native void CSNBKTPJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_token,
    byte[] key_type,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_value,
    hikmNativeInteger master_key_verification_pattern_v3,
    hikmNativeInteger reserved_field_2,
```
Key Token Parse (CSNBKTP)

```c
byte[] reserved_field_3,
byte[] control_vector,
byte[] reserved_field_4,
hkmNativeInteger reserved_field_5,
byte[] reserved_field_6,
byte[] master_key_verification_pattern_v0);
```
Use the Key Token Parse2 verb to disassemble a variable-length symmetric key-token into separate pieces of information.

The verb can disassemble an external or internal variable-length symmetric key-token in application storage. To parse a fixed-length symmetric key-token, see “Key Token Parse (CSNBKTP)” on page 224.

The *key_token* input parameter specifies the external or internal key token to disassemble. The verb returns some of the key-token information in a set of variables identified by individual parameters, and returns the remaining information as keywords in the rule array.

The key-usage field and key-management field information is returned in keywords found in the rule array when the verb can fully parse the fields. See the *rule_array* parameter on page 231 for a table of supported keywords. If the token cannot be parsed successfully, the verb returns a warning using reason code 2039 (X'7F7'). If a warning or error occurs during processing, the verb updates all of the count and length variables with a value of zero.

The Key Token Parse2 verb performs no cryptographic services.

To use this verb, specify the following:

- An external or internal variable-length symmetric key-token (version X'05') to be parsed
  This parameter does not accept a key label. The key token must be provided from application storage. If a key token located in key storage needs to be parsed, use the AES Key Record Read verb to retrieve it into application storage before calling this verb.
  See “HMAC key token” on page 640 for the format of this key token. A review of this format information will greatly assist in understanding the output variables of this verb.

- A rule-array-count value large enough for the verb to return keywords about the input key-token in the rule-array buffer
  To determine the exact count required, and also the required lengths of the other string variables, specify a value of zero. This causes the verb to return all count and length values without updating any string variables.

- Adequate buffer sizes for all of the output variables using the length parameters
Key Token Parse2 (CSNBKTP2)

Format
The format of CSNBKTP:

```
CSNBKTP2(
    return_code,  
    reason_code,  
    exit_data_length,  
    exit_data,  
    key_token_length,  
    key_token,  
    key_type,  
    rule_array_count,  
    rule_array,  
    key_material_state,  
    payload_bit_length,  
    payload,  
    key_verification_pattern_type,  
    key_verification_pattern_length,  
    key_verification_pattern,  
    key_wrapping_method,  
    key_hash_algorithm,  
    key_name_length,  
    key_name,  
    TLV_data_length,  
    TLV_data,  
    user_associated_data_length,  
    user_associated_data,  
    reserved_length,  
    reserved)
```

Parameters
The parameters for CSNBKTP:

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**key_token_length**

**Direction:** Input  
**Type:** Integer  
A pointer to an integer variable containing the number of bytes of data in the `key_token` variable.

**key_token**

**Direction:** Input  
**Type:** String  
A pointer to a string variable containing an external or internal variable-length symmetric key-token to be disassembled. This parameter must not point to a key label.

**key_type**

**Direction:** Output  
**Type:** String  
A pointer to a string variable containing a keyword for the key type of the input key. The keyword is 8 bytes in length and is left-aligned and padded on the right with space characters. Valid `key_type` keywords are shown here:

CIPHERIMPORTER  
EXPORTERMAC
Key Token Parse2 (CSNBKTP2)

rule_array_count

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of 8-byte elements in the rule_array variable. The minimum returned value is 3, and the maximum returned value is approximately 50. To determine the exact count required, and also the required lengths of the other string variables, specify a value of zero. This causes the verb to return all count and length values without updating any string variables.

On output, the variable is updated with the actual count of the rule-array keywords. An error is returned if a key token cannot be parsed or any of the output buffers are too small.

rule_array

Direction: Output
Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length, and are left-aligned and padded on the right with space characters. The returned rule array keywords express the contents of the token.

Table 52. Keywords for Key Token Parse2 control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header section</td>
<td></td>
</tr>
<tr>
<td>Token identifier (one returned)</td>
<td></td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>External key</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>Internal key</td>
</tr>
<tr>
<td>Wrapping information section</td>
<td></td>
</tr>
<tr>
<td>Key status (one returned). See key_material_state variable for additional details.</td>
<td></td>
</tr>
<tr>
<td>KEY</td>
<td>Key token contains a partial or complete key. The payload variable contains the clear or encrypted key.</td>
</tr>
<tr>
<td>NO-KEY</td>
<td>Key token does not contain a key. The payload variable is empty.</td>
</tr>
<tr>
<td>Key verification pattern (KVP) type Note: Not a keyword. Value returned in key_verification_pattern_type variable.</td>
<td></td>
</tr>
<tr>
<td>KVP Note: Not a keyword. Value returned in key_verification_pattern variable.</td>
<td></td>
</tr>
<tr>
<td>Encrypted section key-wrapping method Note: Not a keyword. Value returned in key_wrapping_method variable.</td>
<td></td>
</tr>
<tr>
<td>Hash algorithm used for wrapping Note: Not a keyword. Value returned in key_hash_algorithm variable.</td>
<td></td>
</tr>
<tr>
<td>Associated data section</td>
<td></td>
</tr>
<tr>
<td>Type of algorithm for which key can be used (one returned)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>AES</td>
</tr>
<tr>
<td>HMAC</td>
<td>HMAC</td>
</tr>
<tr>
<td>Key type</td>
<td></td>
</tr>
<tr>
<td>Note: Keyword is returned in key_type variable</td>
<td></td>
</tr>
<tr>
<td>Key-usage field 1, high-order byte</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 6. Managing AES and DES cryptographic keys 231
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt key-usage control (one or two returned, CIPHER key type only)</td>
<td></td>
</tr>
<tr>
<td>ENCRYPT</td>
<td>Key can be used for encryption. Used by Symmetric Algorithm Encipher.</td>
</tr>
<tr>
<td>DECRYPT</td>
<td>Key can be used for decryption. Used by Symmetric Algorithm Decipher.</td>
</tr>
<tr>
<td>Generate key-usage control (one returned, MAC key type only)</td>
<td></td>
</tr>
<tr>
<td>GENERATE</td>
<td>Key can be used for generate and key can be used for verify. Used by HMAC Generate.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Key cannot be used generate and key can be used for verify. Used by HMAC Verify.</td>
</tr>
<tr>
<td>EXPORTER key-usage control (one or more returned, EXPORTER key type only)</td>
<td></td>
</tr>
<tr>
<td>EXPORT</td>
<td>Key can be used for EXPORT. Used by Symmetric Key Export.</td>
</tr>
<tr>
<td>TRANSLAT</td>
<td>Key can be used for TRANSLAT. Used by Key Translate2.</td>
</tr>
<tr>
<td>GEN-OPEX</td>
<td>Key can be used for GENERATE-OPEX. Used by Key Generate2.</td>
</tr>
<tr>
<td>GEN-IMEX</td>
<td>Key can be used for GENERATE-IMEX. Used by Key Generate2.</td>
</tr>
<tr>
<td>GEN-EXEX</td>
<td>Key can be used for GENERATE-EXEX. Used by Key Generate2.</td>
</tr>
<tr>
<td>GEN-PUB</td>
<td>Key can be used for GENERATE-PUB. Used by Key Generate2.</td>
</tr>
<tr>
<td>IMPORTER key-usage control (one or more returned, IMPORTER key type only)</td>
<td></td>
</tr>
<tr>
<td>IMPORT</td>
<td>Key can be used for IMPORT. Used by Symmetric Key Import.</td>
</tr>
<tr>
<td>TRANSLAT</td>
<td>Key can be used for TRANSLAT. Used by Used by Key Translate2.</td>
</tr>
<tr>
<td>GEN-OPIM</td>
<td>Key can be used for GENERATE-OPIM. Used by Key Generate2.</td>
</tr>
<tr>
<td>GEN-IMEX</td>
<td>Key can be used for GENERATE-IMEX. Used by Key Generate2.</td>
</tr>
<tr>
<td>GEN-IMIM</td>
<td>Key can be used for GENERATE-IMIM. Used by Key Generate2.</td>
</tr>
<tr>
<td>GEN-PUB</td>
<td>Key can be used for GENERATE-PUB. Used by Key Generate2.</td>
</tr>
</tbody>
</table>

**Key-usage field 1, low-order byte**

<table>
<thead>
<tr>
<th>User-defined extension control (any number returned, all key types)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UDX-ONLY</td>
<td>Indicates that this key can only be used in UDXs.</td>
</tr>
<tr>
<td>UDX-001</td>
<td>Indicates that the rightmost user-defined UDX bit is set on.</td>
</tr>
<tr>
<td>UDX-010</td>
<td>Indicates that the middle user-defined UDX bit is set on.</td>
</tr>
<tr>
<td>UDX-100</td>
<td>Indicates that the leftmost user-defined UDX bit is set on.</td>
</tr>
</tbody>
</table>

**Key-usage field 2, high-order byte**

<table>
<thead>
<tr>
<th>Key-usage mode (one returned, CIPHER key type only)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CBC</td>
<td>Indicates that this key can be used for Cipher Block Chaining.</td>
</tr>
<tr>
<td>ECB</td>
<td>Indicates that this key can be used for Electronic Code Book.</td>
</tr>
<tr>
<td>CFB</td>
<td>Indicates that this key can be used for Cipher Feedback.</td>
</tr>
<tr>
<td>OFB</td>
<td>Indicates that this key can be used for Output Feedback.</td>
</tr>
<tr>
<td>GCM</td>
<td>Indicates that this key can be used for Galois/Counter Mode.</td>
</tr>
<tr>
<td>XTS</td>
<td>Indicates that this key can be used for Xor-Encrypt-Xor-based Tweaked Stealing.</td>
</tr>
</tbody>
</table>

**Hash method (one or more returned, MAC key type only)**

| SHA-1 | SHA-1 hash method is allowed for use by the key. |
| SHA-224 | SHA-224 hash method is allowed for use by the key. |
| SHA-256 | SHA-256 hash method is allowed for use by the key. |
### Table 52. Keywords for Key Token Parse2 control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-384</td>
<td>SHA-384 hash method is allowed for use by the key.</td>
</tr>
<tr>
<td>SHA-512</td>
<td>SHA-512 hash method is allowed for use by the key.</td>
</tr>
</tbody>
</table>

**Key-usage field 2, low-order byte**

*Key-encrypting key control (zero or one returned, Exporter or Importer key types only)*

| WR-TR31 | Indicates that this key-encrypting key can wrap or unwrap a TR-31 key block. |

**Key-usage field 3, high-order byte**

*Key-usage wrap algorithm (one or more returned, Exporter or Importer key types only)*

| WR-DES   | Key can wrap DES keys.                                                 |
| WR-AES   | Key can wrap AES keys.                                                 |
| WR-HMAC  | Key can wrap HMAC keys.                                                |
| WR-RSA   | Key can to wrap RSA keys.                                              |
| WR-ECC   | Key can wrap ECC keys.                                                 |

**Key-usage field 3, low-order byte**

*Reserved byte*

**Key-usage field 4, high-order byte**

*Key-usage wrap class (one or more returned, Exporter or Importer key types only)*

| WR-DATA  | Key can wrap DATA class keys.                                         |
| WR-KEK   | Key can wrap KEK class keys.                                          |
| WR-PIN   | Key can wrap PIN class keys.                                          |
| WRDERIVE | Key can wrap DERIVATION class keys.                                   |
| WR-CARD  | Key can wrap CARD class keys.                                         |

**Key-usage field 4, low-order byte**

*Reserved byte*

**Key-management field 1, high order byte**

*Symmetric-key export control (one returned, all key types)*

| NOEX-SYM | Prohibit export using symmetric key.                                  |
| XPRT-SYM | Allow export using symmetric key.                                    |

*Unauthenticated asymmetric-key export control (one returned, all key types)*

| NOEXUASY | Prohibit export using an unauthenticated asymmetric key.            |
| XPRTUASY | Allow export using unauthenticated asymmetric key.                  |

*Authenticated asymmetric-key export control (one returned, all key types)*

| NOEXAASY | Prohibit export using authenticated asymmetric key.                 |
| XPRTAASY | Allow export using authenticated asymmetric key.                   |

**Key-management field 1, low-order byte**

*DES-key export control (one returned, AES algorithm only)*

| NOEX-DES | Prohibit export using DES key.                                       |
| XPRT-DES | Allow export using DES key.                                         |

*AES-key export control (one returned, AES algorithm only)*

| NOEX-AES | Prohibit export using AES key.                                      |
Table 52. Keywords for Key Token Parse2 control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPRT-AES</td>
<td>Allow export using AES key.</td>
</tr>
<tr>
<td>RSA-key export control (one returned, AES algorithm only)</td>
<td></td>
</tr>
<tr>
<td>NOEX-RSA</td>
<td>Prohibit export using RSA key.</td>
</tr>
<tr>
<td>XPRT-RSA</td>
<td>Allow export using RSA key.</td>
</tr>
</tbody>
</table>

**Key-management field 2, high order byte**

<table>
<thead>
<tr>
<th>Key completeness (one returned, all key types)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN3PART</td>
<td>Key if present is incomplete. Key requires at least 2 more parts.</td>
</tr>
<tr>
<td>MIN2PART</td>
<td>Key if present is incomplete. Key requires at least 1 more part.</td>
</tr>
<tr>
<td>MIN1PART</td>
<td>Key if present is incomplete. Key can be completed or have more parts added.</td>
</tr>
<tr>
<td>KEYCMPLT</td>
<td>Key if present is complete. No more parts can be added.</td>
</tr>
</tbody>
</table>

**Key-management field 2, low-order byte**

<table>
<thead>
<tr>
<th>Security history (one returned, all key types)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UNTRUSTD</td>
<td>Key was encrypted with an untrusted KEK.</td>
</tr>
<tr>
<td>WOTUATTR</td>
<td>Key was in a format without type/usage attributes.</td>
</tr>
<tr>
<td>WREAKKEY</td>
<td>Key was encrypted with key weaker than itself.</td>
</tr>
<tr>
<td>NOTCCAFM</td>
<td>Key was in a non-CCA format.</td>
</tr>
<tr>
<td>WECBMODE</td>
<td>Key was encrypted in ECB mode.</td>
</tr>
</tbody>
</table>

**Key-management field 3, high order byte**

<table>
<thead>
<tr>
<th>Pedigree original rules (one returned, all key types)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>POUNKNWN</td>
<td>Unknown.</td>
</tr>
<tr>
<td>POOTHER</td>
<td>Other. Method other than those defined here, probably used in UDX.</td>
</tr>
<tr>
<td>PORANDOM</td>
<td>Randomly generated.</td>
</tr>
<tr>
<td>POKEYAGR</td>
<td>Established by key agreement such as Diffie-Hellman.</td>
</tr>
<tr>
<td>POCLRC</td>
<td>Created from cleartext key components.</td>
</tr>
<tr>
<td>POCLRKV</td>
<td>Entered as a cleartext key value.</td>
</tr>
<tr>
<td>PODERVD</td>
<td>Derived from another key.</td>
</tr>
<tr>
<td>POKPSEC</td>
<td>Cleartext keys or key parts that were entered at TKE and secured from there to the target card.</td>
</tr>
</tbody>
</table>

**Key-management field 3, low-order byte**

<table>
<thead>
<tr>
<th>Pedigree current rule (one returned, all key types)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PCUNKWNWN</td>
<td>Unknown.</td>
</tr>
<tr>
<td>PCOTHER</td>
<td>Other. Method other than those defined here, probably used in UDX.</td>
</tr>
<tr>
<td>PRANDOM</td>
<td>Randomly generated.</td>
</tr>
<tr>
<td>PCKEYAGR</td>
<td>Established by key agreement such as Diffie-Hellman.</td>
</tr>
<tr>
<td>PCCCLRC</td>
<td>Created from cleartext key components.</td>
</tr>
<tr>
<td>PCCCLRKV</td>
<td>Entered as a cleartext key value.</td>
</tr>
<tr>
<td>PCDERVD</td>
<td>Derived from another key.</td>
</tr>
<tr>
<td>PCMVARWP</td>
<td>Imported from CCA version X'05' variable-length symmetric key-token with pedigree field.</td>
</tr>
</tbody>
</table>
Key Token Parse2 (CSNBKTP2)

Table 52. Keywords for Key Token Parse2 control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCMVARNP</td>
<td>Imported from CCA version X'05' variable-length symmetric key-token with no pedigree field.</td>
</tr>
<tr>
<td>PCMWCV</td>
<td>Imported from CCA key-token that contained a nonzero control vector.</td>
</tr>
<tr>
<td>PCMNOCV</td>
<td>Imported from CCA key-token that had no control vector or contained a zero control vector.</td>
</tr>
<tr>
<td>PCMT31WC</td>
<td>Imported from a TR-31 key block that contained a control vector (ATTR-CV option).</td>
</tr>
<tr>
<td>PCMT31NC</td>
<td>Imported from a TR-31 key block that did not contain a control vector.</td>
</tr>
<tr>
<td>PCMPK1-2</td>
<td>Imported using PKCS 1.2 RSA encryption.</td>
</tr>
<tr>
<td>PCMOAEP</td>
<td>Imported using PKCS OAEP encryption.</td>
</tr>
<tr>
<td>PCMPKA92</td>
<td>Imported using PKA92 RSA encryption.</td>
</tr>
<tr>
<td>PCMZ-PAD</td>
<td>Imported using RSA ZERO-PAD encryption.</td>
</tr>
<tr>
<td>PCCNVTWC</td>
<td>Converted from a CCA key-token that contained a nonzero control vector.</td>
</tr>
<tr>
<td>PCCNVTNC</td>
<td>Converted from a CCA key-token that had no control vector or contained a zero control vector.</td>
</tr>
<tr>
<td>PCKPSEC</td>
<td>Cleartext keys or key parts that were entered at TKE and secured from there to the target card.</td>
</tr>
<tr>
<td>PCXVARWP</td>
<td>Exported from CCA version X'05' variable-length symmetric key-token with pedigree field.</td>
</tr>
<tr>
<td>PCXVARNP</td>
<td>Exported from CCA version X'05' variable-length symmetric key-token with no pedigree field.</td>
</tr>
<tr>
<td>PCXOAEP</td>
<td>Exported using PKCS OAEP encryption.</td>
</tr>
</tbody>
</table>

Optional clear key or encrypted AESKW payload section

Payload
Note: Not a keyword. Value returned in the payload variable.

**key_material_state**

Direction: Output
Type: Integer
A pointer to an integer variable containing the indicator for the current state of the key material. The valid values are:
0 No key present (internal or external)
1 Key is clear (internal), payload bit length is clear-key bit length
2 Key is encrypted under a KEK (external)
3 Key is encrypted under the master key (internal)

**payload_bit_length**

Direction: Input/Output
Type: Integer
A pointer to an integer variable containing the number of bits in the token payload. If no key is present, the returned value is 0.

If a clear key is present, the returned value is in the following range:
AES 128, 192, or 256
HMAC 80 - 2048

payload
Key Token Parse2 (CSNBKTP2)

Direction: Output
Type: String

A pointer to a string variable containing the key material payload. The payload parameter must be addressable up to the nearest byte boundary above the payload_bit_length if the payload_bit_length is not a multiple of 8. This field will contain the clear key or the encrypted key material.

key_verification_pattern_type

Direction: Output
Type: Integer

A pointer to an integer variable containing the indicator for the type of key verification pattern used. The valid values are:

0  No KVP
1  AESMK (8 left-most bytes of SHA-256 hash(X'01' || clear AES MK))
2  KEK VP (8 left-most bytes of SHA-256 hash(X'01' || clear KEK))

key_verification_pattern_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the key_verification_pattern parameter. The valid values are 0, 8, or 16. The value 16 is reserved.

key_verification_pattern

Direction: Output
Type: String

A pointer to a string variable containing the key verification pattern (KVP) of the key-encrypting key used to wrap this key. If the key_verification_pattern_type value indicates that a key verification pattern is present, the pattern will be copied from the token, otherwise this variable is empty.

key_wrapping_method

Direction: Output
Type: Integer

A pointer to an integer variable containing the indicator for the encrypted section key-wrapping method used to protect the key payload. The valid values are:

0  NONE (for clear keys or no key)
2  AESKW (for external or internal key wrapped with an AES KEK)
3  PKOAEP2 (for external tokens wrapped with an RSA public key)

key_hash_algorithm

Direction: Output
Type: Integer

A pointer to an integer variable containing the indicator for the hash algorithm used for wrapping in the key token. The valid values are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Hash algorithm</th>
<th>Key-wrapping method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No hash</td>
<td>X'00' (clear key)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02' (AESKW)</td>
</tr>
<tr>
<td>1</td>
<td>SHA-1</td>
<td>X'03' (PKOAEP2)</td>
</tr>
</tbody>
</table>
Key Token Parse2 (CSNKTP2)

<table>
<thead>
<tr>
<th>Value</th>
<th>Hash algorithm</th>
<th>Key-wrapping method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X'00' (clear key)</td>
</tr>
<tr>
<td>2</td>
<td>SHA-256</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SHA-384</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SHA-512</td>
<td></td>
</tr>
</tbody>
</table>

key_name_length
- **Direction**: Input/Output
- **Type**: Integer

A pointer to an integer variable containing the number of bytes of data in the key_name variable. The returned value can be 0 or 64.

key_name
- **Direction**: Output
- **Type**: String

A pointer to a string variable containing the optional key label to be stored in the associated data structure of the key token. If there is no key name, then this variable is empty.

TLV_data_length
- **Direction**: Input/Output
- **Type**: Integer

A pointer to an integer variable containing the number of bytes of data in the TLV_data variable. The returned value is currently always zero.

TLV_data
- **Direction**: Output
- **Type**: String

A pointer to a string variable containing the optional tag-length-value (TLV) section. This field is currently unused and will be empty.

user_associated_data_length
- **Direction**: Input/Output
- **Type**: Integer

The user_associated_data_length parameter is a pointer to an integer variable containing the number of bytes of data in the user_associated_data variable. The returned value is 0 - 255.

user_associated_data
- **Direction**: Output
- **Type**: String

A pointer to a string variable containing the user-associated data to be stored in the key token. This user-definable data is cryptographically bound to the key if it is encrypted. If there is no user-defined associated data, this variable is empty.

reserved_length
- **Direction**: Input/Output
- **Type**: Integer
Key Token Parse2 (CSNBKTP2)

A pointer to an integer variable containing the number of bytes of data in the `reserved` variable. The returned value is zero.

reserved

   Direction: Output
   Type: String

A pointer to a string variable that is reserved. This variable is empty.

Required commands

The required commands for CSNBKTP.

None.

Usage notes

The usage notes for CSNBKTP2.

If an error occurs while processing the input token, all output lengths will be updated to zero and an error will be returned.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTP2J.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKTP2J are shown here.

Format

```java
public native void CSNBKTP2J(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger key_token_length,
    byte[] key_token,
    byte[] key_type,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_material_state,
    hikmNativeInteger payload_bit_length,
    byte[] payload,
    hikmNativeInteger key_verification_pattern_type,
    hikmNativeInteger key_verification_pattern_length,
    byte[] key_verification_pattern,
    hikmNativeInteger key_wrapping_method,
    hikmNativeInteger key_hash_algorithm,
    hikmNativeInteger key_name_length,
    byte[] key_name,
    hikmNativeInteger TLV_data_length,
    byte[] TLV_data,
    hikmNativeInteger user_associated_data_length,
    byte[] user_associated_data,
    hikmNativeInteger reserved_length,
    byte[] reserved);
```
Key Translate (CSNBKTR)

The Key Translate verb uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.

Note: All key labels must be unique.

Format

The format of CSNBKTR.

```
CSNBKTR(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   input_key_token,
   input_KEK_key_identifier,
   output_KEK_key_identifier,
   output_key_token)
```

Parameters

The parameters for CSNBKTR.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`input_key_token`

Direction: Input
Type: String

A 64-byte string variable containing an external key token. The external key token contains the key to be re-enciphered (translated).

`input_KEK_key_identifier`

Direction: Input/Output
Type: String

A 64-byte string variable containing the internal key token or the key label of an internal key token record in the DES key storage file. The internal key token contains the key-encrypting key used to decipher the key. The internal key token must contain a control vector that specifies an IMPORTER or IKEYXLAT key type. The control vector for an IMPORTER key must have the XLATE bit set to B'1'.

`output_KEK_key_identifier`

Direction: Input/Output
Type: String

A 64-byte string variable containing the internal key token or the key label of an internal key token record in the DES key storage file. The internal key token contains the key-encrypting key used to encipher the key. The internal key token must contain a control vector that specifies an EXPORTER or OKEYXLAT key type. The control vector for an EXPORTER key must have the XLATE bit set to B'1'.

`output_key_token`
Key Translate (CSNBKTR)

Direction: Output
Type: String

A 64-byte string variable containing an external key token. The external key token contains the re-enciphered key.

Restrictions
The restrictions for CSNBKTR.

Triple length DATA key tokens are not supported.

Required commands
The required commands for CSNBKTR.

This verb requires the Key Translate command (offset X'001F') to be enabled in the active role.

Usage notes
The usage notes for CSNBKTR.

None

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBKTRJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKTRJ are shown here.

Format
public native void CSNBKTRJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] input_key_token,
    byte[] input_KEK_key_identifier,
    byte[] output_KEK_key_identifier,
    byte[] output_key_token);
The Key Translate2 verb uses one key-encrypting key to decipher an input key and then enciphers this key using another key-encrypting key within the secure environment.

It can also be used to change the wrapping method of the key with a single key-encrypting key.

To reencipher a key token, specify the external key token, input and output key-encrypting keys. You can specify which key wrapping method to use. If no wrapping method is specified, the wrapping method of the input_key_token will be used.

To change the wrapping method of an external key token, specify the REFORMAT rule array keyword, the wrapping method to use, the external key token, and the input key-encrypting key. If no wrapping method is specified, the wrapping method of the input_key_token will be used. Note that the output_KEK_identifier will be ignored.

**Note:** All key labels must be unique.

**Format**

The format of CSNBKTR2.

```
CSNBKTR2(  return_code,
          return_code,
          exit_data_length,
          exit_data,
          rule_array_count,
          rule_array,
          input_key_length,
          input_key_token,
          input_KEK_length,
          input_KEK_identifier,
          output_KEK_length,
          output_KEK_identifier,
          output_key_length,
          output_key_token)
```

**Parameters**

The parameters for CSNBKTR2.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

  *Direction: Input*
  
  *Type: Integer*

  The number of keywords you supplied in the `rule_array` parameter. This value must be 0, 1, 2, or 3.

**rule_array**

  *Direction: Input*
  
  *Type: String*
Keywords that provide control information to the verb. The keywords must be 8 bytes of contiguous storage with the keyword left-aligned in its 8-byte location and padded on the right with blanks. The \textit{rule_array} keywords are described in Table 53.

**Table 53. Keywords for Key Translate2 control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Reencipherment} (Optional)</td>
<td></td>
</tr>
<tr>
<td>REFORMAT</td>
<td>Rewrap the \textit{input_key_token} with the key wrapping method specified. Only the \textit{input_KEK_identifier} will be used. The \textit{output_KEK_identifier} is ignored.</td>
</tr>
<tr>
<td>\textit{Key-wrapping method} (One optional)</td>
<td></td>
</tr>
<tr>
<td>USECONFIG</td>
<td>Wrap the key using the configuration setting for the default wrapping method. This is the default.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>Use enhanced key wrapping method, which is compliant with the ANSI X9.24 standard.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Use original key wrapping method, which uses ECB wrapping for DES key tokens and CBC wrapping for AES key tokens.</td>
</tr>
<tr>
<td>\textit{Translation control} (Optional)</td>
<td></td>
</tr>
<tr>
<td>ENH-ONLY</td>
<td>Restrict rewrapping of the \textit{output_key_token}. After the token has been wrapped with the enhanced method, it cannot be rewrapped using the original method.</td>
</tr>
</tbody>
</table>

**input_key_length**

- Direction: Input
- Type: Integer

The length of the \textit{input_key_token} in bytes. The maximum value allowed is 725.

**input_key_token**

- Direction: Input
- Type: String

A variable length string variable containing the external key token. The external key token contains the key to be re-enciphered (or rewrapped).

**input_KEK_length**

- Direction: Input
- Type: Integer

The length of the \textit{input_KEK_identifier} in bytes. The maximum value allowed is 725.

**input_KEK_identifier**

- Direction: Input/Output
- Type: String

A variable length string variable containing the internal key token or the key label of an internal key token record in the key storage file. The internal key token contains the key-encrypting key used to decipher the key. The internal key token must contain a control vector that specifies an IMPORTER or IKEYXLAT key type. The control vector for an IMPORTER key must have the XLATE bit set to B1.

**output_KEK_length**

- Direction: Input
- Type: Integer
Key Translate2 (CSNBKTR2)

The length of the output_KEK_identifier in bytes. The maximum value is 725.

If the REFORMAT keyword is specified, this value must be 0.

output_KEK_identifier
  Direction: Input/Output
  Type: String
  A variable length string variable containing the internal key token or the key label of an internal key token record in the key storage file. The internal key token contains the key-encrypting key used to encipher the key. The internal key token must contain a control vector that specifies an EXPORTER or OKEYXLAT key type. The control vector for an exporter key must have the XLATE bit set to B'1'.
  
  If the REFORMAT keyword is specified, this parameter is ignored.

output_key_length
  Direction: Input/Output
  Type: Integer
  On input, the length of the output area provided for the output_key_token. This must be at least 64 bytes. On output, the parameter is updated with the length of the token copied to the output_key_token.

output_key_token
  Direction: Output
  Type: String
  A variable length string variable containing an external key token. The external key token contains the re-enciphered key.

Restrictions

The restrictions for CSNBKTR2.

This verb does not support version X'10' external DES key tokens (RKX key tokens).

This verb was introduced with CCA 4.1.0.

Required commands

The required commands for CSNBKTR2.

This verb requires the Key Translate2 - Allow use of REFORMAT command (offset X'014B') to be enabled in the active role if the REFORMAT reencipherment keyword is used.

Otherwise, the verb requires the Key Translate2 command (offset X'0149') to be enabled.

To use the translation control keyword WRAP-ECB or WRAP-ENH when the default key-wrapping method setting does not match the keyword, the Key Translate2 - Allow wrapping override keywords command (offset X'014A') must be enabled.

If the WRAP-ECB translation-control keyword is specified and the key in the input key token is wrapped by the enhanced wrapping method (WRAP-ENH), the verb
Key Translate2 (CSNBKTR2)

requires the CKDS Conversion2 - Convert from enhanced to original command (offset X'0147') to be enabled. An active role with offset X'0149' enabled can also use the Key Token Change verb to translate a key from the enhanced key-wrapping method to the less-secure legacy method.

The Key Translate2 - Disallow AES version 5 to version 4 conversion (offset X'032A') command prevents CIPHER keys, which are in variable-length AES key tokens (newer version X'05') and wrapped under the AES master-key, from being reformatted into DATA keys, which are in fixed-length AES key tokens (older version X'04') and wrapped under the less-secure DES master-key. This command overrides the Key Translate2 - Allow use of REFORMAT command (offset X'014B').

Usage notes

The usage notes for CSNBKTR2.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKTR2J.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBKTR2J are shown here.

Format

```java
public native void CSNBKTR2J(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger input_key_length,
    byte[] input_key_token,
    hikmNativeInteger input_KEK_length,
    byte[] input_KEK_identifier,
    hikmNativeInteger output_KEK_length,
    byte[] output_KEK_identifier,
    hikmNativeInteger output_key_length,
    byte[] output_key_token);
```
Use this verb to decrypt (unwrap) a formatted key value. This verb unwraps the key, parses it, and returns the parsed value to the application in the clear.

PKCS 1.2 and ZERO-PAD formatting are supported. For PKCS 1.2, the decrypted data is examined to ensure that it meets RSA DSI PKCS #1 block type 2 format specifications. ZERO-PAD is supported only for external or clear RSA private keys.

This verb allows the use of clear or encrypted RSA private keys. If an external clear key token is used, the master keys are not required to be installed in any cryptographic coprocessor and PKA verbs do not have to be enabled.

Format

The format of CSNDPKD.

```
CSNDPKD(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    PKA_enciphered_keyvalue_length,
    PKA_enciphered_keyvalue,
    data_structure_length,
    data_structure,
    PKA_key_identifier_length,
    PKA_key_identifier,
    target_keyvalue_length,
    target_keyvalue)
```

Parameters

The parameters for CSNDPKD.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input  
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1.

`rule_array`

Direction: Input  
Type: String

The keyword that provides control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The `rule_array` keywords are described in Table 54.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Method</td>
<td>(One, required). Specifies the method to use to recover the key value.</td>
</tr>
</tbody>
</table>
PKA Decrypt (CSNDPKD)

Table 54. Keywords for PKA Decrypt control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKCS-1.2</td>
<td>RSA DSI PKCS #1 block type 02 will be used to recover the key value. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>The input PKA_enciphered_keyvalue is decrypted using the RSA private key. The entire result (including leading zeros) will be returned in the target_keyvalue field. The PKA_key_identifier must be an external RSA token or the label of an external token.</td>
</tr>
</tbody>
</table>

PKA_enciphered_keyvalue_length
- Direction: Input
- Type: Integer
- The length of the PKA_enciphered_keyvalue parameter in bytes. The maximum size that you can specify is 256 bytes. The length should be the same as the modulus length of the PKA_key_identifier.

PKA_enciphered_keyvalue
- Direction: Input
- Type: String
- This field contains the key value protected under an RSA public key. This byte-length string is left-aligned within the PKA_enciphered_keyvalue parameter.

data_structure_length
- Direction: Input
- Type: Integer
- This value must be 0.

data_structure
- Direction: Input
- Type: String
- This parameter is ignored.

PKA_key_identifier_length
- Direction: Input
- Type: Integer
- The length of the PKA_key_identifier parameter. When the PKA_key_identifier is a key label, this field specifies the length of the label. The maximum size that you can specify is 2500 bytes.

PKA_key_identifier
- Direction: Input
- Type: String
- An internal RSA private key token, the label of an internal RSA private key token, or an external RSA private key token containing a clear RSA private key in Modulus-Exponent or Chinese Remainder Theorem format. The corresponding public key was used to wrap the key value.

target_keyvalue_length
- Direction: Input/Output
- Type: Integer
The length of the \textit{target_keyvalue} parameter. The maximum size that you can specify is 256 bytes. On return, this field is updated with the actual length of \textit{target_keyvalue}.

If \texttt{ZERO-PAD} is specified, this length will be the same as the \textit{PKA_enciphered_keyvalue_length} which is equal to the RSA modulus byte length.

\textbf{target_keyvalue}

\hspace{1em} \textbf{Direction: Output}
\hspace{1em} \textbf{Type: String}

This field will contain the decrypted, parsed key value. If \texttt{ZERO-PAD} is specified, the decrypted key value, including leading zeros, will be returned.

\textbf{Restrictions}

The restrictions for \texttt{CSNDPKD}.

The exponent of the RSA public key must be odd.

\textbf{Required commands}

The required commands for \texttt{CSNDPKD}.

This verb requires the PKA Decrypt command (offset X'011F') to be enabled in the active role.

\textbf{Usage notes}

The usage notes for \texttt{CSNDPKD}.

The RSA private key must be enabled for key management functions.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

\textbf{JNI version}

This verb has a Java Native Interface (JNI) version, which is named \texttt{CSNDPKDJ}.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for \texttt{CSNDPKDJ} are shown here.

\textbf{Format}

\begin{verbatim}
public native void CSNDPKDJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger enciphered_key_length,
    byte[] enciphered_key,
    hikmNativeInteger data_struct_length,
    byte[] data_struct,
    hikmNativeInteger RSA_private_key_length,
    byte[] RSA_private_key,
    hikmNativeInteger key_value_length,
    byte[] key_value);
\end{verbatim}
PKA Encrypt (CSNDPKE)

This verb encrypts a supplied clear key value under an RSA public key.

The supplied key can be formatted using the **PKCS 1.2** or **ZERO-PAD** methods prior to encryption. The **rule_array** keyword specifies the format of the key prior to encryption.

**Format**

The format of CSNDPKE.

```plaintext
CSNDPKE(    return_code,    reason_code,    exit_data_length,    exit_data,    rule_array_count,    rule_array,    keyvalue_length,    keyvalue,    data_structure_length,    data_structure,    PKA_key_identifier_length,    PKA_key_identifier,    PKA_enciphered_keyvalue_length,    PKA_enciphered_keyvalue)
```

**Parameters**

The parameters for CSNDPKE.

For the definitions of the **return_code**, **reason_code**, **exit_data_length**, and **exit_data** parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

- Direction: Input
- Type: Integer

A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value must be 1.

**rule_array**

- Direction: Input
- Type: String

A keyword that provides control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The **rule_array** keywords are described in Table 55.

**Table 55. Keywords for PKA Encrypt control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Formatting Method</strong> (One, required). Specifies the method to use to format the key value prior to encryption.</td>
<td></td>
</tr>
<tr>
<td>PKCS-1.2</td>
<td>RSA DSI PKCS #1 block type 02 format will be used to format the supplied key value. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>The key value will be padded on the left with binary zeros to the length of the PKA key modulus. The exponent of the public key must be odd.</td>
</tr>
</tbody>
</table>
### PKA Encrypt (CSNDPKE)

#### Table 55. Keywords for PKA Encrypt control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRP</td>
<td>The key value will be padded on the left with binary zeros to the length of the PKA key modulus. The RSA public key can have an even or odd exponent.</td>
</tr>
</tbody>
</table>

- **keyvalue_length**
  - Direction: Input
  - Type: Integer
  - The length of the `keyvalue` parameter. The maximum field size is 256 bytes. The actual maximum size depends on the modulus length of `PKA_key_identifier` and the formatting method you specify in the `rule_array` parameter. See [Usage notes](#) on page 250.

- **keyvalue**
  - Direction: Input
  - Type: String
  - This field contains the supplied clear key value to be encrypted under the `PKA_key_identifier`.

- **data_structure_length**
  - Direction: Input
  - Type: Integer
  - This value must be 0.

- **data_structure**
  - Direction: Input
  - Type: String
  - This field is currently ignored.

- **PKA_key_identifier_length**
  - Direction: Input
  - Type: Integer
  - The length of the `PKA_key_identifier` parameter. When the `PKA_key_identifier` is a key label, this field specifies the length of the label. The maximum size that you can specify is 2500 bytes.

- **PKA_key_identifier**
  - Direction: Input
  - Type: String
  - The RSA public or private key token or the label of the RSA public or private key to be used to encrypt the supplied key value.

- **PKA_enciphered_keyvalue_length**
  - Direction: Input/Output
  - Type: Integer
  - The length of the `PKA_enciphered_keyvalue` parameter in bytes. The maximum size that you can specify is 256 bytes. On return, this field is updated with the actual length of `PKA_enciphered_keyvalue`.
  - This length should be the same as the modulus length of the `PKA_key_identifier`.  

Chapter 6. Managing AES and DES cryptographic keys  249
PKA Encrypt (CSNDPKE)

PKA_enciphered_keyvalue
Direction: Output
Type: String

This field contains the key value protected under an RSA public key. This byte-length string is left-aligned within the PKA_enciphered_keyvalue parameter.

Restrictions
The restrictions for CSNDPKE.

IMPORTANT
Take note of these important restrictions.
• A message can be encrypted provided that it is smaller than the public key modulus.
  The term ‘smaller’ refers to the exact bit count, not the byte count of the modulus. For example, counting bits, the hexadecimal number X’FF’ is several bits longer than the number X’1F’, even though both numbers are one byte long as represented in computer memory.
• The exponent of the RSA public key must be odd unless the MRP keyword is supplied.
• The RSA public key modulus size (key size) is limited by the Function Control Vector to accommodate governmental export and import regulations.

Required commands
The required commands for CSNDPKE.

This verb requires the PKA Encrypt command (offset X’011E’) to be enabled in the active role.

Usage notes
The usage notes for CSNDPKE.
• For RSA DSI PKCS #1 formatting, the key value length must be a minimum of 11 bytes less than the modulus length of the RSA key.
• The hardware configuration sets the limit on the modulus size of keys for key management; thus, this service will fail if the RSA key modulus bit length exceeds this limit.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDPKEJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDPKEJ are shown here.

Format
public native void CSNDPKEJ(  
  hikmNativeInteger return_code,  
  hikmNativeInteger reason_code,  
  hikmNativeInteger exit_data_length,  
  byte[] exit_data,  
  hikmNativeInteger rule_array_count,  
  byte[] rule_array,
hikmNativeInteger key_value_length,
byte[] key_value,
hikmNativeInteger data_struct_length,
byte[] data_struct,
hikmNativeInteger RSA_public_key_length,
byte[] RSA_public_key,
hikmNativeInteger RSA_encipher_length,
byte[] RSA_encipher);
Prohibit Export (CSNBPEX)

Use this verb to modify an operational key so that it cannot be exported.

Format

The format of CSNBPEX.

```java
CSNBPEX(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   key_identifier)
```

Parameters

The parameters for CSNBPEX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`key_identifier`

- **Direction:** Input/Output
- **Type:** String

A 64-byte string variable containing the internal key token to be modified. The returned `key_identifier` will be encrypted under the current master key.

Restrictions

The restrictions for CSNBPEX.

None

Required commands

The required commands for CSNBPEX.

This verb requires the Prohibit Export command (offset X'00CD') to be enabled in the active role.

Usage notes

The usage notes for CSNBPEX.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPEXJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBPEXJ are shown here.

Format

```java
public native void CSNBPEXJ(
   hikmNativeInteger return_code,
   hikmNativeInteger reason_code,
```

252   Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Prohibit Export (CSNBPEX)

```c
hkmNativeInteger exit_data_length,
byte[] exit_data,
byte[] key_identifier);
```
Use this verb to modify an exportable external CCA DES key-token so that its key can no longer be exported.

This verb performs the following functions:

- Multiply deciphers the source key under a key formed by the XOR of the source key’s control vector and the specified key-encrypting key (KEK).
- Turns from on to off the XPORT-OK bit in the source key’s control vector (bit 17).
- Multiply enciphers the key under a key formed by the XOR of the KEK key and the source key’s modified control vector. The encrypted key and the modified control vector are stored in the source-key key token, and the TVV is updated.

Format

The format of CSNBPEXX.

```c
CSNBPEXX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    source_key_token,
    KEK_key_identifier)
```

Parameters

The parameters for CSNBPEXX.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

**source_key_token**

Direction: Input/Output

Type: String

A pointer to a string variable containing an external key-token.

**KEK_key_identifier**

Direction: Input

Type: String

A pointer to a string variable containing an internal key-encrypting token, or the key label of an internal key-encrypting token record.

Restrictions

The restrictions for CSNBPEXX.

This verb does not support version X’10’ external DES key tokens (RKX key tokens).

Required commands

The required commands for CSNBPEXX.
This verb requires the Prohibit Export Extended command (offset X'0301') to be enabled in the active role.

**Usage notes**
The usage notes for CSNBPEXX.

None

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNBPEXXJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBPEXXJ are shown here.

**Format**
```java
public native void CSNBPEXXJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] source_key_token,
    byte[] KEK_key_identifier);
```
Restrict Key Attribute (CSNBRKA)

Use the Restrict Key Attribute verb to modify an exportable internal or external variable-length symmetric key-token so that its key can no longer be exported.

Format

The format of CSNBPEXX.

```c
CSNBRKA (    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array
    key_identifier_length
    key_identifier
    key_encrypting_key_identifier_length
    key_encrypting_key_identifier
    opt_parameter1_length
    opt_parameter1
    opt_parameter2_length
    opt_parameter2)
```

Parameters

The parameters for CSNBPEXX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

Direction: Input
Type: Integer

The number of keywords you supplied in the `rule_array` parameter. This value must be 1 - 10.

**rule_array**

Direction: Input
Type: String

The `rule_array` contains keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The `rule_array` keywords are described in Table 56.

### Table 56. Keywords for Restrict Key Attribute control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token type</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Specifies the key token is an AES key token.</td>
</tr>
<tr>
<td>DES</td>
<td>Specifies the key token is a DES key token.</td>
</tr>
<tr>
<td>HMAC</td>
<td>Specifies the key token is an HMAC key token.</td>
</tr>
<tr>
<td>Export control (One, optional)</td>
<td></td>
</tr>
<tr>
<td>CCAXPORT</td>
<td>For DES internal tokens, set bit 17 of the CV to B'0' to prohibit any export of the key. This rule is only valid for Token type DES.</td>
</tr>
</tbody>
</table>

256 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
### Table 56. Keywords for Restrict Key Attribute control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOEXPORT</td>
<td>Prohibits the key from being exported by any verb. The use of this keyword always causes each available export control attribute to be lowered. If no export control keywords are used, this is the default. Variable-length symmetric key-token: This keyword is equivalent to providing all of the keywords listed under export control for AES or HMAC (NOEX-SYM, NOEXUASY, and so forth). This is the default if no export control for AES or HMAC keywords are used. DES key-token: Use this keyword to set CV bit 17 = B'0' (NO-XPORT) and CV bit 27 = B'1' (NOT31XPT). This is the default if no export control for DES keywords are used.</td>
</tr>
<tr>
<td>NOEX-AES</td>
<td>Specifies to prohibit export using an AES key. This rule is not valid for Token type DES.</td>
</tr>
<tr>
<td>NOEX-DES</td>
<td>Specifies to prohibit export using a DES key. This rule is not valid for Token type DES.</td>
</tr>
<tr>
<td>NOEX-RAW</td>
<td>Specifies to prohibit export in RAW format. This rule is not valid for Token type DES.</td>
</tr>
<tr>
<td>NOEX-RSA</td>
<td>Specifies to prohibit export using an RSA key. This rule is not valid for Token type DES.</td>
</tr>
<tr>
<td>NOEX-SYM</td>
<td>Prohibits the key from being exported using a symmetric key. This rule is not valid for Token type DES.</td>
</tr>
<tr>
<td>NOEXUASY</td>
<td>Prohibits the key from being exported using an unauthenticated asymmetric key. This rule is not valid for Token type DES.</td>
</tr>
<tr>
<td>NOEXAAASY</td>
<td>Prohibits the key from being exported using an authenticated asymmetric key (for example, an RSA key in a trusted block token). This rule is not valid for Token type DES.</td>
</tr>
<tr>
<td>NOT31XPT</td>
<td>For DES internal tokens, set bit 57 of the CV to B'1' to prohibit TR-31 export of the key. This rule is only valid for Token type DES.</td>
</tr>
</tbody>
</table>

**Input Transport Key (Optional)**

<table>
<thead>
<tr>
<th>IVEK-AES</th>
<th>Specifies the KEK is an AES transport key. This is the default for Token types AES and HMAC, and is not allowed with Token type DES.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVEK-DES</td>
<td>Specifies the KEK is a DES transport key Valid only for token type DES. This is the default for Token type DES.</td>
</tr>
<tr>
<td>IVEK-PKA</td>
<td>Specifies the KEK is a PKA transport key. This is not allowed with Token type DES.</td>
</tr>
</tbody>
</table>

**key_identifier_length**

- **Direction:** Input
- **Type:** Integer

The length of the `key_identifier` parameter in bytes. The maximum value is 725.

**key_identifier**

- **Direction:** Input
- **Type:** String

The key for which the export control is to be updated. The parameter contains an internal token or the 64-byte label of the key in key storage. If a label is specified, the key token will be updated in key storage and not returned by this verb.

**key_encrypting_key_identifier_length**

- **Direction:** Input
- **Type:** Integer

The byte length of the `key_encrypting_key_identifier` parameter. This value must be 0.

**key_encrypting_key_identifier**
Restrict Key Attribute (CSNBRKA)

Direction: Input
Type: String
This parameter is ignored.

**opt_parameter1_length**

Direction: Input
Type: Integer
The byte length of the *opt_parameter1* parameter. This value must be 0.

**opt_parameter1**

Direction: Input
Type: String
This parameter is ignored.

**opt_parameter2_length**

Direction: Input
Type: Integer
The byte length of the *opt_parameter2* parameter. This value must be 0.

**opt_parameter2**

Direction: Input
Type: String
This parameter is ignored.

**Restrictions**
The restrictions for CSNBPEXX.

This verb was introduced with CCA 4.1.0.

**Required commands**
The required commands for CSNBPEXX.

The currently supported service (Restrict Export of HMAC tokens) requires the Restrict Key Attribute - Export Control command (offset X'00E9') to be enabled in the active role.

**Usage notes**
The usage notes for CSNBPEXX.

This verb is available starting with CCA 4.1.0.

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNBRKAAJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBRKAAJ are shown here.

**Format**

```java
public native void CSNBRKAAJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
```
Restrict Key Attribute (CSNBRKA)

hikmNativeInteger exit_data_length,
byte[] exit_data,
hikmNativeInteger rule_array_count,
byte[] rule_array,
hikmNativeInteger key_identifier_length,
byte[] key_identifier,
hikmNativeInteger key_encrypting_key_identifier_length,
byte[] key_encrypting_key_identifier,
hikmNativeInteger opt_parameter1_length,
byte[] opt_parameter1,
hikmNativeInteger opt_parameter2_length,
byte[] opt_parameter2);
Random Number Generate (CSNBRNG)

This verb uses the cryptographic feature to generate a cryptographic-quality random number.

Format
The format of CSNBRNG.

```
CSNBRNG(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    form,
    random_number)
```

Parameters
The parameters for CSNBRNG.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`form`
- Direction: Input
- Type: String

The 8-byte keyword that defines the characteristics of the random number should be left-aligned and padded on the right with blanks. The keywords are listed in Table 57.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEN</td>
<td>Generate a 64-bit random number with even parity in each byte.</td>
</tr>
<tr>
<td>ODD</td>
<td>Generate a 64-bit random number with odd parity in each byte.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>Generate a 64-bit random number.</td>
</tr>
</tbody>
</table>

Parity is calculated on the seven high-order bits in each byte and is presented in the low-order bit in the byte.

`random_number`
- Direction: Output
- Type: String

The generated number returned by the verb in an 8-byte variable.

Restrictions
The restrictions for CSNBRNG.

None

Required commands
The required commands for CSNBRNG.
Random Number Generate (CSNBRNG)

This verb requires the Key Generate - OP_IM_EX command (offset X'008E') to be enabled in the active role.

Usage notes
The usage notes for CSNBRNG.

None

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBRNGJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBRNGJ are shown here.

Format
public native void CSNBRNGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] form,
    byte[] random_number);
Random Number Generate Long (CSNBRNGL)

Random Number Generate Long (CSNBRNGL)

This verb uses the cryptographic feature to generate a cryptographic-quality random number from 1 - 8192 bytes in length.

Choose the parity of each generated random byte as even, odd, or random. This verb returns the random number in a string variable.

Because this verb uses cryptographic processes, the quality of the output is better than that which higher-level language compilers typically supply.

Format

The format of CSNBRNGL.

```
CSNBRNGL(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   rule_array_count,
   rule_array,
   seed_length,
   seed,
   random_number_length,
   random_number)
```

Parameters

The parameters for CSNBRNGL.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1.

`rule_array`

Direction: Input
Type: String

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length, and must be left-aligned and padded on the right with space characters. The `rule_array` keywords are described in Table 58.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity adjust (One required)</td>
<td></td>
</tr>
<tr>
<td>EVEN</td>
<td>Specifies that each generated random byte is adjusted for even parity.</td>
</tr>
<tr>
<td>ODD</td>
<td>Specifies that each generated random byte is adjusted for odd parity.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>Specifies that each generated random byte is not adjusted for parity.</td>
</tr>
</tbody>
</table>

`seed_length`
Random Number Generate Long (CSNBRNGL)

**Direction: Input**
- **Type: Integer**
  
  A pointer to an integer variable containing the number of bytes in the seed variable. This value must be 0.

**seed**

**Direction: Input**
- **Type: String**
  
  This parameter is ignored.

**random_number_length**

**Direction: Input/Output**
- **Type: Integer**
  
  A pointer to an integer variable containing the number of bytes in the random_number variable. On input, the minimum value is 1 and the maximum value is 8192.
  
  Use this variable to specify the number of random bytes that the verb is to return. On output, this variable contains the number of bytes returned by the verb in the random_number variable.

**random_number**

**Direction: Output**
- **Type: String**
  
  A pointer to a string variable containing the random number generated.

**Restrictions**

The restrictions for CSNBRNGL.

None

**Required commands**

The required commands for CSNBRNGL.

None

**Usage notes**

The usage notes for CSNBRNGL.

None

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBRNGLJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBRNGLJ are shown here.

**Format**

```java
public native void CSNBRNGLJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
```
Random Number Generate Long (CSNBRNGL)

byte[] exit_data,
hikmNativeInteger rule_array_count,
byte[] rule_array,
hikmNativeInteger reserved_seed_length,
byte[] reserved_seed,
hikmNativeInteger random_number_length,
byte[] random_number;
Symmetric Key Export (CSNDSYX)

Use the Symmetric Key Export verb to transfer an application-supplied AES DATA (version X'04'), DES DATA, or variable-length symmetric key token key from encryption under the AES or DES master key to encryption under an application-supplied RSA public key or AES EXPORTER key.

The application-supplied key must be an AES, DES or HMAC internal key token or the label of an AES or DES key token in the AES or DES key storage file.

Beginning with CCA 4.1.0, the verb can also export an HMAC key that is contained in an internal variable-length symmetric key-token. The exported key is returned in an external variable-length symmetric key-token.

Use the Symmetric Key Import verb to import a key exported using the AES or DES algorithm, and the Symmetric Key Import2 verb to import a key exported using the HMAC algorithm.

Different methods are supported for formatting the output key. Not all of these methods are available for each supported source key-token. The AESKW key-formatting method uses an AES EXPORTER key-encrypting key to wrap the output key before returning it in an external variable-length symmetric key-token. The other key formatting methods each use a different scheme to format the key before it is enciphered using an asymmetric RSA public-key. The formatted and enciphered key is returned as an opaque data buffer, and is not in a key token.

Format

The format of CSNDSYX.

```c
CSNDSYX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    source_key_identifier_length,
    source_key_identifier,
    transport_key_identifier_length,
    transport_key_identifier,
    enciphered_key_length,
    enciphered_key)
```

Parameters

The parameters for CSNDSYX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer
A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1, 2, or 3.

`rule_array`
**Symmetric Key Export (CSNDSYX)**

*Direction: Input*
*Type: String*

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The *rule_array* keywords are described in Table 59.

**Table 59. Keywords for Symmetric Key Export control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Export an AES key. If <em>source_key_identifier</em> is a variable-length symmetric key token or label, only the <strong>PKOAEP2</strong> and <strong>AESKW</strong> key formatting methods are supported.</td>
</tr>
<tr>
<td>DES</td>
<td>Export a DES key. This is the default.</td>
</tr>
<tr>
<td>HMAC</td>
<td>Export an HMAC key. Only the <strong>PKOAEP2</strong> and <strong>AESKW</strong> key formatting methods are supported.</td>
</tr>
<tr>
<td><strong>Recovery method</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>AESKW</td>
<td>Specifies that the key is to be formatted using <strong>AESKW</strong> and placed in an external variable length CCA token. The <em>transport_key_identifier</em> must be an AES EXPORTER. This rule is not valid with the DES algorithm keyword or with AES DATA (version X'04') keys.</td>
</tr>
<tr>
<td>PKCSOAEP</td>
<td>Specifies using the method found in RSA DSI PKCS #1V2 OAEP. See “PKCS #1 formats” on page 720. The default hash method is SHA-1. Use the SHA-256 keyword for the SHA-256 hash method. Use the SHA-384 keyword for the SHA-384 hash method. Use the SHA-512 keyword for the SHA-512 hash method.</td>
</tr>
<tr>
<td>PKCS-1.2</td>
<td>Specifies using the method found in RSA DSI PKCS #1 block type 02 to recover the symmetric key. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format. See “PKCS #1 formats” on page 720.</td>
</tr>
<tr>
<td>PKOAEP2</td>
<td>Specifies that the key is formatted as defined in the RSA PKCS #1 v2.1 standard for the RSAES-OAEP encryption mechanism. Valid only with algorithm HMAC. This keyword was introduced with CCA 4.1.0. See “PKCS #1 formats” on page 720.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>The clear key is right-aligned in the field provided, and the field is padded to the left with zeros up to the size of the RSA encryption block (which is the modulus length).</td>
</tr>
<tr>
<td><strong>Hash method</strong> (One, optional for PKCSOAEP, required for PKOAEP2. Not valid with any other key formatting method)</td>
<td></td>
</tr>
<tr>
<td>SHA-1</td>
<td>Specifies to use the SHA-1 hash method to calculate the OAEP message hash. Valid only with key-formatting methods PKCSOAEP or PKOAEP2. This is the default for PKCSOAEP.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Specifies to use the SHA-256 hash method to calculate the OAEP message hash. Valid only with key-formatting methods PKCSOAEP or PKOAEP2.</td>
</tr>
<tr>
<td>SHA-384</td>
<td>Specifies to use the SHA-384 hash method to calculate the OAEP message hash. Valid only with key-formatting method PKOAEP2.</td>
</tr>
<tr>
<td>SHA-512</td>
<td>Specifies to use the SHA-512 hash method to calculate the OAEP message hash. Valid only with key-formatting method PKOAEP2.</td>
</tr>
</tbody>
</table>

**source_key_identifier_length**

*Direction: Input*
*Type: Integer*

The length of the *source_key_identifier* parameter. The minimum size is 64 bytes. The maximum size is 725 bytes.

**source_key_identifier**

*Direction: Input*
*Type: String*
Symmetric Key Export (CSNDSYX)

The label or internal token of a secure AES DATA (version X'04'), DES DATA, or variable-length symmetric key token to encrypt under the supplied RSA public key or AES EXPORTER key. The key in the key identifier must match the algorithm in the rule_array. DES is the default algorithm.

transporter_key_identifier_length

Direction: Input
Type: Integer

The length of the transporter_key_identifier parameter. The maximum size is 3500 bytes for an RSA key token or 725 for an AES EXPORTER key token. The length must be 64 if transporter_key_identifier is a label.

transporter_key_identifier

Direction: Input
Type: String

A pointer to a string variable containing an RSA public key token, AES EXPORTER token, or label of the key to protect the exported symmetric key.

When the AESKW key formatting method is specified, this parameter must be an AES EXPORTER key token or label with the EXPORT bit on in the key-usage field. Otherwise, this parameter must be an RSA public key token or label.

enciphered_key_length

Direction: Input/Output
Type: Integer

The length of the enciphered_key parameter. This variable is updated with the actual length of the enciphered_key generated. The maximum size you can specify in this parameter is 900 bytes.

enciphered_key

Direction: Output
Type: String

A pointer to a string variable containing the key after it has been formatted and enciphered by the transport key. The enciphered key is returned either as an opaque data buffer or in an external variable-length symmetric key-token. For key-formatting method PKOAEP2, the key token has no key verification pattern.

Restrictions

The restrictions for CSNDSYX.

- The RSA public-key modulus size (key length) is limited by the Function Control Vector (FCV) to accommodate potential government export and import regulations.
- Retained keys are not supported.
- The maximum public exponent is 17 bits for any key that has a modulus greater than 2048 bits.

Required commands

The required commands for CSNDSYX.

This verb requires the following commands to be enabled in the active role based on the key-formatting method and the algorithm:
Symmetric Key Export (CSNDSYX)

<table>
<thead>
<tr>
<th>Key-formatting method</th>
<th>Algorithm</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESKW</td>
<td>AES</td>
<td>X'0327'</td>
<td>Export AES Key (AESKW)</td>
</tr>
<tr>
<td>PKOAEP2</td>
<td>AES</td>
<td>X'00FC'</td>
<td>Symmetric Key Export - AES, PKOAEP2</td>
</tr>
<tr>
<td>PKOAEP2</td>
<td>HMAC</td>
<td>X'00F5'</td>
<td>Symmetric Key Export - HMAC_PKCSOAEP</td>
</tr>
<tr>
<td>PKCSOAEP or PKCS-1.2</td>
<td>AES</td>
<td>X'0130'</td>
<td>Symmetric Key Export - AES_PKCSOAEP_PKCS-1.2</td>
</tr>
<tr>
<td>PKCSOAEP or PKCS-1.2</td>
<td>DES</td>
<td>X'0105'</td>
<td>Symmetric Key Export - DES_PKCS-1.2</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>AES</td>
<td>X'0131'</td>
<td>Symmetric Key Export - AES_ZERO-PAD</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>DES</td>
<td>X'023E'</td>
<td>Symmetric Key Export - DES_ZERO-PAD</td>
</tr>
</tbody>
</table>

To disallow the wrapping of a key with a weaker key-encrypting key, enable the Disallow Weak Key Wrap command (offset X'0328') in the active role. This command affects multiple verbs. See Appendix G, “Access control points and verbs,” on page 723.

To receive a warning when wrapping a key with a weaker key-encrypting key, enable the Warn when Wrapping Weak Keys command (offset X'032C') in the active role. The Disallow Weak Key Wrap command (offset X'0328') overrides this command.

Usage notes

The usage notes for CSNDSYX.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

The strength of the exporter key expected by Symmetric Key Export depends on the attributes of the key being exported. The resulting return code and reason code when using an exporter KEK that is weaker depends on the Disallow Weak Key Wrap command (offset X'0328') and Warn when Wrapping Weak Keys command (offset X'032C') access control points:

- If the Disallow Weak Key Wrap command (offset X'0328') access control point is disabled (the default), the key strength requirement will not be enforced. Using a weaker key will result in return code 0 with a nonzero reason code if the Warn when Wrapping Weak Keys command (offset X'032C') access control point is enabled. Otherwise, a reason code of zero will be returned.
- If the Disallow Weak Key Wrap (offset X'0328') access control point is enabled (using TKE), the key strength requirement will be enforced, and attempting to use a weaker key will result in return code 8.

For AES DATA and AES CIPHER keys, the AES EXPORTER key must be at least as long as the key being exported to be considered sufficient strength.

Note that wrapping an AES 192-bit key or an AES 256-bit key with any RSA key will always be considered a weak wrap.

For HMAC keys, the AES EXPORTER must be sufficient strength as described in Table 60 on page 269.
Table 60. AES EXPORTER strength required for exporting an HMAC key under an AES EXPORTER

<table>
<thead>
<tr>
<th>Key-usage field 2 in the HMAC key contains</th>
<th>Minimum strength of AES EXPORTER to adequately protect the HMAC key</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-256, SHA-384, SHA-512</td>
<td>256 bits</td>
</tr>
<tr>
<td>SHA-224</td>
<td>192 bits</td>
</tr>
<tr>
<td>SHA-1</td>
<td>128 bits</td>
</tr>
</tbody>
</table>

If an RSA public key is specified as the `transporter_key_identifier`, the RSA key used must have a modulus size greater than or equal to the total PKOAEP2 message bit length (key size plus total overhead), as described in Table 61.

Table 61. Minimum RSA modulus strength required to contain a PKOAEP2 block when exporting an AES key

<table>
<thead>
<tr>
<th>AES key size</th>
<th>Total message sizes (and therefore minimum RSA key size) when the hash method is:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SHA-1</td>
</tr>
<tr>
<td>128 bits</td>
<td>736 bits</td>
</tr>
<tr>
<td>192 bits</td>
<td>800 bits</td>
</tr>
<tr>
<td>256 bits</td>
<td>800 bits</td>
</tr>
</tbody>
</table>

For AES keys, the AES EXPORTER must be sufficient strength as described in Table 62.

Table 62. Minimum RSA modulus length to adequately protect an AES key

<table>
<thead>
<tr>
<th>AES key to be exported</th>
<th>Minimum strength of RSA wrapping key to adequately protect the AES key</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES 128</td>
<td>3072 bits</td>
</tr>
<tr>
<td>AES 192</td>
<td>7860 bits</td>
</tr>
<tr>
<td>AES 256</td>
<td>15360 bits</td>
</tr>
</tbody>
</table>

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSYXJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDSYXJ are shown here.

Format

```java
public native void CSNDSYXJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger source_key_identifier_length,
    byte[] source_key_identifier,
    hikmNativeInteger transporter_key_identifier_length,
);```

Chapter 6. Managing AES and DES cryptographic keys 269
Symmetric Key Export (CSNDSYX)

byte[] transporter_key_identifier,
hikmNativeInteger enciphered_key_length,
byte[] enciphered_key);
Symmetric Key Generate (CSNDSYG)

Use the Symmetric Key Generate verb to generate an AES or DES DATA key and return the key in two forms: enciphered under the master key and encrypted under an RSA public key.

You can import the RSA public key encrypted form by using the Symmetric Key Import or Symmetric Key Import2 verbs at the receiving node.

Also use the Symmetric Key Generate verb to generate any DES importer or exporter key-encrypting key encrypted under a RSA public key according to the PKA92 formatting structure. See “PKA92 key format and encryption process” on page 718 for more details about PKA92 formatting.

Format

The format of CSNDSYG.

```c
CSNDSYG(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_encrypting_key_identifier,
    RSA_public_key_identifier_length,
    RSA_public_key_identifier,
    local_enciphered_key_identifier_length,
    local_enciphered_key_identifier,
    RSA_enciphered_key_length,
    RSA_enciphered_key
)
```

Parameters

The parameters for CSNDSYG.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`rule_array_count`

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1 - 7.

`rule_array`

**Direction:** Input  
**Type:** String

Keywords that provide control information to the verb. The recovery method is the method to use to recover the symmetric key. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 63 on page 272.
Symmetric Key Generate (CSNDSYG)

Table 63. Keywords for Symmetric Key Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong></td>
<td>(One, optional)</td>
</tr>
<tr>
<td>AES</td>
<td>Specifies to generate an AES key.</td>
</tr>
<tr>
<td>DES</td>
<td>Specifies to generate a DES key. This is the default.</td>
</tr>
<tr>
<td><strong>Key-formatting method</strong></td>
<td>(One required)</td>
</tr>
<tr>
<td>PKA92</td>
<td>Specifies the key-encrypting key is to be encrypted under a PKA96 RSA public key according to the PKA92 formatting structure.</td>
</tr>
<tr>
<td>PKCSOAEP</td>
<td>Specifies to use the method found in RSA DSI PKCS #1V2 OAEP. Supported by the DES and AES algorithms. The default hash method is SHA-1. Use the SHA-256 keyword for the SHA-256 hash method.</td>
</tr>
<tr>
<td>PKCS-1.2</td>
<td>Specifies the method found in RSA DSI PKCS #1 block type 02. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>The clear key is right-aligned in the field provided, and the field is padded to the left with zeros up to the size of the RSA encryption block (which is the modulus length).</td>
</tr>
<tr>
<td><strong>Key length</strong></td>
<td>(One, optional use with PKA92)</td>
</tr>
<tr>
<td>SINGLE-R</td>
<td>Generates a key-encrypting key that has equal left and right halves allowing it to perform as a single-length key. Valid only for the recovery method of PKA92.</td>
</tr>
<tr>
<td><strong>Key length</strong></td>
<td>(One, optional use with PKCSOAEP, PKCS-1.2, or ZERO-PAD)</td>
</tr>
<tr>
<td>SINGLE, KEYLN8</td>
<td>Generates a single-length DES key. This is the default for DES keys.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Generates a double-length DES key. Valid only for DES keys.</td>
</tr>
<tr>
<td>KEYLN16</td>
<td>Generates a double-length DES DATA key. This is the default for AES keys.</td>
</tr>
<tr>
<td>KEYLN24</td>
<td>Generates a triple-length DES DATA key. Valid only for AES keys</td>
</tr>
<tr>
<td>KEYLN32</td>
<td>Generates a 32-byte AES key. Valid only for AES keys</td>
</tr>
<tr>
<td><strong>Encipherment method for the local enciphered copy of the key</strong></td>
<td>(One, optional for use with PKCSOAEP, PKCS-1.2, and ZERO-PAD)</td>
</tr>
<tr>
<td>EX</td>
<td>The DES enciphered key is enciphered by an EXPORTER key that is provided through the key_encrypting_key_identifier parameter.</td>
</tr>
<tr>
<td>IM</td>
<td>The DES enciphered key is enciphered by an IMPORTER key that is provided through the key_encrypting_key_identifier parameter.</td>
</tr>
<tr>
<td>OP</td>
<td>The DES enciphered key is enciphered by the master key. The key_encrypting_key_identifier parameter is ignored. This is the default.</td>
</tr>
<tr>
<td><strong>Key-wrapping method</strong></td>
<td>(One, optional)</td>
</tr>
<tr>
<td>USECONFIG</td>
<td>Specifies to wrap the key using the configuration setting for the default wrapping method. This keyword is ignored for AES keys. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>Use enhanced key wrapping method, which is compliant with the ANSI X9.24 standard.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Use original key wrapping method, which uses ECB wrapping for DES key tokens and CBC wrapping for AES key tokens.</td>
</tr>
<tr>
<td><strong>Translation control</strong></td>
<td>(Optional) This is valid only with key-wrapping method WRAP-ENH or with USECONFIG when the default wrapping method is WRAP-ENH. This option cannot be used on a key with a control vector valued to binary zeros.</td>
</tr>
<tr>
<td>ENH-ONLY</td>
<td>Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B’1’. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td><strong>Hash method</strong></td>
<td>(Optional). Valid only with keyword PKCSOAEP.</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Specifies to use the SHA-1 hash method to calculate the OAEP message hash. This is the default.</td>
</tr>
</tbody>
</table>
**Table 63. Keywords for Symmetric Key Generate control information (continued)**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-256</td>
<td>Specifies to use the SHA-256 hash method to calculate the OAEP message hash.</td>
</tr>
</tbody>
</table>

**key_encrypting_key_identifier**

- **Direction**: Input/Output
- **Type**: String

The label or internal token of a key-encrypting key. If the `rule_array` specifies IM, this DES key must be an IMPORTER. If the `rule_array` specifies EX, this DES key must be an EXPORTER.

**RSA_public_key_identifier_length**

- **Direction**: Input
- **Type**: Integer

The length of the `RSA_public_key_identifier` parameter. If the `RSA_public_key_identifier` parameter is a label, this parameter specifies the length of the label. The maximum size is 3500 bytes.

**RSA_public_key_identifier**

- **Direction**: Input
- **Type**: String

The token, or label, of the RSA public key to be used for protecting the generated symmetric key.

**local_enciphered_key_identifier_length**

- **Direction**: Input/Output
- **Type**: Integer

The length of the `local_enciphered_key_identifier`. This field is updated with the actual length of the `local_enciphered_key_identifier` that is generated. The maximum length is 3500 bytes. However, this value should be 64 as in current CCA practice a DES key-token or a key label is always a 64-byte structure.

**local_enciphered_key_identifier**

- **Direction**: Input/Output
- **Type**: String

A pointer to a string variable containing either a key name or a key token. The control vector for the local key is taken from the identified key token. On output, the generated key is inserted into the identified key token.

On input, you must specify a token type consistent with your choice of local-key encryption. If you specify IM or EX, you must specify an external key-token. Otherwise, specify an internal key-token or a null key-token.

When PKCSOAEP, PKCS-1.2, or ZERO-PAD is specified, a null key-token can be specified. In this case, an AES DATA or DES DATA key is returned. For an internal key (OP), a default AES DATA or DATA control-vector is returned in the key token. For an external key (IM or EX), the control vector is set to null.

**RSA_enciphered_key_length**

- **Direction**: Input/Output
- **Type**: Integer
Symmetric Key Generate (CSNDSYG)

The length of the `RSA_enciphered_key` parameter. This verb updates this with the actual length of the `RSA_enciphered_key` it generates. The maximum size is 3500 bytes.

`RSA_enciphered_key`
- **Direction:** Input/Output
- **Type:** String

A pointer to a string variable containing the generated RSA-enciphered key returned by the verb. If you specify `PKCSOAEP`, `PKCS-1.2`, or `ZERO-PAD`, on input specify a null key token. If you specify `PKA92`, on input specify an internal (operational) CCA DES key-token.

**Restrictions**
The restrictions for CSNDSYG.

None

**Required commands**
The required commands for CSNDSYG.

This verb requires the following commands to be enabled in the active role based on the key-formatting method and the algorithm:

<table>
<thead>
<tr>
<th>Key-formatting method</th>
<th>Algorithm</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKCSOAEP or PKCS-1.2</td>
<td>AES</td>
<td>'X'012C'</td>
<td>Symmetric Key Generate - AES_ PKCSOAEP_ PKCS-1.2</td>
</tr>
<tr>
<td>PKCSOAEP or PKCS-1.2</td>
<td>DES</td>
<td>'X'023F'</td>
<td>Symmetric Key Generate - DES_ PKCS-1.2</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>AES</td>
<td>'X'012D'</td>
<td>Symmetric Key Generate - AES_ ZERO-PAD</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>DES</td>
<td>'X'023C'</td>
<td>Symmetric Key Generate - DES_ ZERO-PAD</td>
</tr>
<tr>
<td>PKA92</td>
<td>DES</td>
<td>'X'010D'</td>
<td>Symmetric Key Generate - DES_ PKA92</td>
</tr>
</tbody>
</table>

The use of the `WRAP-ECB` or `WRAP-ENH` key-wrapping method keywords requires the Symmetric Key Generate - Allow wrapping override keywords command (offset `X'013E'`) to be enabled.

**Usage notes**
The usage notes for CSNDSYG.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

Specification of `PKA92` with an input NOCV key-encrypting key token is not supported.

Use the `PKA92` key-formatting method to generate a key-encrypting key. The verb enciphers one key copy using the key encipherment technique employed in the IBM Transaction Security System (TSS) 4753, 4755, and AS/400® cryptographic product PKA92 implementations (see “PKA92 key format and encryption process” on page 718.)
Symmetric Key Generate (CSNDSYG)

The control vector for the RSA-enciphered copy of the key is taken from an internal (operational) DES key token that must be present on input in the RSA_enciphered_key variable.

Only key-encrypting keys that conform to the rules for an OPEX case under the Key Generate verb are permitted. The control vector for the local key is taken from a DES key token that must be present on input in the local_enciphered_key_identifier variable. The control vector for one key copy must be from the EXPORTER class, while the control vector for the other key copy must be from the IMPORTER class.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSYGJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDSYGJ are shown here.

Format

```java
public native void CSNDSYGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_encrypting_key_identifier,
    hikmNativeInteger RSA_public_key_identifier_length,
    byte[] RSA_public_key_identifier,
    hikmNativeInteger local_enciphered_key_identifier_length,
    byte[] local_enciphered_key_identifier,
    hikmNativeInteger RSA_enciphered_key_token_length,
    byte[] RSA_enciphered_key_token);
```
Symmetric Key Import (CSNDSYI)

Symmetric Key Import (CSNDSYI)

Use the Symmetric Key Import verb to import a symmetric AES DATA or DES DATA key enciphered under an RSA public key. The verb returns the key in operational form, enciphered under the master key.

This verb also supports import of a PKA92-formatted DES key-encrypting key under a PKA96 RSA public key.

Format

The format of CSNDSYI.

```plaintext
CSNDSYI(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    RSA_enciphered_key_length,
    RSA_enciphered_key,
    RSA_private_key_identifier_length,
    RSA_private_key_identifier,
    target_key_identifier_length,
    target_key_identifier)
```

Parameters

The parameters for CSNDSYI.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1-5.

`rule_array`

Direction: Input

Type: String

The keyword that provides control information to the verb. The recovery method is the method to use to recover the symmetric key. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The `rule_array` keywords are described in Table 64.

Table 64. Keywords for Symmetric Key Import control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm (One, optional)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Export an AES key.</td>
</tr>
<tr>
<td>DES</td>
<td>Export a DES key. This is the default.</td>
</tr>
<tr>
<td>Recovery method (One required)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 64. Keywords for Symmetric Key Import control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKA92</td>
<td>Specifies the key-encrypting key is encrypted under a PKA96 RSA public key according to the PKA92 formatting structure.</td>
</tr>
<tr>
<td>PKCSOAEP</td>
<td>Specifies to use the method found in RSA DSI PKCS #1V2 OAEP. Supported by the DES and AES algorithms. The default hash method is SHA-1. Use the SHA-256 keyword for the SHA-256 hash method.</td>
</tr>
<tr>
<td>PKCS-1.2</td>
<td>Specifies the method found in RSA DSI PKCS #1 block type 02. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>The clear key is right-aligned in the field provided, and the field is padded to the left with zeros up to the size of the RSA encryption block (which is the modulus length).</td>
</tr>
</tbody>
</table>

**Key-wrapping method** (One, optional)

| USECONFG   | Specifies to wrap the key using the configuration setting for the default wrapping method. This keyword is ignored for AES keys. This is the default. This keyword was introduced with CCA 4.1.0. |
| WRAP-ENH   | Specifies to wrap the key using the legacy wrapping method. This keyword is ignored for AES keys. This keyword was introduced with CCA 4.1.0. |
| WRAP-ECB   | Specifies to wrap the key using the enhanced wrapping method. Valid only for DES keys. This keyword was introduced with CCA 4.1.0. |

**Translation control** (Optional) This is valid only with key-wrapping method **WRAP-ENH** or with **USECONFG** when the default wrapping method is **WRAP-ENH**. This option cannot be used on a key with a control vector valued to binary zeros.

| ENH-ONLY   | Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'. This keyword was introduced with CCA 4.1.0. |

**Hash method** (Optional). Valid only with keyword **PKCSOAEP**.

| SHA-1      | Specifies to use the SHA-1 hash method to calculate the OAEP message hash. This is the default. |
| SHA-256    | Specifies to use the SHA-256 hash method to calculate the OAEP message hash. |

### Symmetric Key Import (CSNDSYI)

**RSA_enciphered_key_length**

**Direction:** Input  
**Type:** Integer  

The length of the `RSA_enciphered_key` parameter. The maximum size is 3500 bytes.

**RSA_enciphered_key**

**Direction:** Input  
**Type:** String  

The key to import, protected under an RSA public key. The encrypted key is in the low-order bits (right-aligned) of a string whose length is the minimum number of bytes that can contain the encrypted key. This string is left-aligned within the `RSA_enciphered_key` parameter.

**RSA_private_key_identifier_length**

**Direction:** Input  
**Type:** Integer  

The length of the `RSA_private_key_identifier` parameter. When the `RSA_private_key_identifier` parameter is a key label, this field specifies the length of the label. The maximum size is 3500 bytes.
Symmetric Key Import (CSNDSYI)

**RSA_private_key_identifier**
- **Direction:** Input
- **Type:** String

An internal RSA private key token or label whose corresponding public key protects the symmetric key.

**target_key_identifier_length**
- **Direction:** Input/Output
- **Type:** Integer

The length of the `target_key_identifier` parameter. This field is updated with the actual length of the `target_key_identifier` that is generated. The maximum length is 3500 bytes.

**target_key_identifier**
- **Direction:** Input/Output
- **Type:** String

This field contains the internal token of the imported symmetric key.

Except for **PKA92** processing, this verb produces a DATA key token with a key of the same length as that contained in the imported token.

**Restrictions**
The restrictions for CSNDSYI.

None.

**Required commands**
The required commands for CSNDSYI.

This verb requires the following commands to be enabled in the active role based on the key-formatting method and the algorithm:

<table>
<thead>
<tr>
<th>Key-formatting method</th>
<th>Algorithm</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKA92 and</td>
<td>DES</td>
<td>X’0235’</td>
<td>Symmetric Key Import - DES_PKA92 KEK</td>
</tr>
<tr>
<td>DATA, MAC,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MACVER,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYGENKY,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPORTER, or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OKEYXLAT key</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKCSOAEP or</td>
<td>AES</td>
<td>X’012F’</td>
<td>Symmetric Key Import - AES_PKCSOAEP_PKCS-1.2</td>
</tr>
<tr>
<td>PKCS-1.2</td>
<td>DES</td>
<td>X’0106’</td>
<td>Symmetric Key Import - AES_PKCS-1.2</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>AES</td>
<td>X’012F’</td>
<td>Symmetric Key Import - AES_ZERO-PAD</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>X’023D’</td>
<td>Symmetric Key Import - AES_ZERO-PAD</td>
</tr>
</tbody>
</table>

The use of the **WRAP-ECB** or **WRAP-ENH** key-wrapping method keywords when the default key-wrapping method setting does not match the keyword, requires the Symmetric Key Import - Allow wrapping override keywords command (offset X’0144’) to be enabled.

**Usage notes**
The usage notes for CSNDSYI.
Symmetric Key Import (CSNDSYI)

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

Use of PKA92 with an input NOCV key-encrypting key token is not supported.

During initialization of a CEX*C, an Environment Identifier (EID) of zero will be set in the coprocessor. This will be interpreted by the Symmetric Key Import verb to mean that environment identification checking is to be bypassed. Thus it is possible on a Linux on IBM System z system for a key-encrypting key RSA-enciphered at a node (EID) to be imported at the same node.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDSYIJ. See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDSYIJ are shown here.

Format

```java
public native void CSNDSYIJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger RSA_enciphered_key_length,
    byte[] RSA_enciphered_key,
    hikmNativeInteger RSA_private_key_identifier_length,
    byte[] RSA_private_key_identifier,
    hikmNativeInteger target_key_identifier_length,
    byte[] target_key_identifier);
```
Symmetric Key Import2 (CSNDSYI2)

Use the Symmetric Key Import2 verb to import an HMAC or AES key enciphered under an RSA public key or AES EXPORTER key.

It returns the key in operational form, enciphered under the master key.

This verb returns a variable-length CCA key token and uses the AESKW wrapping method.

Format
The format of CSNDSYI.

```
CSNDSYI2(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    enciphered_key_length,
    enciphered_key,
    transport_key_identifier_length,
    transport_key_identifier,
    key_name_length,
    key_name,
    target_key_identifier_length,
    target_key_identifier)
```

Parameters
The parameters for CSNDSYI.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`
Direction: Input
Type: Integer
The number of keywords you supplied in the `rule_array` parameter. This value must be 2.

`rule_array`
Direction: Input
Type: String
The keywords that provide control information to the verb. The following table provides a list. The recovery method is the method to use to recover the symmetric key. The keywords must be 8 bytes of contiguous storage with the keyword left-aligned in its 8-byte location and padded on the right with blanks. The `rule_array` keywords are described in Table 65.

Table 65. Keywords for Symmetric Key Import2 control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>(One, required)</td>
</tr>
<tr>
<td>AES</td>
<td>The key being imported is an AES key.</td>
</tr>
</tbody>
</table>
Table 65. Keywords for Symmetric Key Import2 control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMAC</td>
<td>The key being imported is an HMAC key. Only the PKOAEP2 recovery method is supported.</td>
</tr>
<tr>
<td>Recovery method (One, required)</td>
<td></td>
</tr>
<tr>
<td>AESKW</td>
<td>Specifies the enciphered key has been wrapped with the AESKW formatting method.</td>
</tr>
<tr>
<td>PKOAEP2</td>
<td>Specifies to format the key according to the method found in RSA DSI PKCS #1 v2.1 RSAES-OAEP documentation.</td>
</tr>
</tbody>
</table>

**Note:** There is no need for a hash method keyword, because the hash method is encoded in the external key-token carrying the encoded and encrypted payload.

**enciphered_key_length**

**Direction:** Input  
**Type:** Integer  

The length of the `enciphered_key` parameter. The maximum size is 900 bytes.

**enciphered_key**

**Direction:** Input  
**Type:** String  

The key to import, protected under either an RSA public key or an AES KEK. If the recovery method is PKOAEP2, the encrypted key is in the low-order bits (right-aligned) of a string whose length is the minimum number of bytes that can contain the encrypted key. If the recovery method is AESKW, the encrypted key is an AES key or HMAC key in the external variable length key token.

**transport_key_identifier_length**

**Direction:** Input  
**Type:** Integer  

The length of the `transport_key_identifier` parameter. When the `transport_key_identifier` parameter is a key label, this field must be 64. The maximum size is 3500 bytes for an RSA private key, or 725 bytes for an AES IMPORTER KEK.

**transport_key_identifier**

**Direction:** Input  
**Type:** String  

An internal RSA private key token, internal AES IMPORTER KEK, or the 64-byte label of a key token whose corresponding key protects the symmetric key.

When the AESKW key formatting method is specified, this parameter must be an AES IMPORTER key with the IMPORT bit on in the key-usage field. Otherwise, this parameter must be an RSA private key.

**key_name_length**

**Direction:** Input  
**Type:** Integer  

The length of the `key_name` parameter for `target_key_identifier`. Valid values are 0 and 64.
Symmetric Key Import2 (CSNDSYI2)

**key_name**
- **Direction:** Input
- **Type:** String

A 64-byte key store label to be stored in the associated data structure of target_key_identifier.

**target_key_identifier_length**
- **Direction:** Input/Output
- **Type:** Integer

On input, the length in bytes of the buffer for the target_key_identifier parameter. The buffer must be large enough to receive the target key token. The maximum value is 725 bytes.

On output, the parameter will hold the actual length of the target key token.

**target_key_identifier**
- **Direction:** Output
- **Type:** String

This parameter contains the internal token of the imported symmetric key.

**Restrictions**

The restrictions for CSNDSYI.

The exponent of the RSA public key must be odd.

**Required commands**

The required commands for CSNDSYI.

The Symmetric Key Import2 verb requires the following commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Key-formatting method keyword</th>
<th>Token algorithm keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESKW (Rel. 4.2 or later)</td>
<td>AES</td>
<td>X’0329’</td>
<td>Import AES Key (AESKW)</td>
</tr>
<tr>
<td>PKOAEP2</td>
<td>AES (Rel. 4.2 or later)</td>
<td>X’00FD’</td>
<td>Import AES Key (PKOAEP2)</td>
</tr>
<tr>
<td>PKOAEP2</td>
<td>HMAC</td>
<td>X’00F4’</td>
<td>Import HMAC Key (PKOAEP2)</td>
</tr>
</tbody>
</table>

To disallow the wrapping of a stronger key with a weaker key, the Disallow Weak Import command (offset X’032B’) must be enabled in the active role. This command affects multiple verbs. See Appendix G, “Access control points and verbs,” on page 723.

To receive a warning against the wrapping of a stronger key with a weaker key, the Warn When Wrapping Weak Keys command (offset X’032C’) must be enabled in the active role. The Disallow Weak Import command overrides this command.

**Usage notes**

The usage notes for CSNDSYI.
This is the message layout used to encode the key material exported with the PKOAEP2 formatting method.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash field</td>
<td>32 bytes</td>
<td>SHA-256 hash of associated data section in the source key identifier</td>
</tr>
<tr>
<td>Key bit length</td>
<td>2 bytes</td>
<td>Variable</td>
</tr>
<tr>
<td>Key material</td>
<td>length in bytes of the key material (rounded up to the nearest byte)</td>
<td>Variable</td>
</tr>
</tbody>
</table>

**Hash field**

The associated data for the HMAC variable length token is hashed using SHA-256.

**Key bit length**

A 2-byte key bit length field.

**Key material**

The key material is padded to the nearest byte with '0' bits.

The hardware configuration sets the limit on the modulus size of keys for key management; thus, this verb will fail if the RSA key modulus bit length exceeds this limit.

Specification of PKA92 with an input NOCV key-encrypting key token is not supported.

During initialization of a CEX*C, an Environment Identifier (EID) of zero will be set in the coprocessor. This will be interpreted by the Symmetric Key Import2 verb to mean that environment identification checking is to be bypassed. Thus it is possible on a Linux on IBM System z system for a key-encrypting key RSA-enciphered at a node (EID) to be imported at the same node.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNDSYI2J.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDSYI2J are shown here.

**Format**

```java
public native void CSNDSYI2J(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger enciphered_key_length,
    byte[] enciphered_key,
    hikmNativeInteger transport_key_identifier_length,
    byte[] transport_key_identifier,
    hikmNativeInteger key_name_length,
    byte[] key_name,
    hikmNativeInteger target_key_identifier_length,
    byte[] target_key_identifier);
```
Symmetric Key Import2 (CSNDSYI2)
Chapter 7. Protecting data

Use CCA to protect sensitive data stored on your system, sent between systems, or stored off your system on magnetic tape.

To protect data, encipher it under a key. When you want to read the data, decipher it from ciphertext to plaintext form.

CCA provides Encipher and Decipher verbs to perform these functions. If you use a key to encipher data, you must use the same key to decipher the data. The Encipher and Decipher verbs use encrypted keys as input. You can also use clear keys, indirectly, by first using the Clear Key Import verb and then using the Encipher and Decipher verbs.

This chapter describes the following verbs used for protecting data using DES or AES:

- “Decipher (CSNBDEC)” on page 288
- “Encipher (CSNBENC)” on page 293
- “Symmetric Algorithm Decipher (CSNBSAD)” on page 298
- “Symmetric Algorithm Encipher (CSNBSAE)” on page 305

Modes of operation

Different algorithms are used to encipher or decipher DES data or keys and AES data or keys.

To encipher or decipher DES data or keys, CCA uses the U.S. National Institute of Standards and Technology (NIST) Data Encryption Standard (DES) algorithm, with single-length, double-length, or triple-length keys.

To encipher or decipher AES data or keys, CCA uses the U.S. National Institute of Standards and Technology (NIST) Advanced Encryption Standard (AES) algorithm, with 16-byte, 24-byte or 32-byte keys.

The Encipher and Decipher verbs operate in DES CBC (Cipher Block Chaining) mode.

Cipher Block Chaining (CBC) mode

The CBC mode uses an initial chaining vector (ICV) in its processing.

The CBC mode processes blocks of data only in exact multiples of the blocksize. The ICV is exclusive ORed with the first block of plaintext prior to the encryption step. The block of ciphertext just produced is exclusive-ORed with the next block of plaintext, and so on. You must use the same ICV to decipher the data. This disguises any pattern that may exist in the plaintext. CBC mode is the default for encrypting and decrypting data using the Encipher and Decipher verbs. “Ciphering methods” on page 702 describes the cipher processing rules in detail.

Electronic Code Book (ECB) mode

In ECB mode, each block of plaintext is separately enciphered and each block of the ciphertext is separately deciphered.
In other words, the encipherment or decipherment of a block is totally independent of other blocks.

**Processing rules**

This topic discusses the different types of processing rules you can use for block chaining.

“Ciphering methods” on page 702 describes the cipher processing rules in detail.

CCA handles chaining for each block of data, from the first block until the last complete block of data in each Encipher or Symmetric Algorithm Encipher call. There are different types of processing rules you can choose for block chaining:

**ANSI X9.23**
Data is not necessarily in exact multiples of the block size. This processing rule pads the plaintext so the ciphertext produced is in exact multiples of the block size.

**Cipher block chaining (CBC)**
Data must be an exact multiple of the block size, and output will have the same length.

**Cryptographic Unit Support Program (CUSP)**
CBC mode (cipher block chaining) that is compatible with IBM’s CUSP and PCF products. The data need not be in exact multiples of the block size. The ciphertext is the same length as the plaintext.

**Electronic Code Book (ECB)**
The data length must be a multiple of the block size. See “Electronic Code Book (ECB) mode” on page 285.

**Information Protection System (IPS)**
CBC mode that is compatible with IBM’s IPS product. The data need not be in exact multiples of the block size. The ciphertext is the same length as the plaintext.

**PKCS-PAD**
The data is padded on the right with between one and 16 bytes of pad characters, making ciphertext a multiple of the block size.

The resulting chaining value (except for ECB mode), after an Encipher or Symmetric Algorithm Encipher call, is known as an output chaining vector (OCV). When there are multiple cipher requests, the application can pass the OCV from the previous Encipher or Symmetric Algorithm Encipher call, as the input chaining vector (ICV) in the next Encipher or Symmetric Algorithm Encipher call. This produces chaining between successive calls, which is known as record chaining. CCA provides the ICV selection keyword CONTINUE in the rule_array parameter used to select record chaining with the CBC processing rule.

**Triple-DES encryption**

Triple-DES encryption uses a triple-length DATA key comprised of three 8-byte DES keys to encipher eight bytes of data.

To encipher the data it uses the following method:
- Encipher the data using the first key
- Decipher the result using the second key
- Encipher the second result using the third key
The procedure is reversed to decipher data that has been triple-DES enciphered:
- Decipher the data using the third key
- Encipher the result using the second key
- Decipher the second result using the first key

A variation of the triple-DES algorithm supports the use of a double-length DATA key comprised of two 8-byte DATA keys. In this method, the first 8-byte key is reused in the last encipherment step.

Due to export regulations, triple-DES encryption might not be available on your processor.
Decipher (CSNBDEC)

Use the Decipher verb to decipher data using the DES cipher block chaining mode.

CCA supports the following processing rules to decipher data. You choose the type of processing rule that the Decipher verb should use for block chaining.

**Processing Rule**

<table>
<thead>
<tr>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI X9.23</td>
</tr>
<tr>
<td>Cipher Block Chaining (CBC)</td>
</tr>
<tr>
<td>Cryptographic Unit Support Program (CUSP)</td>
</tr>
<tr>
<td>Information Protection System (IPS)</td>
</tr>
</tbody>
</table>

- **ANSI X9.23**
  - For cipher block chaining. The ciphertext must be an exact multiple of eight bytes, but the plaintext will be between 1 and 8 bytes shorter than the ciphertext. The `text_length` will also be reduced to show the original length of the plaintext.

- **Cipher Block Chaining (CBC)**
  - The ciphertext must be an exact multiple of eight bytes and the plaintext will have the same length.

- **Cryptographic Unit Support Program (CUSP)**
  - CBC mode (cipher block chaining) that is compatible with IBM’s CUSP and PCF products. The data need not be in exact multiples of eight bytes. The ciphertext is the same length as the plaintext.

- **Information Protection System (IPS)**
  - CBC mode (cipher block chaining) that is compatible with IBM’s IPS product. The data need not be in exact multiples of eight bytes. The ciphertext is the same length as the plaintext.

The cipher block chaining (CBC) mode uses an initial chaining value (ICV) in its processing. The first eight bytes of ciphertext is deciphered and then the ICV is XORed with the resulting eight bytes of data to form the first 8-byte block of plaintext. Thereafter, the 8-byte block of ciphertext is deciphered and XORed with the previous 8-byte block of ciphertext until all the ciphertext is deciphered.

The selection between single-DES decryption mode and triple-DES decryption mode is controlled by the length of the key supplied in the `key_identifier` parameter. If a single-length key is supplied, single-DES decryption is performed. If a double-length or triple-length key is supplied, triple-DES decryption is performed.

A different ICV could be passed on each call to the Decipher verb. However, the same ICV that was used in the corresponding Encipher verb must be passed.

Short blocks are text lengths of between one and seven bytes. A short block can be the only block. Trailing short blocks are blocks of between one and seven bytes that follow an exact multiple of eight bytes. For example, if the text length is 21, there are two 8-byte blocks and a trailing short block of five bytes. Because the DES processes text only in exact multiples of eight bytes, some special processing is required to decipher such short blocks.

These methods of treating short blocks and trailing short blocks do not increase the length of the ciphertext compared to the length of the plaintext. If the plaintext was padded during encipherment, the length of the ciphertext will always be an exact multiple of eight bytes.

CCA supports the ANSI X9.23 padding method.
**Host CPU acceleration: CPACF**

Only keys with a key type of DATA can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a DATA key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see “CPACF support” on page 12.

**Format**

The format of CSNBDEC.

```c
CSNBDEC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_identifier,
    text_length,
    cipher_text,
    initialization_vector,
    rule_array_count,
    rule_array,
    chaining_vector,
    clear_text)
```

**Parameters**

The parameters for CSNBDEC.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**key_identifier**

**Direction:** Input/Output  
**Type:** String

A 64-byte string that is the internal key token containing the data-encrypting key or the label of a DES key storage record containing a data-encrypting key to be used for deciphering the data. If the key token or key label contains a single-length key, single-DES decryption is performed. If the key token or key label contains a double-length or triple-length key, triple-DES decryption is performed.

Double length CIPHER and DECIPHER keys are also supported.

**text_length**

**Direction:** Input/Output  
**Type:** Integer

On entry, you supply the length of the ciphertext. The maximum length of text is 214,783,647 bytes. A zero value for the `text_length` parameter is not valid. If the returned deciphered text (`clear_text` parameter) is a different length because of the removal of padding bytes, the value is updated to the length of the plaintext.

The application program passes the length of the ciphertext to the verb. The verb returns the length of the plaintext to your application program.
Decipher (CSNBDEC)

cipher_text
Direction: Input
Type: String
The text to be deciphered.

initialization_vector
Direction: Input
Type: String
The 8-byte supplied string for the cipher block chaining. The first block of the
ciphertext is deciphered and XORed with the initial chaining vector (ICV) to
get the first block of cleartext. The input block is the next ICV. To decipher the
data, you must use the same ICV used when you enciphered the data.

rule_array_count
Direction: Input
Type: Integer
A pointer to an integer variable containing the number of elements in the
rule_array variable. This value must be 1, 2, or 3.

rule_array
Direction: Input
Type: String
An array of 8-byte keywords providing the processing control information. The
array is positional. The first keyword in the array is the processing rule. You
choose the processing rule you want the verb to use for deciphering the data.
The second keyword is the ICV selection keyword. The third keyword (or the
second if the ICV selection keyword is allowed to default) is the encryption
algorithm to use. The rule_array keywords are described in Table 67.

Table 67. Keywords for Decipher control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Rule (One, required)</td>
<td></td>
</tr>
<tr>
<td>CBC</td>
<td>Performs ANSI X3.102 cipher block chaining. The data must be a multiple of eight bytes. An OCV is produced and placed in the chaining_vector parameter. If the ICV selection keyword CONTINUE is specified, the CBC OCV from the previous call is used as the ICV for this call.</td>
</tr>
<tr>
<td>CUSP</td>
<td>Performs Cryptographic Unit Support Program (CUSP) cipher block chaining.</td>
</tr>
<tr>
<td>IPS</td>
<td>Performs Information Protection System (IPS) cipher block chaining.</td>
</tr>
<tr>
<td>X9.23</td>
<td>Deciphers with cipher block chaining and text length reduced to the original value. This is compatible with the requirements in ANSI standard X9.23. The ciphertext length must be an exact multiple of eight bytes. Padding is removed from the plaintext.</td>
</tr>
<tr>
<td>ICV Selection (One, optional)</td>
<td></td>
</tr>
<tr>
<td>CONTINUE</td>
<td>This specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the chaining_vector parameter points. CONTINUE is valid only for the CBC processing rule.</td>
</tr>
<tr>
<td>INITIAL</td>
<td>This specifies taking the initialization vector from the initialization_vector parameter. INITIAL is the default value.</td>
</tr>
<tr>
<td>Encryption algorithm (Optional)</td>
<td></td>
</tr>
<tr>
<td>DES</td>
<td>This specifies using the data encryption standard and ignoring the token marking.</td>
</tr>
</tbody>
</table>
“Ciphering methods” on page 702 describes the cipher processing rules in detail.

**chaining_vector**

**Direction:** Input/Output  
**Type:** String

An 18-byte field CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector holds the output chaining vector (OCV) from the caller. The OCV is the first eight bytes in the 18-byte string.

The direction is Output if the ICV selection keyword of the **rule_array** parameter is **INITIAL**. The direction is Input/Output if the ICV selection keyword of the **rule_array** parameter is **CONTINUE**.

**clear_text**

**Direction:** Output  
**Type:** String

The field where the verb returns the deciphered text.

**Restrictions**

The restrictions for CSNBDEC.

This verb will fail if the key token contains double or triple-length keys and triple-DES is not enabled.

**Required commands**

The required commands for CSNBDEC.

This verb requires the Decipher - DES command (offset X'000F') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBDEC.

None

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBDECJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBDECJ are shown here.

**Format**

```java
public native void CSNBDECJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_identifier,
    hikmNativeInteger text_length,
    byte[] ciphertext,
    byte[] initialization_vector,
```

Chapter 7. Protecting data 291
Decipher (CSNBDEC)

```c
hikmNativeInteger rule_array_count,
byte[] rule_array,
byte[] chaining_vector,
byte[] plaintext);
```
Encipher (CSNBENC)

Use the Encipher verb to encipher data using the DES cipher block chaining mode.

CCA supports the following processing rules to encipher data. You choose the type of processing rule that the Encipher verb should use for the block chaining.

**Processing Rule**

**Purpose**

**Cipher block chaining (CBC)**

In exact multiples of eight bytes.

**Cryptographic Unit Support Program (CUSP)**

CBC mode (cipher block chaining) that is compatible with IBM’s CUSP and PCF products. The data need not be in exact multiples of eight bytes. The ciphertext is the same length as the plaintext.

**Information Protection System (IPS)**

CBC mode (cipher block chaining) that is compatible with IBM’s IPS product. The data need not be in exact multiples of eight bytes. The ciphertext is the same length as the plaintext.

**ANSI X9.23**

For block chaining not necessarily in exact multiples of eight bytes. This process rule pads the plaintext so that ciphertext produced is an exact multiple of eight bytes.

For more information about the processing rules, see Table 68 on page 295 and “Ciphering methods” on page 702.

The cipher block chaining (CBC) mode of operation uses an initial chaining vector (ICV) in its processing. The ICV is XORed with the first eight bytes of plaintext before the encryption step and thereafter, the 8-byte block of ciphertext just produced is XORed with the next 8-byte block of plaintext and so on. This disguises any pattern that might exist in the plaintext.

The selection between single-DES encryption mode and triple-DES encryption mode is controlled by the length of the key supplied in the `key_identifier` parameter. If a single-length key is supplied, single-DES encryption is performed. If a double-length or triple-length key is supplied, triple-DES encryption is performed.

To nullify the CBC effect on the first 8-byte block, supply eight bytes of zero. However, the ICV might require zeros.

Cipher block chaining also produces a resulting chaining value called the output chaining vector (OCV). The application can pass the OCV as the ICV in the next encipher call. This results in record chaining.

Note that the OCV that results is the same, whether an Encipher or a Decipher verb was invoked, assuming the same text, ICV, and key were used.

Short blocks are text lengths of between one and seven bytes. A short block can be the only block. Trailing short blocks are blocks of between one and seven bytes that follow an exact multiple of eight bytes. For example, if the text length is 21, there are two 8-byte blocks, and a trailing short block of five bytes.
An alternative method is to pad the plaintext and produce a ciphertext that is longer than the plaintext. The plaintext can be padded with up to eight bytes using one of several padding methods. This padding produces a ciphertext that is an exact multiple of eight bytes in length.

If the cleartext is already a multiple of eight, the ciphertext can be created using any processing rule.

Because of padding, the returned ciphertext length is longer than the provided plaintext; the `text_length` parameter will have been modified. The returned ciphertext field should be eight bytes longer than the length of the plaintext to accommodate the maximum amount of padding.

**Attention:** If you lose the data-encrypting key under which the data (plaintext) is enciphered, the data enciphered under that key (ciphertext) cannot be recovered.

**Host CPU acceleration: CPACF**

Only keys with a key type of DATA can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a DATA key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see “CPACF support” on page 12.

**Format**

The format of CSNBENC.

```
csnbenc(   return_code,   reason_code,   exit_data_length,   exit_data,   key_identifier,   text_length,   clear_text,   initialization_vector,   rule_array_count,   rule_array,   pad_character,   chaining_vector,   cipher_text)   
```

**Parameters**

The parameters for CSNBENC.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**key_identifier**

- **Direction:** Input/Output
- **Type:** String

A 64-byte string that is the internal key token containing the data-encrypting key or the label of a DES key storage record containing the data-encrypting
Encipher (CSNBENC)

key, to be used for encrypting the data. If the key token or key label contains a single-length key, single-DES encryption is performed. If the key token or key label contains a double-length or triple-length key, triple-DES encryption is performed.

Single and double-length CIPHER and ENCRYPTER keys are also supported.

text_length

 Direction: Input/Output
 Type: Integer

On entry, the length of the plaintext (clear_text parameter) you supply. The maximum length of text is 214,783,647 bytes. A zero value for the text_length parameter is not valid. If the returned enciphered text (cipher_text parameter) is a different length because of the addition of padding bytes, the value is updated to the length of the ciphertext.

The application program passes the length of the plaintext to the verb. This verb returns the length of the ciphertext to the application program.

clear_text

 Direction: Input
 Type: String

The text that is to be enciphered.

initialization_vector

 Direction: Input
 Type: String

The 8-byte supplied string for the cipher block chaining. The first eight bytes (or less) block of the data is XORed with the ICV and then enciphered. The input block is enciphered and the next ICV is created. You must use the same ICV to decipher the data.

rule_array_count

 Direction: Input
 Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1, 2, or 3.

rule_array

 Direction: Input
 Type: String

An array of 8-byte keywords providing the processing control information. The array is positional. The first keyword in the array is the processing rule. You choose the processing rule you want the verb to use for enciphering the data. The second keyword is the ICV selection keyword. The third keyword (or the second if the ICV selection keyword is allowed to default to INITIAL) is the encryption algorithm to use. The rule_array keywords are described in Table 68.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Rule</td>
<td>(One, required)</td>
</tr>
</tbody>
</table>

Table 68. Keywords for Encipher control information
Encipher (CSNBENC)

Table 68. Keywords for Encipher control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CBC</em></td>
<td>Performs ANSI X3.102 cipher block chaining. The data must be a multiple of eight bytes. An OCV is produced and placed in the chaining_vector parameter. If the ICV selection keyword CONTINUE is specified, the CBC OCV from the previous call is used as the ICV for this call.</td>
</tr>
<tr>
<td><em>CUSP</em></td>
<td>Performs Cryptographic Unit Support Program (CUSP) cipher block chaining.</td>
</tr>
<tr>
<td><em>IPS</em></td>
<td>Performs Information Protection System (IPS) cipher block chaining.</td>
</tr>
<tr>
<td><em>X9.23</em></td>
<td>Performs cipher block chaining with 1 - 8 bytes of padding. This is compatible with the requirements in ANSI X9.23. If the data is not in exact multiples of eight bytes, X9.23 pads the plaintext so the ciphertext produced is an exact multiple of eight bytes. The plaintext is padded to the next multiple eight bytes, even if this adds eight bytes. An OCV is produced.</td>
</tr>
</tbody>
</table>

**ICV Selection** (One, optional)

| CONTINUE | This specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the chaining_vector parameter points. CONTINUE is valid only for the CBC processing rule. |
| INITIAL | This specifies taking the initialization vector from the initialization_vector parameter. INITIAL is the default value. |

**Encryption Algorithm** (Optional)

| DES | This specifies using the data encryption standard and ignoring the token marking. |

"Ciphering methods" on page 702 describes the cipher processing rules in detail.

**pad_character**

Direction: Input

Type: Integer

An integer, 0 - 255, that is used as a padding character for the X9.23 process rule (rule_array parameter).

**chaining_vector**

Direction: Input/Output

Type: String

An 18-byte field CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector holds the output chaining vector (OCV) from the caller. The OCV is the first eight bytes in the 18-byte string.

The direction is Output if the ICV selection keyword of the rule_array parameter is INITIAL.

The direction is Input/Output if the ICV selection keyword of the rule_array parameter is CONTINUE.

**cipher_text**

Direction: Output

Type: String

The enciphered text the verb returns. The length of the ciphertext is returned in the text_length parameter. The cipher_text could be eight bytes longer than the length of the clear_text field because of the padding that is required for some processing rules.
Restrictions

The restrictions for CSNBENC.

This verb will fail if the key token contains double-length or triple-length keys and triple-DES is not enabled.

Required commands

The required commands for CSNBENC.

This verb requires the Encipher - DES command (offset X'000E') to be enabled in the active role.

Usage notes

The usage notes for CSNBENC.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBENCJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBENCJ are shown here.

Format

```java
public native void CSNBENCJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_identifier,
    hikmNativeInteger text_length,
    byte[] plaintext,
    byte[] initialization_vector,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger pad_character,
    byte[] chaining_vector,
    byte[] ciphertext);
```
Symmetric Algorithm Decipher (CSNBSAD)

Symmetric Algorithm Decipher (CSNBSAD)

Use the Symmetric Algorithm Decipher verb to decipher data with an AES algorithm.

CCA supports the following processing rules to decipher data. You choose the type of processing rule that the verb should use for block chaining.

**Cipher Block Chaining (CBC)**
- The plaintext must be an exact multiple of eight bytes, and the ciphertext will have the same length.

**Electronic Code Book (ECB)**
- The plaintext length must be a multiple of the block size.

**PKCS-PAD**
- The plaintext was padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the block size.

The AES key used to decipher the data can either be 16, 24, or 32 bytes (128, 192, or 256 bits) in length. The key can be supplied to the verb in any of three forms:
1. A cleartext key consisting of only the key bytes, not contained in a key token.
2. A cleartext key contained in an internal fixed or variable length AES key-token.
3. An encrypted key contained in an internal fixed or variable length AES key-token, where the key is wrapped (encrypted) with the AES master key.

To use this verb, specify:
- The `rule_array`:
  1. The algorithm identifier keyword `AES`, which is the only symmetric algorithm currently supported.
  2. An optional processing rule using keyword `CBC` (the default), `ECB`, or `PKCS-PAD`, which selects the decryption mode.
  3. An optional key rule using the keyword `KEY-CLR` (the default) or `KEYIDENT`, which selects whether the `key_identifier` parameter points to a 16-byte, 24-byte, or 32-byte clear key, or a key contained in a 64-byte AES key-token, either in application storage or a key label of such a key in key storage.
  4. An optional initial chaining value (ICV) selection using the keyword `INITIAL` (the default) or `CONTINUE`, which indicates whether it is the first or a subsequent request, and which parameter points to the initialization vector.

- For a key rule of `KEY-CLR`, a key identifier containing a 16-byte, 24-byte, or 32-byte clear key. For a key rule of `KEYIDENT`, a fixed-length or variable-length internal AES key-token or the key label of such a key in AES key-storage. The key token can contain either a clear or enciphered key.

A variable-length AES key-token must have a key that can be used for decryption (key-usage field 1 high-order byte is B’x1xx xxxx’). Also, for processing rule `CBC` or `PKCS-PAD`, key usage must allow the key to be used for Cipher Block Chaining (KUF2 high-order byte is X’00’). For processing rule `ECB`, key usage must allow the key to be used for Electronic Code Book (KUF2 high-order byte is X’01’).

- A block size of 16 for the cryptographic algorithm.
- For cipher block chaining, either one of these:
Symmetric Algorithm Decipher (CSNBSAD)

1. For an ICV selection of INITIAL, a 16-byte initialization vector of your choosing and a 32-byte chain data buffer.
2. For an ICV selection of CONTINUE, no initialization vector and the 32-byte chain data buffer from the output of the previous chained call. The electronic code book algorithm does not use an initialization vector or a chain data buffer.

- The ciphertext to be deciphered.
- A cleartext buffer large enough to receive the deciphered output.

This verb does the following when it decipheres the data:
1. Verifies the AES key-token for keyword KEYIDENT.
2. Verifies that the ciphertext length is a multiple of the block size.
3. Deciphers the input AES key if the key is encrypted (MKVP was present in token).
4. Deciphers the ciphertext with the AES clear key according to the encryption mode specified.
5. Removes from 1 - 16 pad characters from the right of the clear data for keyword PKCS-PAD.
6. Returns the cleartext data and its length.
7. Returns the chain data and its length if keyword ECB is not specified.

Only keys with a key type of DATA can be used successfully with the CPACF exploitation layer through this verb. Specifically, a DATA key has a Control Vector (CV) of all X'00' bytes for all active bytes of the CV (eight bytes for all AES keys). Note that as of CCA Release 3.30 (the first release of AES function on the CEX2C 4764 adapter) to Release 4.1.0 (the most recent release of CCA on the CEX*C feature), AES keys were only available as DATA keys. For details about CPACF, see "CPACF support" on page 12.

Format

The format of CSNBSAD.

```c
CSNBSAD(    return_code,    reason_code,    exit_data_length,    exit_data,    rule_array_count,    rule_array,    key_identifier_length,    key_identifier,    key_parms_length,    key_parms,    block_size,    initialization_vector_length,    initialization_vector,    chain_data_length,    chain_data,    ciphertext_length,    cipher_text,    cleartext_length,    cleartext,    optional_data_length,    optional_data)
```
Symmetric Algorithm Decipher (CSNBSAD)

Parameters

The parameters for CSNBSAD.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1, 2, 3, or 4.

rule_array

- **Direction:** Input
- **Type:** Array

An array of 8-byte keywords providing the processing control information. The keywords must be left-aligned and padded on the right with space characters. The rule_array keywords are described in Table 69.

Table 69. Keywords for Symmetric Algorithm Decipher control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decryption algorithm</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Specifies use of the Advanced Encryption Standard (AES) as the deciphering algorithm. The block size for AES is 16 bytes, and the key length is 16, 24, or 32 bytes. AES is the only algorithm currently supported by this verb.</td>
</tr>
<tr>
<td><strong>Processing rule</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>CBC</td>
<td>Performs ANSI X3.102 cipher block chaining. The data must be a multiple of eight bytes. An OCV is produced and placed in the chaining_vector parameter. If the ICV selection keyword CONTINUE is specified, the CBC OCV from the previous call is used as the ICV for this call.</td>
</tr>
<tr>
<td>ECB</td>
<td>Specifies deciphering in Electronic Code Book mode. The ciphertext length must be a multiple of the block size.</td>
</tr>
</tbody>
</table>
| PKCS-PAD         | Specifies that the cleartext was padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the block size, before the data was enciphered. Each pad character is valued to the number of pad characters added. 

The output cleartext is stripped of any pad characters and the cleartext length is 1 - 16 bytes less than the ciphertext length. |
| **Key rule** (One, optional)                                                 |
| KEY-CLR          | Specifies that the key_identifier parameter points to a cleartext AES key. Only the key value is allowed; the key is not contained in a key token. This is the default value. |
| KEYIDENT         | Specifies that the key_identifier parameter points to an internal AES key-token or the label of an internal key-token in AES key-storage. |
| **ICV selection** (One, optional)                                            |
| CONTINUE          | This specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the chaining_vector parameter points. CONTINUE is valid only for the CBC processing rule. |
| INITIAL           | This specifies taking the initialization vector from the initialization_vector parameter. INITIAL is the default value. |

"Ciphering methods" on page 702 describes the cipher processing rules in detail.
Symmetric Algorithm Decipher (CSNBSAD)

**key_identifier_length**
- **Direction**: Input
- **Type**: Integer

A pointer to an integer variable containing the number of bytes of data in the `key_identifier` variable. This value must be 16, 24, 32, or \( \geq 64 \).

**key_identifier**
- **Direction**: Input
- **Type**: String

A pointer to a string variable containing either a cleartext AES key or the internal key-token or a label for an internal key-token record in AES key-storage. This is the key used to decipher the data pointed to by the `ciphertext` parameter.

For `rule_array` keyword **KEY-CLR**, a 16-byte, 24-byte, or 32-byte clear AES key is required. For `rule_array` keyword **KEYIDENT**, a 64-byte internal key-token or key label for an internal key-token record in AES key-storage is required.

**key_parms_length**
- **Direction**: Input
- **Type**: Integer

A pointer to an integer variable containing the number of bytes of data in the `key_parms` parameter. This value must be 0.

**key_parms**
- **Direction**: Input
- **Type**: String

A pointer to a string variable for key-related parameters. It is currently unused.

**block_size**
- **Direction**: Input
- **Type**: Integer

A pointer to an integer variable containing the block size used by the cryptographic algorithm. This value must be 16.

**initialization_vector_length**
- **Direction**: Input
- **Type**: Integer

A pointer to an integer variable containing the number of bytes of data in the `initialization_vector` variable. For cipher block chaining (CBC or PKCS-PAD) with an **INITIAL** ICV selection, this value must be 16. For processing rule **ECB** or ICV selection **CONTINUE**, this value should be 0.

**initialization_vector**
- **Direction**: Input
- **Type**: String

A pointer to a string variable containing the initialization vector for the **INITIAL** call to CBC mode decryption. It is not used if the process rule is **ECB** or the ICV selection is **CONTINUE**. The same initialization vector must be used when deciphering the data.

**chain_data_length**
Symmetric Algorithm Decipher (CSNBSAD)

**chain_data**

**Direction:** Input/Output  
**Type:** String

A pointer to a string variable used as a work area for CBC decipher requests. This work area is not used for ECB mode decryption. When the verb performs a CBC decipher operation and the ICV selection is INITIAL, the `chain_data` variable is an output-only buffer that receives data used as input for deciphering the next part of the input data, if any. When the ICV selection is CONTINUE, the `chain_data` variable is both an input and output buffer. The application must not change any intermediate data in this string.

**ciphertext_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the `ciphertext` variable. The `ciphertext_length` value must be a multiple of the block size. This value must not be 0. If PKCS-PAD is specified, the output `cleartext_length` variable will be 1 - 16 bytes less than the `ciphertext_length` value.

**ciphertext**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing the data to be deciphered, including any pad bytes.

**cleartext_length**

**Direction:** Input/Output  
**Type:** Integer

On input, this parameter is a pointer to an integer variable containing the number of bytes of data in the `cleartext` variable. On output, this variable is updated to contain the actual length of text output in the `cleartext` variable. If PKCS-PAD is specified, the `cleartext` value is updated with 1 - 16 bytes of data less than the `ciphertext_length` value.

**cleartext**

**Direction:** Input/Output  
**Type:** String

A pointer to a string variable used to contain the data to be deciphered, excluding any pad bytes.

**optional_data_length**
Symmetric Algorithm Decipher (CSNBSAD)

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the `optional_data` variable. This value should be 0.

**optional_data**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing optional data for the decryption. It is currently not used.

**Restrictions**

The restrictions for CSNBSAD.

None.

**Required commands**

The required commands for CSNBSAD.

This verb requires the Symmetric Algorithm Decipher - secure AES keys command (offset X'012B') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBSAD.

None

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBSADJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBSADJ are shown here.

**Format**

```java
public native void CSNBSADJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_length,
    byte[] key_identifier,
    hikmNativeInteger key_parms_length,
    byte[] key_parms,
    hikmNativeInteger block_size,
    hikmNativeInteger iv_length,
    byte[] iv,
    hikmNativeInteger chain_data_length,
    byte[] chain_data,
    hikmNativeInteger cipher_text_length,
    byte[] cipher_text,
    hikmNativeInteger clear_text_length,
)
```
Symmetric Algorithm Decipher (CSNBSAD)

byte[] clear_text,
hikmNativeInteger optional_data_length,
byte[] optional_data;
Symmetric Algorithm Encipher (CSNBSAE)

Use the Symmetric Algorithm Encipher verb to encipher data using the AES algorithm.

CCA supports the following processing rules to encipher data. You choose the type of processing rule that the verb should use for block chaining.

**Cipher Block Chaining (CBC)**

The plaintext must be an exact multiple of eight bytes, and the ciphertext will have the same length.

**Electronic Code Book (ECB)**

The plaintext length must be a multiple of the block size.

**PKCS-PAD**

The plaintext was padded on the right with 1 - 16 bytes of pad characters, making the padded text a multiple of the block size.

The AES key used to encipher the data can either be 16, 24, or 32 bytes (128, 192, or 256 bits) in length. The key can be supplied to the verb in any of three forms:

1. A cleartext key consisting of only the key bytes, not contained in a key token.
2. A cleartext key contained in an internal fixed or variable length AES key-token.
3. An encrypted key contained in an internal fixed or variable length AES key-token, where the key is wrapped (encrypted) with the AES master key.

To use this verb, specify:

- The **rule_array**:
  1. The algorithm identifier keyword **AES**, which is the only symmetric algorithm currently supported.
  2. An optional processing rule using keyword **CBC** (the default), **ECB**, or **PKCS-PAD**, which selects the encryption mode.
  3. An optional key rule using the keyword **KEY-CLR** (the default) or **KEYIDENT**, which selects whether the key_identifier parameter points to a 16-byte, 24-byte, or 32-byte clear key, or a key contained in a 64-byte AES key-token in application storage or a key label of such a key in AES key-storage.
  4. An optional ICV (initial chaining value) selection using the keyword **INITIAL** (the default) or **CONTINUE**, which indicates whether it is the first or a subsequent request, and which parameter points to the initialization vector.

- For a key rule of **KEY-CLR**, a key identifier containing a 16-byte, 24-byte, or 32-byte clear key. For a key rule of **KEYIDENT**, a fixed-length or variable-length internal AES key-token or the key label of such a key in AES key-storage. The key token can contain either a clear or enciphered key.

A variable-length AES key-token must have a key that can be used for decryption (key-usage field 1 high-order byte is B'x1xx xxxx'). Also, for processing rule **CBC** or **PKCS-PAD**, key usage must allow the key to be used for Cipher Block Chaining (KUF2 high-order byte is X'00'). For processing rule **ECB**, key usage must allow the key to be used for Electronic Code Book (KUF2 high-order byte is X'01').

- A block size of 16 for the cryptographic algorithm.

- For cipher block chaining, either one of these:
Symmetric Algorithm Encipher (CSNBSAE)

1. For an ICV selection of INITIAL, a 16-byte initialization vector of your choosing and a 32-byte chain data buffer.
2. For an ICV selection of CONTINUE, no initialization vector and the 32-byte chain data buffer from the output of the previous chained call. The electronic code book algorithm does not use an initialization vector or a chain data buffer.
   - The cleartext to be enciphered.
   - A ciphertext buffer large enough to receive the enciphered output.

This verb does the following when it enciphers the data:
1. Verifies the AES key-token for keyword KEYIDENT.
2. Deciphers the input AES key if the key is encrypted (MKVP was present in token).
3. Pads the cleartext data with 1 - 16 bytes on the right for keyword PKCS-PAD, otherwise verifies that the cleartext length is a multiple of the block size.
4. Enciphers the cleartext, including any pad characters, with the AES clear key according to the encryption mode specified.
5. Returns the ciphertext data and its length.
6. Returns the chain data and its length if keyword ECB is not specified.

Only keys with a key type of DATA can be used successfully with the CPACF exploitation layer through this verb. Specifically, a DATA key has a Control Vector (CV) of all X'00' bytes for all active bytes of the CV (eight bytes for all AES keys). Note that as of CCA Release 3.30 (the first release of AES function on the CEX2C 4764 adapter) to Release 4.2 (the most recent release of CCA on the CEXC adapter), AES keys were only available as DATA keys. For details about CPACF, see [CPACF support](page 12).

Format

The format of CSNBSAE.

```c
CSNBSAE(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier_length,
    key_identifier,
    key_parms_length,
    key_parms,
    block_size,
    initialization_vector_length,
    initialization_vector,
    chain_data_length,
    chain_data,
    cleartext_length,
    cleartext,
    ciphertext_length,
    cipher_text,
    optional_data_length,
    optional_data)
```

Parameters

The parameters for CSNBSAE.
Symmetric Algorithm Encipher (CSNBSAE)

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**
- **Direction:** Input
- **Type:** Integer
- A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1, 2, 3, or 4.

**rule_array**
- **Direction:** Input
- **Type:** Array
- A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length, and must be left-aligned and padded on the right with space characters. The rule_array keywords are described in Table 70.

Table 70. Keywords for Symmetric Algorithm Encipher control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encryption algorithm</strong> (Required)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Specifies use of the Advanced Encryption Standard (AES) as the encryption algorithm. The block size for AES is 16 bytes, and the key length is 16, 24, or 32 bytes. AES is the only algorithm currently supported by this verb.</td>
</tr>
<tr>
<td><strong>Processing rule</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>CBC</td>
<td>Performs ANSI X3.102 cipher block chaining. The data must be a multiple of eight bytes. An OCV is produced and placed in the chaining_vector parameter. If the ICV selection keyword CONTINUE is specified, the CBC OCV from the previous call is used as the ICV for this call.</td>
</tr>
<tr>
<td>ECB</td>
<td>Specifies enciphering in Electronic Code Book mode. The cleartext length must be a multiple of the block size.</td>
</tr>
<tr>
<td>PKCS-PAD</td>
<td>Specifies padding of the cleartext on the right with 1-16 bytes of pad characters, making the padded text a multiple of the block size. Each pad character is valued to the number of pad characters added. The ciphertext length must be large enough to include the added pad characters. The ciphertext length must be large enough to include the added pad characters. The padded cleartext is enciphered in Cipher Block Chaining mode.</td>
</tr>
<tr>
<td><strong>Key rule</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>KEY-CLR</td>
<td>Specifies that the key_identifier parameter points to a cleartext AES key. Only the key value is allowed; the key is not contained in a key token. This is the default value.</td>
</tr>
<tr>
<td>KEYIDENT</td>
<td>Specifies that the key_identifier parameter points to an internal AES key-token or the label of an internal key-token in AES key-storage.</td>
</tr>
<tr>
<td><strong>ICV selection</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Specifies taking the initialization vector from the output chaining vector (OCV) contained in the work area to which the chaining_vector parameter points. CONTINUE is valid only for the CBC processing rule.</td>
</tr>
<tr>
<td>INITIAL</td>
<td>Specifies taking the initialization vector from the initialization_vector parameter. INITIAL is the default value.</td>
</tr>
</tbody>
</table>

"Ciphering methods" on page 702 describes the cipher processing rules in detail.

**key_identifier_length**
- **Direction:** Input
- **Type:** Integer
Symmetric Algorithm Encipher (CSNBSAE)

A pointer to an integer variable containing the number of bytes of data in the
key_identifier variable. This value must be 16, 24, 32, or ≥ 64.

key_identifier
Direction: Input
Type: String

A pointer to a string variable containing either a cleartext AES key or the
internal key-token or a label for an internal key-token record in AES
key-storage. This is the key used to encipher the data pointed to by the
cleartext parameter. For rule_array keyword KEY-CLR, a 16-byte, 24-byte, or
32-byte clear AES key is required. For rule_array keyword KEYIDENT, a
64-byte fixed or variable-length internal AES key-token or key label for such a
key in AES key-storage is required.

key_parms_length
Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the
key_parms parameter. This value must be 0.

key_parms
Direction: Input
Type: String

A pointer to a string variable for key-related parameters. It is currently unused.

block_size
Direction: Input
Type: Integer

A pointer to an integer variable containing the block size used by the
cryptographic algorithm. This value must be 16.

initialization_vector_length
Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the
initialization_vector variable. For cipher block chaining (CBC or PKCS-PAD)
with an INITIAL ICV selection, this value must be 16. For processing rule ECB
or ICV selection CONTINUE, this value should be 0.

initialization_vector
Direction: Input
Type: String

A pointer to a string variable containing the initialization vector for the
INITIAL call to CBC mode encryption. It is not used if the process rule is ECB
or the ICV selection is CONTINUE. The same initialization vector must be
used when deciphering the data.

chain_data_length
Direction: Input/Output
Type: Integer
Symmetric Algorithm Encipher (CSNBSAE)

A pointer to an integer variable containing the number of bytes of data in the chain_data variable. On input, this variable contains the length of the buffer provided and should have a value of 32 or greater for CBC mode encryption, or 0 for ECB mode encryption.

On output, the variable is updated with the length of the data returned in the chain_data variable. The chain_data_length parameter must not be changed by the calling application until chained operations are complete.

chain_data
Direction: Input/Output
Type: String

A pointer to a string variable used as a work area for CBC encipher requests. This work area is not used for ECB mode encryption. When the verb performs a CBC encipher operation and the ICV selection is INITIAL, the chain_data variable is an output-only buffer that receives data used as input for enciphering the next part of the input data, if any. When the ICV selection is CONTINUE, the chain_data variable is both an input and output buffer. The application must not change any intermediate data in this string.

cleartext_length
Direction: Input
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the cleartext variable. This length must be a multiple of the block_size variable unless processing rule PKCS-PAD is specified. A value of zero is not permitted.

cleartext
Direction: Input
Type: String

A pointer to a string variable used to contain the data to be enciphered, excluding any pad bytes.

ciphertext_length
Direction: Input/Output
Type: Integer

On input, the ciphertext_length parameter is a pointer to an integer variable containing the number of bytes of data in the ciphertext variable. On output, the ciphertext_length variable is updated to contain the actual length of text output in the ciphertext variable. If PKCS-PAD is specified, the ciphertext_length value must be greater than or equal to the next higher multiple of 16 as the cleartext_length value (from 1 - 16 bytes longer). Otherwise, the ciphertext_length value must be greater than or equal to the cleartext_length variable.

ciphertext
Direction: Input/Output
Type: String

A pointer to a string variable used as an output buffer where the verb returns the enciphered data. If PKCS-PAD is specified, on output the ciphertext buffer contains 1 - 16 bytes of data more than the cleartext input buffer contains.

optional_data_length
Symmetric Algorithm Encipher (CSNBSAE)

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the `optional_data` variable. This value should be 0.

**optional_data**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing optional data for the encryption. It is currently not used.

**Restrictions**  
The restrictions for CSNBSAE.

None.

**Required commands**  
The required commands for CSNBSAE.

This verb requires the Symmetric Algorithm Encipher - secure AES keys command (offset X'012A') to be enabled in the active role.

**Usage notes**  
The usage notes for CSNBSAE.

None

**JNI version**  
This verb has a Java Native Interface (JNI) version, which is named CSNBSAEJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBSAEJ are shown here.

**Format**

```java
public native void CSNBSAEJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_length,
    byte[] key_identifier,
    hikmNativeInteger key_parms_length,
    byte[] key_parms,
    hikmNativeInteger block_size,
    hikmNativeInteger iv_length,
    byte[] iv,
    hikmNativeInteger chain_data_length,
    byte[] chain_data,
    hikmNativeInteger clear_text_length,
    byte[] clear_text,
    hikmNativeInteger cipher_text_length,
```

Symmetric Algorithm Encipher (CSNBSAE)

byte[]                cipher_text,
hkmNativeInteger    optional_data_length,
byte[]                optional_data;
Symmetric Algorithm Encipher (CSNBSAE)
CCA provides methods to verify the integrity of transmitted messages and stored data.

The methods provided are:
- Message authentication code (MAC)
- Hash functions, including Modification Detection Code (MDC) processing and one-way hash generation

**Note:** You can also use digital signatures (see Chapter 11, “Using digital signatures,” on page 467) to authenticate messages.

The choice of verb depends on the security requirements of the environment in which you are operating. If you need to ensure the authenticity of the sender as well as the integrity of the data and both the sender and receiver can share a secret key, consider Message Authentication Code processing. If you need to ensure the integrity of transmitted data in an environment where it is not possible for the sender and the receiver to share a secret cryptographic key, consider hashing functions.

The verbs described in this chapter include:
- “HMAC Generate (CSNBHMG)” on page 315
- “HMAC Verify (CSNBHMV)” on page 319
- “MAC Generate (CSNBMGN)” on page 323
- “MAC Verify (CSNBMVR)” on page 327
- “MDC Generate (CSNBMDG)” on page 332
- “One-Way Hash (CSNBOWH)” on page 342

### How MACs are used

When a message is sent, an application program can generate an authentication code for it using the MAC Generate verb.

CCA supports the ANSI X9.9-1 basic procedure and both the ANSI X9.19 basic procedure and optional double key MAC procedure. The MAC Generate verb computes the text of the Message Authentication Code using the algorithm and a key. The ANSI X9.9-1 or ANSI X9.19 basic procedures accept either a single-length MAC generation (MAC) key or a data-encrypting (DATA) key, and the message text. The ANSI X9.19 optional double key MAC procedure accepts a double-length MAC key and the message text. The originator of the message sends the MAC with the message text.

When the receiver gets the message, an application program calls the MAC Verify verb. The MAC Generate verb generates a MAC using the same algorithm as the sender and either the single-length or double-length MAC verification key, the single-length or double-length MAC generation key, or DATA key, and the message text. The MAC Verify verb compares the MAC it generates with the one sent with the message and issues a return code that indicates whether the MACs match. If
the return code indicates that the MACs match, the receiver can accept the message as genuine and unaltered. If the return code indicates that the MACs do not match, the receiver can assume the message is either fraudulent or has been altered. The newly computed MAC is not revealed outside the cryptographic coprocessor.

In a similar manner, MACs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

Secure use of the MAC Generate and MAC Verify verbs requires the use of MAC and MACVER keys in these verbs, respectively. To accomplish this, the originator of the message generates a MAC/MACVER key pair, uses the MAC key in the MAC Generate verb, and exports the MACVER key to the receiver. The originator of the message enforces key separation on the link by encrypting the MACVER key under a transport key that is not an NOCV key before exporting the key to the receiver. With this type of key separation enforced, the receiver can receive only a MACVER key and can use only this key in the MAC Verify verb. This ensures that the receiver cannot alter the message and produce a valid MAC with the altered message. These security features are not present if DATA keys are used in the MAC Generate verb or if DATA or MAC keys are used in the MAC Verify verb.

By using MACs you get the following benefits:

- **For data transmitted over a network**, you can validate the authenticity of the message as well as ensure the data has not been altered during transmission. For example, an active eavesdropper can tap into a transmission line and interject fraudulent messages or alter sensitive data being transmitted. If the data is accompanied by a MAC, the recipient can use a verb to detect whether the data has been altered. Because both the sender and receiver share a secret key, the receiver can use a verb that calculates a MAC on the received message and compares it to the MAC transmitted with the message. If the comparison is equal, the message could be accepted as unaltered. Furthermore, because the shared key is secret, when a MAC is verified it can be assumed that the sender was, in fact, the other person who knew the secret key.

- **For data stored on tape or DASD**, you can ensure that the data read back onto the system was the same as the data written onto the tape or DASD. For example, someone might be able to bypass access controls. Such an access might escape the notice of auditors. However, if a MAC is stored with the data, and verified when the data is read, you can detect alterations to the data.

### How hashing functions and MDCs are used

Hashing functions include the MDC and one-way hash.

You need to hash text before submitting it to the Digital Signature Generate and Digital Signature Verify verbs (see Chapter 11, “Using digital signatures,” on page 467). CCA supports the SHA-1, MD5, and RIPEMD-160 hashing functions.

When a message is sent, an application program can generate a hash or a Modification Detection Code (MDC) for it using the One-Way Hash verb. This verb computes the hash or MDC, a short, fixed-length value, using a one-way cryptographic function and the message text. The originator of the message ensures the hash or MDC is transmitted with integrity to the intended receiver of the message. For example, the value could be published in a reliable source of public information.
When the receiver gets the message, an application program calls the One-Way Hash verb to generate a new hash or MDC using the same function and message text that were used by the sender. The application program can compare the new value with the one generated by the originator of the message. If the two values match, the receiver knows the message was not altered.

In a similar manner, hashes and MDCs can be used to ensure the integrity of data stored on the system or on removable media, such as tape.

By using hashes and MDCs, you get the following benefits:

- **For data transmitted over a network between locations that do not share a secret key**, you can ensure the data has not been altered during transmission. It is easy to compute a hash or MDC for specific data, yet hard to find data that will result in a given hash or MDC. In effect, the problem of ensuring the integrity of a large file is reduced to ensuring the integrity of a short, fixed-length value.

- **For data stored on tape or DASD**, you can ensure that the data read back onto the system was the same as the data written onto the tape or DASD. After a hash has been established for a file, the One-Way Hash verb can be run at any later time on the file. The resulting value can be compared with the stored value to detect deliberate or inadvertent modification.

For more information, see “Modification Detection Code calculation” on page 700.

### HMAC Generate (CSNBHMG)

Use the HMAC Generate verb to generate a keyed hash Message Authentication Code (HMAC) for the message string provided as input.

An HMAC key that can be used for generate is required to calculate the HMAC.

#### Format

The format of CSNBHMG.

```plaintext
CSNBHMG(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  rule_array_count,
  rule_array,
  key_identifier_length,
  key_identifier,
  text_length,
  text,
  chaining_vector_length,
  chaining_vector,
  mac_length,
  mac)
```

#### Parameters

The parameters for CSNBHMG.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`
The number of keywords you supplied in the `rule_array` parameter. This value must be 2 or 3.

**rule_array**

**Direction: Input**
**Type: String**

Keywords that provide control information to the verb. The following table lists the keywords. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 71.

### Table 71. Keywords for HMAC Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token algorithm</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td>HMAC</td>
<td>Specifies the HMAC algorithm to be used to generate the MAC.</td>
</tr>
<tr>
<td><strong>Hash method</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td>SHA-1</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-1 hash method, a symmetric key and text to produce a 20-byte (160-bit) MAC.</td>
</tr>
<tr>
<td>SHA-224</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-224 hash method, a symmetric key and text to produce a 28-byte (224-bit) MAC.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-256 hash method, a symmetric key and text to produce a 32-byte (256-bit) MAC.</td>
</tr>
<tr>
<td>SHA-384</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-384 hash method, a symmetric key and text to produce a 48-byte (384-bit) MAC.</td>
</tr>
<tr>
<td>SHA-512</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-512 hash method, a symmetric key and text to produce a 64-byte (512-bit) MAC.</td>
</tr>
<tr>
<td><strong>Segmenting control</strong> (One optional)</td>
<td></td>
</tr>
<tr>
<td>FIRST</td>
<td>First call, this is the first segment of data from the application program.</td>
</tr>
<tr>
<td>LAST</td>
<td>Last call; this is the last data segment.</td>
</tr>
<tr>
<td>MIDDLE</td>
<td>Middle call; this is an intermediate data segment.</td>
</tr>
<tr>
<td>ONLY</td>
<td>Only call; segmenting is not employed by the application program. This is the default value.</td>
</tr>
</tbody>
</table>

**key_identifier_length**

**Direction: Input**
**Type: Integer**

The length of the `key_identifier` parameter. The maximum value is 725.

**key_identifier**

**Direction: Input**
**Type: String**

The 64-byte label or internal token of an encrypted HMAC key.

**text_length**

**Direction: Input**
**Type: Integer**

The length of the text you supply in the `text` parameter. The maximum length of `text` is 214783647 bytes. For `FIRST` and `MIDDLE` calls, the `text_length` must...
be a multiple of 64 for SHA-1, SHA-224 and SHA-256 hash methods, and a multiple of 128 for SHA-384 and SHA-512 hash methods.

text
Direction: Input
Type: String

The application-supplied text for which the MAC is generated.

chaining_vector_length
Direction: Input/Output
Type: Integer

The length of the chaining_vector in bytes. This value must be 128.

chaining_vector
Direction: Input/Output
Type: String

An 128-byte string used as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mac_length
Direction: Input/Output
Type: Integer

The length of the mac parameter in bytes. This parameter is updated to the actual length of the mac parameter on output. The minimum value is 4, and the maximum value is 64.

mac
Direction: Output
Type: String

The field in which the verb returns the MAC value if the segmenting rule is ONLY or LAST.

Restrictions
The restrictions for CSNBHMG.

This verb was introduced with CCA 4.1.0.

Required commands
The required commands for CSNBHMG.

This verb requires the commands shown in the following table to be enabled in the active role:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>X'00E4'</td>
<td>HMAC Generate - SHA-1</td>
</tr>
<tr>
<td>SHA-224</td>
<td>X'00E5'</td>
<td>HMAC Generate - SHA-224</td>
</tr>
<tr>
<td>SHA-256</td>
<td>X'00E6'</td>
<td>HMAC Generate - SHA-256</td>
</tr>
<tr>
<td>SHA-384</td>
<td>X'00E7'</td>
<td>HMAC Generate - SHA-384</td>
</tr>
</tbody>
</table>
HMAC Generate (CSNBHMG)

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-512</td>
<td>X'00E8'</td>
<td>HMAC Generate - SHA-512</td>
</tr>
</tbody>
</table>

Usage notes

The usage notes for CSNBHMG.

None

Related information

More information about CSNBHMG is provided in other topics.

The HMAC Verify verb is described in “HMAC Verify (CSNBHMV)” on page 319.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBHMGJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBHMGJ are shown here.

Format

```java
public native void CSNBHMGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_identifier_length
    byte[] key_identifier,
    hikmNativeInteger text_length,
    byte[] text,
    hikmNativeInteger chaining_vector_length,
    byte[] chaining_vector,
    hikmNativeInteger mac_length,
    byte[] MAC);`
HMAC Verify (CSNBHMV)

Use the HMAC Verify verb to verify a keyed hash Message Authentication Code (HMAC) for the message string provided as input.

A MAC key contained in an internal variable-length symmetric key-token is required to verify the HMAC. The key must have the same value as the key used to generate the HMAC.

Format

The format of CSNBHMV.

```plaintext
CSNBHMV(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier_length,
    key_identifier,
    text_length,
    text,
    chaining_vector_length,
    chaining_vector,
    mac_length,
    mac)
```

Parameters

The parameters for CSNBHMV.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

Direction: Input
Type: Integer

The number of keywords you supplied in the `rule_array` parameter. This value must be 2 or 3.

**rule_array**

Direction: Input
Type: String

Keywords that provide control information to the verb. The following table lists the keywords. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 72.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HMAC</strong></td>
<td>Specifies that the HMAC algorithm is to be used to verify the MAC.</td>
</tr>
</tbody>
</table>

Table 72. Keywords for HMAC Verify control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token algorithm</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td><strong>HMAC</strong></td>
<td>Specifies that the HMAC algorithm is to be used to verify the MAC.</td>
</tr>
<tr>
<td><strong>Hash method</strong> (One required)</td>
<td></td>
</tr>
</tbody>
</table>
Table 72. Keywords for HMAC Verify control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-1 hash method, a symmetric key and text to produce a 20-byte (160-bit) MAC.</td>
</tr>
<tr>
<td>SHA-224</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-224 hash method, a symmetric key and text to produce a 28-byte (224-bit) MAC.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-256 hash method, a symmetric key and text to produce a 32-byte (256-bit) MAC.</td>
</tr>
<tr>
<td>SHA-384</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-384 hash method, a symmetric key and text to produce a 48-byte (384-bit) MAC.</td>
</tr>
<tr>
<td>SHA-512</td>
<td>Specifies the FIPS-198 HMAC procedure using the SHA-512 hash method, a symmetric key and text to produce a 64-byte (512-bit) MAC.</td>
</tr>
</tbody>
</table>

**Segmenting control** (Optional)

| FIRST  | First call, this is the first segment of data from the application program. |
| LAST   | Last call; this is the last data segment. |
| MIDDLE | Middle call; this is an intermediate data segment. |
| ONLY   | Only call; segmenting is not employed by the application program. This is the default value. |

**key_identifier_length**

- **Direction:** Input
- **Type:** Integer

  The length of the `key_identifier` parameter. The maximum value is 725.

**key_identifier**

- **Direction:** Input/Output
- **Type:** String

  The 64-byte label or internal token of an encrypted HMAC or HMACVER key.

**text_length**

- **Direction:** Input
- **Type:** Integer

  The length of the text you supply in the `text` parameter. The maximum length of `text` is 214783647 bytes. For `FIRST` and `MIDDLE` calls, the `text_length` must be a multiple of 64 for `SHA-1`, `SHA-224`, and `SHA-256` hash methods, and a multiple of 128 for `SHA-384` and `SHA-512` hash methods.

**text**

- **Direction:** Input
- **Type:** String

  The application-supplied text for which the HMAC is to be verified.

**chaining_vector_length**

- **Direction:** Input/Output
- **Type:** Integer

  The length of the `chaining_vector` in bytes. This value must be 128.

**chaining_vector**

- **Direction:** Input/Output
- **Type:** String
HMAC Verify (CSNBHMV)

An 128-byte string used as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mac_length
Direction: Input
Type: Integer
The length of the mac parameter in bytes. The maximum value is 64.

mac
Direction: Input
Type: String
The field that contains the MAC value you want to verify.

Restrictions
The restrictions for CSNBHMV.

This verb was introduced with CCA 4.1.0.

Required commands
The required commands for CSNBHMV.

This verb requires the commands shown in the following table to be enabled in the active role:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>X'00F7'</td>
<td>HMAC Verify - SHA-1</td>
</tr>
<tr>
<td>SHA-224</td>
<td>X'00F8'</td>
<td>HMAC Verify - SHA-224</td>
</tr>
<tr>
<td>SHA-256</td>
<td>X'00F9'</td>
<td>HMAC Verify - SHA-256</td>
</tr>
<tr>
<td>SHA-384</td>
<td>X'00FA'</td>
<td>HMAC Verify - SHA-384</td>
</tr>
<tr>
<td>SHA-512</td>
<td>X'00FB'</td>
<td>HMAC Verify - SHA-512</td>
</tr>
</tbody>
</table>

Usage notes
The usage notes for CSNBHMV.

None

Related information
Additional information about CSNBHMV,

The HMAC Generate verb is described in "HMAC Generate (CSNBHMG)" on page 315.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBHMVJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBHMVJ are shown here.
HMAC Verify (CSNBHMV)

Format

class CSNBHMVJ{
    public native void CSNBHMVJ(
        hikmNativeInteger return_code,
        hikmNativeInteger reason_code,
        hikmNativeInteger exit_data_length,
        byte[] exit_data,
        hikmNativeInteger rule_array_count,
        byte[] rule_array,
        hikmNativeInteger key_identifier_length
        byte[] key_identifier,
        hikmNativeInteger text_length,
        byte[] text,
        hikmNativeInteger chaining_vector_length,
        byte[] chaining_vector,
        hikmNativeInteger mac_length,
        byte[] MAC);
MAC Generate (CSNBMGN)

When a message is sent, an application program can generate an authentication code for it using the MAC Generate verb.

This verb generates a 4-byte, 6-byte, or 8-byte Message Authentication Code (MAC) for an application-supplied text string.

This verb computes the Message Authentication Code using one of the following methods:
- Using the ANSI X9.9-1 single key algorithm, a single-length MAC generation key or data-encrypting key, and the message text.
- Using the ANSI X9.19 optional double key algorithm, a double-length MAC generation key and the message text.
- Using the Europay, MasterCard and Visa (EMV) padding rules.

The MAC can be the leftmost 32 or 48 bits of the last block of the ciphertext or the entire last block (64 bits) of the ciphertext. The originator of the message sends the Message Authentication Code with the message text.

Host CPU acceleration: CPACF

Only keys with a key type of DATA can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a DATA key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see "CPACF support" on page 12.

Format

The format of CSNBMGN.

```
CSNBMGN(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  key_identifier,
  text_length,
  text,
  rule_array_count,
  rule_array,
  chaining_vector,
  mac)
```

Parameters

The parameters for CSNBMGN.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

key_identifier

  Direction: Input/Output
  Type: String
MAC Generate (CSNBMGN)

The 64-byte key label or internal key token that identifies a single-length or double-length MAC generate key or a single-length DATA or DATAM key. The type of key depends on the MAC process rule in the rule_array parameter.

text_length

Direction: Input
Type: Integer

The length of the text you supply in the text parameter. If the text_length is not a multiple of eight bytes and if the ONLY or LAST keyword of the rule_array parameter is called, the text is padded in accordance with the processing rule specified.

text

Direction: Input
Type: String

The application-supplied text for which the MAC is generated.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0, 1, 2, or 3.

rule_array

Direction: Input
Type: String

Zero to three keywords that provide control information to the verb. The keywords are described in Table 73. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. For example, 'X9.9-1 MIDDLE MACLEN4'

The order of the rule_array keywords is not fixed.

You can specify one of the MAC processing rules and then choose one of the segmenting control keywords and one of the MAC length keywords. The rule_array keywords are described in Table 73.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAC process rules (One, optional)</strong></td>
<td></td>
</tr>
<tr>
<td>EMVMAC</td>
<td>EMV padding rule with a single-length MAC key. The key_identifier parameter must identify a single-length MAC or a single-length DATA key. The text is always padded with 1 - 8 bytes so the resulting text length is a multiple of eight bytes. The first pad character is X'80'. The remaining pad characters are X'00'.</td>
</tr>
<tr>
<td>EMVMACD</td>
<td>EMV padding rule with a double-length MAC key. The key_identifier parameter must identify a double-length MAC key. The padding rules are the same as for keyword EMVMAC.</td>
</tr>
<tr>
<td>TDES-MAC</td>
<td>ANSI X9.9-1 procedure using ISO 16609 CBC mode triple-DES (TDES) encryption of the data. Uses a double-length key.</td>
</tr>
<tr>
<td>X9.19OPT</td>
<td>ANSI X9.19 optional double key MAC procedure. The key_identifier parameter must identify a double-length MAC key. The padding rules are the same as for keyword X9.9-1.</td>
</tr>
</tbody>
</table>
MAC Generate (CSNBMGN)

Table 73. Keywords for MAC Generate control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X9.9-1</td>
<td>ANSI X9.9-1 and X9.19 basic procedure. The key_identifier parameter must identify a single-length MAC or a single-length DATA key. X9.9-1 causes the MAC to be computed from all of the data. The text is padded only if the text length is not a multiple of eight bytes. If padding is required, the pad character X'00' is used. This is the default value.</td>
</tr>
</tbody>
</table>

Segmenting control (One, optional)

| FIRST   | First call; this is the first segment of data from the application program. |
| LAST    | Last call; this is the last data segment. |
| MIDDLE  | Middle call; this is an intermediate data segment. |
| ONLY    | Only call; segmenting is not employed by the application program. This is the default value. |

MAC length and presentation (One, optional)

| HEX-8   | Generates a 4-byte MAC value and presents it as 8 hexadecimal characters. |
| HEX-9   | Generates a 4-byte MAC value and presents it as two groups of 4 hexadecimal characters with a space between the groups. |
| MACLEN4 | Generates a 4-byte MAC value. This is the default value. |
| MACLEN6 | Generates a 6-byte MAC value. |
| MACLEN8 | Generates an 8-byte MAC value. |

chaining_vector

Direction: Input/Output
Type: String
An 18-byte string that CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter as binary zeros.

mac

Direction: Output
Type: String
The 8-byte or 9-byte field in which the verb returns the MAC value if the segmenting rule is ONLY or LAST. Allocate an 8-byte field for MAC values of four bytes, six bytes, eight bytes, or HEX-8. Allocate a 9-byte MAC field if you specify HEX-9 in the rule_array parameter.

Restrictions

The restrictions for CSNBMGN.

It might seem intuitive that a DATAM key should also be usable for the MAC Generate verb, and a DATAMV key for the MAC Verify verb, with the CPACF exploitation layer. However, this would violate the security restrictions imposed by the user when the user creates a key of type DATAM or DATAMV. A DES key that has been translated for use with the CPACF (see “CPACF support” on page 12) can be used with CPACF DES encrypt and decrypt operations, an operation that is by definition not allowed for a DATAM or DATAMV key type. Also note that by definition both through z/OS CCA-ICSF and in this S390 Linux CCA access layer, a DATA key of 16 bytes or 24 bytes in length is restricted from use.
MAC Generate (CSNBMGN)

with the X9.19OPT and EMVMACD rule_array keyword specified MAC algorithms. The only available MAC algorithm for a 16-byte or 24-byte DATA key is the TDES-MAC algorithm.

Also note that the CPACF exploitation layer is activated only for MAC Generate or MAC Verify calls that specify the ONLY rule_array keyword for segmenting control (this is the default segmenting control if no segmenting control rule_array keyword is specified). The reason for this is that the intermediate MAC context for normal CEX*C calls to MAC Generate and MAC Verify is protected by the adapter Master Key. Because the same security cannot be provided for intermediate results from the host-based CPACF exploitation layer (they are returned in the clear by the CPACF) the FIRST, MIDDLE, and LAST segmenting control keywords will direct operations to the CEX*C.

Required commands
The required commands for CSNBMGN.

This verb requires the MAC Generate command (offset X'0010') to be enabled in the active role.

Usage notes
The usage notes for CSNBMGN.

None

Related information
Additional information about CSNBMGN.

The MAC Verify verb is described in "MAC Verify (CSNBMVR)" on page 327.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBMGNJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBMGNJ are shown here.

Format

```java
public native void CSNBMGNJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_identifier,
    hikmNativeInteger text_length,
    byte[] text,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] chaining_vector,
    byte[] MAC);
```
**MAC Verify (CSNBMVR)**

When the receiver gets a message, an application program calls the MAC Verify verb.

This verb verifies a 4-byte, 6-byte, or 8-byte Message Authentication Code (MAC) for an application-supplied text string. This verb verifies a MAC by generating another MAC and comparing it with the MAC received with the message. This process takes place entirely within the secure module on the coprocessor. If the two codes are the same, the message sent was the same one received. A return code indicates whether the MACs are the same. The generated MAC never appears in storage and is not revealed outside the cryptographic feature.

The MAC Verify verb can use any of the following methods to generate the MAC for authentication:
- The ANSI X9.9-1 single key algorithm, a single-length MAC verification or MAC generation key (or a data-encrypting key), and the message text.
- The ANSI X9.19 optional double key algorithm, a double-length MAC verification or MAC generation key and the message text.
- Using the Europay, MasterCard and Visa (EMV) padding rules.

The method used to verify the MAC should correspond with the method used to generate the MAC.

### Host CPU acceleration: CPACF

Only keys with a key type of **DATA** can be used successfully with the CPACF exploitation layer through this verb.

Specifically, a **DATA** key has a CV (Control Vector) of all X'00' bytes for all active bytes of the CV (eight bytes for 8-byte DES keys, 16 bytes for 16-byte DES keys, and 16 bytes for 24-byte DES keys).

For details about CPACF, see "CPACF support" on page 12.

### Format

The format of CSNBMVR.

```c
CSNBMVR(  return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_identifier,
    text_length,
    text,
    rule_array_count,
    rule_array,
    chaining_vector,
    mac)
```

### Parameters

The parameters for CSNBMVR.

For the definitions of the **return_code**, **reason_code**, **exit_data_length**, and **exit_data** parameters, see "Parameters common to all verbs" on page 20.
MAC Verify (CSNBMVR)

**key_identifier**

**Direction:** Input/Output  
**Type:** String  
The 64-byte key label or internal key token that identifies a single-length or double-length MAC verify key, a single-length or double-length MAC generation key, or a single-length DATA key. The type of key depends on the MAC process rule in the `rule_array` parameter.

**text_length**

**Direction:** Input  
**Type:** Integer  
The length of the clear text you supply in the `text` parameter. If the `text_length` parameter is not a multiple of eight bytes and if the **ONLY** or **LAST** keyword of the `rule_array` parameter is called, the text is padded in accordance with the processing rule specified.

**text**

**Direction:** Input  
**Type:** String  
The application-supplied text for which the MAC is verified.

**rule_array_count**

**Direction:** Input  
**Type:** Integer  
A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0, 1, 2, or 3.

**rule_array**

**Direction:** Input  
**Type:** String  
Zero to three keywords that provide control information to the verb. The keywords are described in Table 74. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. For example,  

'X9.9-1 MIDDLE MACLEN4 '

The order of the `rule_array` keywords is not fixed.

You can specify one of the MAC processing rules, and then choose one of the segmenting control keywords and one of the MAC length keywords. The `rule_array` keywords are described in Table 74.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EMVMAC</strong></td>
<td>EMV padding rule with a single-length MAC key. The <code>key_identifier</code> parameter must identify a single-length MAC, MACVER, or DATA key. The text is always padded with 1-8 bytes, so that the resulting text length is a multiple of eight bytes. The first pad character is X'80'. The remaining pad characters are X'00'.</td>
</tr>
<tr>
<td><strong>EMVMACD</strong></td>
<td>EMV padding rule with a double-length MAC key. The <code>key_identifier</code> parameter must identify a double-length MAC or MACVER key. The padding rules are the same as for <strong>EMVMAC</strong>.</td>
</tr>
</tbody>
</table>

Table 74. Keywords for MAC Verify control information
### Table 74. Keywords for MAC Verify control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDES-MAC</td>
<td>ANSI X9.9-1 procedure using ISO 16609 CBC mode triple-DES (TDES) encryption of the data. Uses a double-length key.</td>
</tr>
<tr>
<td>X9.9-1</td>
<td>ANSI X9.9-1 and X9.19 basic procedure. The key_identifier parameter must identify a single-length MAC, MACVER, or DATA key. X9.9-1 causes the MAC to be computed from all the data. The text is padded only if the text length is not a multiple of eight bytes. If padding is required, the pad character X'00' is used. This is the default value.</td>
</tr>
<tr>
<td>X9.19OPT</td>
<td>ANSI X9.19 optional double-length MAC procedure. The key_identifier parameter must identify a double-length MAC or MACVER key. The padding rules are the same as for X9.9-1.</td>
</tr>
</tbody>
</table>

### Segmenting control (Optional)

- **FIRST**: First call. This is the first segment of data from the application program.
- **LAST**: Last call. This is the last data segment.
- **MIDDLE**: Middle call. This is an intermediate data segment.
- **ONLY**: Only call. The application program does not employ segmenting. This is the default value.

### MAC length and presentation (Optional)

- **HEX-8**: Verifies a 4-byte MAC value represented as 8 hexadecimal characters.
- **HEX-9**: Verifies a 4-byte MAC value represented as two groups of 4 hexadecimal characters with a space character between the groups.
- **MACLEN4**: Verifies a 4-byte MAC value. This is the default value.
- **MACLEN6**: Verifies a 6-byte MAC value.
- **MACLEN8**: Verifies an 8-byte MAC value.

#### chaining_vector

**Direction**: Input/Output  
**Type**: String

An 18-byte string CCA uses as a system work area. Your application program must not change the data in this string. The chaining vector permits data to be chained from one invocation call to another.

On the first call, initialize this parameter to binary zeros.

#### mac

**Direction**: Input  
**Type**: String

The 8-byte or 9-byte field that contains the MAC value you want to verify. The value in the field must be left-aligned and padded with zeros. If you specified the **HEX-9** keyword in the rule_array parameter, the input MAC is nine bytes in length.

### Restrictions

The restrictions for CSNBMVR.

It might seem intuitive that a **DATAM** key should also be usable for the MAC Generate verb, and a **DATAMV** key for the MAC Verify verb, with the CPACF exploitation layer. However, this would violate the security restrictions imposed by the user when the user creates a key of type **DATAM** or **DATAMV**. A DES key that has been translated for use with the CPACF (see ["CPACF support" on page 12](#)) can be used with CPACF DES encrypt and decrypt operations, an operation...
MAC Verify (CSNBMVR)

that is by definition not allowed for a DATAM or DATAMV key type. Also note that by definition both through z/OS CCA-ICSF and in this S390 Linux CCA access layer, a DATA key of 16 bytes or 24 bytes in length is restricted from use with the X9.19OPT and EMVMACD rule_array keyword specified MAC algorithms. The only available MAC algorithm for a 16-byte or 24-byte DATA key is the TDES-MAC algorithm.

Also note that the CPACF exploitation layer is activated only for MAC Generate or MAC Verify calls that specify the ONLY rule_array keyword for segmenting control (this is the default segmenting control if no segmenting control rule_array keyword is specified). The reason for this is that the intermediate MAC context for normal CEX* calls to MAC Generate and MAC Verify is protected by the adapter Master Key. Because the same security cannot be provided for intermediate results from the host-based CPACF exploitation layer (they are returned in the clear by the CPACF) the FIRST, MIDDLE, and LAST segmenting control keywords will direct operations to the CEX*.

Required commands

The required commands for CSNBMVR.

This verb requires the MAC Verify command (offset X'0011') to be enabled in the active role.

Usage notes

The usage notes for CSNBMVR.

To verify a MAC in one call, specify the ONLY keyword on the segmenting rule keyword for the rule_array parameter. For two or more calls, specify the FIRST keyword for the first input block, MIDDLE for intermediate blocks (if any), and LAST for the last block.

For a given text string, the MAC resulting from the verification process is the same regardless of how the text is segmented or how it was segmented when the original MAC was generated.

Related information

Additional information for CSNBMVR.

The MAC Generate verb is described in “MAC Generate (CSNBMGN)” on page 323.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBMVRJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBMVRJ are shown here.

Format

public native void CSNBMVRJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_identifier,
MAC Verify (CSNBMVR)

hikmNativeInteger text_length,
byte[] text,
 hikmNativeInteger rule_array_count,
byte[] rule_array,
byte[] chaining_vector,
byte[] MAC);
MDC Generate (CSNBMDG)

Use this verb to create a 128-bit hash value (Modification Detection Code) on a data string whose integrity you intend to confirm.

**IMPORTANT NOTICE**

In releases before Release 3.30, it was discovered that under certain conditions the MDC Generate verb produced incorrect MDC values. Beginning with Release 3.30.05, these conditions no longer produce incorrect results.

If you have MDC values that were generated using a release before Release 3.30.05, corrective action might be required before using these values with Release 3.30 (or later) to validate data integrity. See "Related information" on page 335 for detailed information.

Use this verb to create a 128-bit hash value (Modification Detection Code) on a data string whose integrity you intend to confirm. After using this verb to generate an MDC, you can compare the MDC to a known value or communicate the value to another entity so that they can compare the MDC hash value to one that they calculate. This verb enables you to perform the following tasks:

- Specify the two-encipherment or four-encipherment version of the algorithm.
- Segment your text into a series of verb calls.
- Use the default or a keyed-hash algorithm.

The user must enable the Generate MDC command with a Trusted Key Entry (TKE) workstation before using this verb.

For a description of the MDC calculations, see "Modification Detection Code calculation" on page 700.

**Specifying two or four encipherments:** Four encipherments per algorithm round improve security; two encipherments per algorithm round improve performance. To specify the number of encipherments, use the MDC-2, MDC-4, PADMDC-2, or PADMDC-4 keyword with the rule_array parameter. Two encipherments create results that differ from four encipherments; ensure that the same number of encipherments are used to verify the MDC.

**Segmenting text:** This verb lets you segment text into a series of verb calls. If you can present all of the data to be hashed in a single invocation of the verb (32 MB) of data, use the rule_array keyword ONLY. Alternatively, you can segment your text and present the segments with a series of verb calls. Use the rule_array keywords and LAST for the first and last segments. If more than two segments are used, specify the rule_array keyword MIDDLE for the additional segments.

Between verb calls, unprocessed text data and intermediate information from the partial MDC calculation is stored in the chaining_vector variable and the MDC key in the MDC variable. During segmented processing, the application program must not change the data in either of these variables.

**Keyed hash:** This verb can be used with a default key, or as a keyed-hash algorithm. A default key is used whenever the ONLY or FIRST segmenting and key control keywords are used. To use the verb as a keyed-hash algorithm, do the following:

1. On the first call to the verb, place the non-null key into the MDC variable.
2. Ensure that the chaining_vector variable is set to null (18 bytes of X'00').
3. Decide if the text will be processed in a single segment or multiple segments.
   - For a single segment of text, use the **LAST** keyword.
   - For multiple segments of text, begin with the **MIDDLE** keyword and continue using the **MIDDLE** keyword up to the final segment of text. For the final segment, use the **LAST** keyword.

As with the default key, you must not alter the value of the **MDC** or **chaining_vector** variables between calls.

### Format

The format of CSNBMDG.

```c
CSNBMDG(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   text_length,
   text,
   rule_array_count,
   rule_array,
   chaining_vector,
   MDC)
```

### Parameters

The parameters for CSNBMDG.

For the definitions of the **return_code**, **reason_code**, **exit_data_length**, and **exit_data** parameters, see "Parameters common to all verbs" on page 20.

**text_length**
- Direction: Input
- Type: Integer
- A pointer to an integer variable containing the number of bytes of data in the **text** variable. See "Restrictions" on page 334.

**text**
- Direction: Input
- Type: String
- A pointer to a string variable containing the text for which the verb calculates the MDC value.

**rule_array_count**
- Direction: Input
- Type: Integer
- A pointer to an integer variable containing the number of elements in the **rule_array** variable. This value can be 0, 1, or 2.

**rule_array**
- Direction: Input
- Type: String
- Keywords that provide control information to the verb. A keyword specifies the method for calculating the RSA digital signature. Each keyword is
MDC Generate (CSNBMDG)

left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 75.

Table 75. Keywords for MDC Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmenting and key control (One, optional)</td>
<td></td>
</tr>
<tr>
<td>ONLY</td>
<td>Specifies that segmenting is not used and the default key is used. This is the default.</td>
</tr>
<tr>
<td>FIRST</td>
<td>Specifies the first segment of text, and use of the default key.</td>
</tr>
<tr>
<td>MIDDLE</td>
<td>Specifies an intermediate segment of text, or the first segment of text and use of a user-supplied key.</td>
</tr>
<tr>
<td>LAST</td>
<td>Specifies the last segment of text, or that segmenting is not used, and use of a user-supplied key.</td>
</tr>
<tr>
<td>Algorithm mode (One, optional)</td>
<td></td>
</tr>
<tr>
<td>MDC-2</td>
<td>Specifies two encipherments for each 8-byte block using MDC procedures. This is the default.</td>
</tr>
<tr>
<td>MDC-4</td>
<td>Specifies four encipherments for each 8-byte block using MDC procedures.</td>
</tr>
<tr>
<td>PADMDC-2</td>
<td>Specifies two encipherments for each 8-byte block using PADMDC procedures.</td>
</tr>
<tr>
<td>PADMDC-4</td>
<td>Specifies four encipherments for each 8-byte block using PADMDC procedures.</td>
</tr>
</tbody>
</table>

**chaining_vector**

- **Direction:** Input/Output
- **Type:** String

A pointer to an 18-byte string variable the security server uses as a work area to hold segmented data between verb invocations.

**IMPORTANT:** When segmenting text, the application program must not change the data in this string between verb calls to the MDC Generate verb.

**MDC**

- **Direction:** Input/Output
- **Type:** String

A pointer to a user-supplied MDC key or to a 16-byte string variable containing the MDC value. This value can be the key that the application program provides. This variable is also used to hold the intermediate MDC result when segmenting text.

**IMPORTANT:** When segmenting text, the application program must not change the data in this string between verb calls to the MDC Generate verb.

**Restrictions**

The restrictions for CSNBMDG.

- When padding is requested (by specifying an algorithm mode keyword of PADMDC-2 or PADMDC-4), a text length of zero is valid for any segment-control keyword specified in the `rule_array` variable FIRST, MIDDLE, LAST, or ONLY. When LAST or ONLY is specified, the supplied text is padded with X’FF’ bytes and a padding count in the last byte to bring the total text length to the next multiple of 8 that is greater than or equal to 16.
- When no padding is requested (by specifying an algorithm mode keyword of MDC-2 or MDC-4), the total length of text provided (over a single or segmented calls) must be a minimum of 16 bytes and a multiple of eight bytes. For
segmented calls (that is, segmenting and key control keyword is not ONLY), a
text length of zero is valid on any of the calls.

**Required commands**

The required commands for CSNBMDG.

In releases prior to CCA 4.1.0 and for installations without CPACF support this
verb requires the MDC Generate command (offset X'008A') to be enabled in the
active role. This command is no longer required in CCA 4.1.0 when CPACF
clear-key function is available (KM, function 1), and enabled for CCA use
(environment variable CSU_HCPUACLR is set to '1', the default value).

The user must enable the Generate MDC command with a Trusted Key Entry
(TKE) workstation before using this verb.

**Usage notes**

The usage notes for CSNBMDG.

None

**Related information**

Additional information about CSNBMDG.

In releases before Release 3.30, it was discovered that the MDC Generate verb
produced incorrect MDC values under certain conditions. If you have any MDC
values generated using Release 3.30.04 or earlier, read this section to determine
what conditions produce incorrect MDC values. If necessary, take corrective action
as described below.

**Audience**

If you are an IBM System i®, System p®, or System x® customer of the IBM CCA
Support Program who generated MDC values using the MDC Generate
(CSNBMDG) verb with Release 3.30.04 or earlier, read the following important
information related to the integrity of your data.

**Overview**

This section describes in detail each scenario that results in incorrect MDC values.

It was discovered that under certain conditions, the MDC Generate verb of the
CCA Support Program generates incorrect MDC values.

**Terminology**

The terminology described in this topic is used to describe the conditions that
produce incorrect MDC values.

**Total text length (TTL)**

The total number of text bytes processed to calculate a final MDC value

**Running text length (RTL)**

The total number of text bytes processed by all previous calls used to
calculate a final MDC value

**Carryover length (COL)**

The number of text bytes that could not be processed in the previous call.
The COL can range from 0 - 15 bytes, and is stored in the chaining vector
between calls.
New text length (NTL)
The carryover length plus the text length for the current call

Note:
1. An intermediate MDC calculation always operates on eight bytes of text at a
time. Any remaining text that is not a multiple of eight bytes gets passed in the
chaining vector as carryover text.
2. A call with keyword \textbf{FIRST} must have a text length greater than or equal to 16
in order to calculate an intermediate MDC value. If the text length is greater
than or equal to 16, the COL is calculated as text length modulo 8, otherwise
the COL equals the text length. Any carryover text bytes get passed in the
chaining vector as carryover text to the next segment call.
3. A call with keyword \textbf{MIDDLE} calculates an intermediate MDC value if the text
bytes to process (COL plus text length) are greater than or equal to 16. If COL
plus text length is less than 16, the text bytes are carried over to the next call in
the chaining vector. \textbf{MIDDLE} calls process text in multiples of 8 (for example,
16, 24, 32). As with \textbf{FIRST}, the remaining text bytes (NTL modulo 8) get passed
in the chaining vector as carryover text to the next segment call.
4. An MDC value is final when calculated by keywords \textbf{ONLY} or \textbf{LAST}.

Examples:

Examples that produce MDC values.

1. Assume a text length of 19 for \textbf{FIRST}, 6 for \textbf{MIDDLE}, and 10 for \textbf{LAST}.

   \[
   \text{TTL} = 19 + 6 + 10 = 35 \text{ bytes.}
   \]
   When \textbf{FIRST} is called, 16 of the 19 text bytes will be processed to produce an
   intermediate MDC. The remaining 19 - 16 = 3 text bytes will be placed in the
   chaining vector.
   When \textbf{MIDDLE} is called,RTL = 19, COL = 19 - 16 = 3, and NTL = COL + text
   length = 3 + 6 = 9 bytes to process.
   After the \textbf{MIDDLE} call completes,RTL = 19 + 6 = 25 and COL = 25 - 16 = 9.
   Because 16 bytes are not available to be processed, the 9 text bytes will be
   placed in the chaining vector as carryover text.
   \textbf{LAST} will process COL + text length = 9 + 10 = 19 bytes. The NTL for the
   LAST call is 19 bytes. If the TTL is not a multiple of 8, use of a \textbf{PADMDC-2} or
   \textbf{PADMDC-4} method is required.

2. Assume a text length of 19 for \textbf{FIRST}, 25 for \textbf{MIDDLE}, and 12 for \textbf{LAST}.

   \[
   \text{TTL} = 19 + 25 + 12 = 56 \text{ bytes.}
   \]
   When \textbf{FIRST} is called, 16 of the 19 text bytes will be processed to produce an
   intermediate MDC. The remaining 19 - 16 = 3 text bytes will be placed in the
   chaining vector.
   When \textbf{MIDDLE} is called,RTL = 19, COL = 19 - 16 = 3, and NTL = COL + text
   length = 3 + 25 = 28 bytes to process.
   After the \textbf{MIDDLE} call completes,RTL = 19 + 25 = 44, COL = 28 modulo 8 = 4,
   and 28 - 4 = 24 bytes will be used to produce an intermediate MDC. The 4 text
   bytes will be placed in the chaining vector as carryover text.
   \textbf{LAST} will process COL + text length = 4 + 12 = 16 bytes. The NTL for the
   LAST call is 16 bytes. These text bytes will be used to produce the final MDC
   value.
Pre-Release 3.30 problems:

The following scenarios describe different situations where the MDC Generate (CSNBMDG) verb was found to produce incorrect MDC values.

These scenarios apply to releases prior to Release 3.30.

**Scenario 1 (pre-Release 3.30)**

Error type: Segmentation error, not padding related.

Keywords affected:
- Algorithm MDC-2, MDC-4, PADMDC-2, PADMDC-4
- Segmenting and key control MIDDLE

The MDC value is calculated incorrectly whenever:
- RTL is greater than or equal to 16 and
- At least one MIDDLE segment is processed that has COL plus text length less than 16.

Under the above conditions, the very next MIDDLE or LAST call loses the intermediate MDC value that was passed on input.

**WARNING:** The integrity of any data processed up to the time that the intermediate MDC value is lost cannot be confirmed.

Example:

MDC-2, MDC-4, PADMDC-2, or PADMDC-4

FIRST text length = 19, MIDDLE text length = 6, LAST text length greater than or equal to 0, an incorrect MDC value is calculated.

**Corrective action:**

None. The intermediate MDC value calculated when keyword FIRST was used to process the 16 bytes of text is lost. The integrity of these first 16 bytes of data cannot be confirmed.

**Scenario 2 (pre-Release 3.30)**

Error type: Padding error related to segmenting.

Keywords affected:
- Algorithm PADMDC-2, PADMDC-4
- Segmenting and key control LAST

The MDC value is calculated incorrectly whenever:
- LAST text length is equal to 0 and
- TTL is greater than or equal to 16 and
- TTL modulo 8 is equal to 0

Under the above conditions, 16 bytes of padding are incorrectly added to the text instead of the required eight bytes of padding.

Example:

PADMDC-2 or PADMDC-4

FIRST text length = 16, LAST text length = 0, an incorrect MDC value is calculated.

**Corrective action:**
Prior to migrating to Release 3.30.05 or later, recalculate each MDC value in the same manner used to calculate the existing MDC value. If the newly calculated MDC value matches the older MDC value, data integrity is confirmed. After all MDC values have been confirmed, migrate to the latest release and recalculate each MDC value. Replace the old MDC values with the new ones.

**Scenario 3 (pre-Release 3.30)**
Error type: Padding error, not segmenting related.

Keywords affected:
- Algorithm **PADMDC-2, PADMDC-4**
- Segmenting and key control **ONLY, LAST**

The MDC value is calculated incorrectly whenever:
- TTL is greater than or equal to 16 and
- TTL modulo 8 is equal to 0

Under the above conditions, no padding is added to the text as required. The incorrect MDC value is identical to calling either **MDC-2** or **MDC-4**.

Example:
**PADMDC-2 or PADMDC-4**

**ONLY** text length is equal to 16, 24, 32, and so forth, an incorrect MDC value is calculated. The MDC value is calculated without adding the required eight bytes of pad characters.

Corrective action:
Prior to migrating to Release 3.30.05 or later, recalculate each MDC value in the same manner used to calculate the existing MDC value. If the newly calculated MDC value matches the older MDC value, data integrity is confirmed. After all MDC values have been confirmed, migrate to the latest release and recalculate each MDC value. Replace the old MDC values with the new ones.

**Scenario 4 (pre-Release 3.30)**
Error type: Padding error related to segmenting.

Keywords affected:
- Algorithm **PADMDC-2, PADMDC-4**
- Segmenting and key control **FIRST, MIDDLE**

The MDC value is calculated incorrectly whenever:
- RTL is greater than or equal to 16 and
- **LAST** is called with COL plus text length greater than zero and less than 8

Under the above conditions, the text is padded with eight bytes more than is required.

Example:
**PADMDC-2 or PADMDC-4**

**FIRST text length = 16, LAST = 7**, an incorrect MDC value is calculated. The MDC value is calculated with 9 pad bytes instead of the required 1 pad byte.

Corrective action:
Prior to migrating to Release 3.30.05 or later, recalculate each MDC value in the same manner used to calculate the existing MDC value. If the newly calculated MDC value matches the older MDC value, data integrity is confirmed. After all MDC values have been confirmed, migrate to the latest release and recalculate each MDC value. Replace the old MDC values with the new ones.

**Scenario 5 (pre-Release 3.30)**

Error type: Segmenting error, not padding related.

Keywords affected:
- Algorithm MDC-2, MDC-4, PADMDC-2, PADMDC-4
- Segmenting and key control MIDDLE without FIRST

The MDC value is calculated incorrectly whenever:
- **FIRST** is not called
- For the first MIDDLE call only, text length is less than 16
- The chaining vector is set to zero
- The MDC value on input is set to a keyed hash value not equal to the default key

Under the above conditions, the keyed hash value that the caller set in the MDC is ignored and the MDC value is incorrectly calculated using the default key.

**Example:**

MDC-2, MDC-4, PADMDC-2, or PADMDC-4

Chaining vector is set to hex zeros.

MDC value is set to a non-default key value (default key = X'5252525252525252 2525252525252525').

MIDDLE text length = 8, LAST text length = 16, an incorrect MDC value is calculated. The MDC value is calculated with the default key and not with the key value of the MDC parameter.

**Corrective action:**

Prior to migrating to Release 3.30.05 or later, recalculate each MDC value in the same manner used to calculate the existing MDC value. If the newly calculated MDC value matches the older MDC value, data integrity is confirmed. After all MDC values have been confirmed, migrate to the latest release and recalculate each MDC value. Replace the old MDC values with the new ones.

**Release 3.30.04 only problems:**

The following scenarios describe different situations where the MDC Generate (CSNBMDG) verb was found to produce incorrect MDC values.

These scenarios apply to Release 3.30.04 only.

**Scenario 1 (Release 3.30 only)**

Error type: Segmentation error, not padding related.

Keywords affected:
- Algorithm MDC-2, MDC-4, PADMDC-2, PADMDC-4
- Segmenting and key control MIDDLE
The MDC value is calculated incorrectly whenever:

- RTL is greater than or equal to 16 and
- MIDDLE is called with COL plus text length less than 16 and
- MIDDLE is called again

Under the above conditions, the first MIDDLE call causes a subsequent MIDDLE call to lose the intermediate MDC value passed to it on input.

**WARNING:** The integrity of any data processed up to the time that the intermediate MDC value is lost cannot be confirmed.

**Example:**

**MDC-2, MDC-4, PADMDC-2, or PADMDC-4**

FIRST text length = 19, MIDDLE text length = 6, subsequent MIDDLE and LAST text length greater than or equal to 0, an incorrect MDC value is calculated. The intermediate MDC value calculated when FIRST processed the 16 bytes of text is lost.

**Corrective action:**

None. The intermediate MDC value calculated when keyword FIRST was used to process the 16 bytes of text is lost. The integrity of these first 16 bytes of data cannot be confirmed.

**Scenario 2 (Release 3.30 only)**

Error type: Padding error related to segmenting.

Keywords affected:

- Algorithm PADMDC-2, PADMDC-4
- Segmenting and key control LAST

The MDC value is calculated incorrectly whenever:

- LAST text length is equal to 0 and
- TTL is greater than or equal to 16 and
- TTL modulo 8 is equal to 0

Under the above conditions, 16 bytes of padding are added to the text instead of the required eight bytes of padding.

**Example:**

**PAINDMC-2 or PADMDC-4**

FIRST text length = 16, LAST text length = 0, an incorrect MDC value is calculated.

**Corrective action:**

Prior to migrating to Release 3.30.05 or later, recalculate each MDC value in the same manner used to calculate the existing MDC value. If the newly calculated MDC value matches the older MDC value, data integrity is confirmed. After all MDC values have been confirmed, migrate to the latest release and recalculate each MDC value. Replace the old MDC values with the new ones.

**Scenario 3 (Release 3.30 only)**

Error type: Padding error related to segmenting.

Keywords affected:

- Algorithm PADMDC-2, PADMDC-4
MDC Generate (CSNBMDG)

- Segmenting and key control

The MDC value is calculated incorrectly whenever:
- TTL is greater than zero and less than 8

Under the above conditions, the text is incorrectly padded with 8 pad bytes less than required.

Example:

**PADMDC-2 or PADMDC-4**

**LAST** text length = 7, an incorrect MDC value is calculated. The MDC value is calculated with only one pad byte instead of the required 9 pad bytes.

**Corrective action:**

Prior to migrating to Release 3.30.05 or later, recalculate each MDC value in the same manner used to calculate the existing MDC value. If the newly calculated MDC value matches the older MDC value, data integrity is confirmed. After all MDC values have been confirmed, migrate to the latest release and recalculate each MDC value. Replace the old MDC values with the new ones.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBMDGJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBMDGJ are shown here.

**Format**

```java
public native void CSNBMDGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger text_length,
    byte[] text_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] chaining_vector,
    byte[] MDC);
```
One-Way Hash (CSNBOWH)

Use the One-Way Hash verb to generate a one-way hash on specified text.

These SHA based hashing functions are supported with the CPACF exploitation layer: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. For details about CPACF, see "CPACF support" on page 12.

Format

The format of CSNBOWH.

```
CSNBOWH(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    text_length,
    text,
    chaining_vector_length,
    chaining_vector,
    hash_length,
    hash)
```

Parameters

The parameters for CSNBOWH.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1 or 2.

`rule_array`

Direction: Input
Type: String

These keywords provide control information to the verb. The optional chaining flag keyword indicates whether calls to this verb are chained together logically to overcome buffer size limitations. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 76.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash method</td>
<td>(One, required). The SHA-based hashing functions use CPACF by default. For details about CPACF, see &quot;CPACF support&quot; on page 12.</td>
</tr>
<tr>
<td>MD5</td>
<td>Hash algorithm is MD5 algorithm. Use this hash method for PKCS-1.0 and PKCS-1.1. Length of hash generated is 16 bytes.</td>
</tr>
<tr>
<td>RPMD-160</td>
<td>Hash algorithm is RIPEMD-160. Length of hash generated is 20 bytes.</td>
</tr>
</tbody>
</table>
### Table 76. Keywords for One-Way Hash control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>Hash algorithm is SHA-1 algorithm. Length of hash generated is 20 bytes.</td>
</tr>
<tr>
<td>SHA-224</td>
<td>Hash algorithm is SHA-224 algorithm. Length of hash generated is 20 bytes.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Hash algorithm is SHA-256 algorithm. Length of hash generated is 20 bytes.</td>
</tr>
<tr>
<td>SHA-384</td>
<td>Hash algorithm is SHA-384 algorithm. Length of hash generated is 20 bytes.</td>
</tr>
<tr>
<td>SHA-512</td>
<td>Hash algorithm is SHA-512 algorithm. Length of hash generated is 20 bytes.</td>
</tr>
</tbody>
</table>

**Chaining flag (One, optional)**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>Specifies this is the first call in a series of chained calls. Intermediate results are stored in the hash field.</td>
</tr>
<tr>
<td>LAST</td>
<td>Specifies this is the last call in a series of chained calls.</td>
</tr>
<tr>
<td>MIDDLE</td>
<td>Specifies this is a middle call in a series of chained calls. Intermediate results are stored in the hash field.</td>
</tr>
<tr>
<td>ONLY</td>
<td>Specifies this is the only call and the call is not chained. This is the default.</td>
</tr>
</tbody>
</table>

**text_length**

- **Direction:** Input
- **Type:** Integer

The length of the `text` parameter in bytes.

**Note:** If you specify the `FIRST` or `MIDDLE` keyword, the text length must be a multiple of the block size of the hash method. For MD5, RPMD-160, and SHA-1, this is a multiple of 64 bytes.

For `ONLY` and `LAST`, this verb performs the required padding according to the algorithm specified.

**text**

- **Direction:** Input
- **Type:** String

The application-supplied text on which this verb performs the hash.

**chaining_vector_length**

- **Direction:** Input
- **Type:** Integer

The byte length of the `chaining_vector` parameter. This must be 128 bytes.

**chaining_vector**

- **Direction:** Input/Output
- **Type:** String

This field is a 128-byte work area. Your application must not change the data in this string. The chaining vector permits chaining data from one call to another.

**hash_length**

- **Direction:** Input
- **Type:** Integer

The length of the supplied `hash` field in bytes.
One-Way Hash (CSNBOWH)

Note: For SHA-1 and RPMD-160 this must be a minimum of 20 bytes. For MD5 this must be a minimum of 16 bytes.

hash

Direction: Input/Output
Type: String

This field contains the hash, left-aligned. The processing of the rest of the field depends on the implementation. If you specify the FIRST or MIDDLE keyword, this field contains the intermediate hash value. Your application must not change the data in this field between the sequence of FIRST, MIDDLE, and LAST calls for a specific message.

Restrictions
The restrictions for CSNBOWH.

None

Required commands
The required commands for CSNBOWH.

None

Usage notes
The usage notes for CSNBOWH.

Although the algorithms accept zero bit length text, it is not supported for any hashing method.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBOWHJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBOWHJ are shown here.

Format
public native void CSNBOWHJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger text_length,
    byte[] text,
    hikmNativeInteger chaining_vector_length,
    byte[] chaining_vector,
    hikmNativeInteger hash_length,
    byte[] hash);
Chapter 9. Key storage mechanisms

This section describes how you can use key storage mechanisms and the associated key record verbs to perform operations on key tokens and key records located in AES, DES, and PKA key storage.

A key-token record consists of a key-token name (key label) and a key token of format null, internal, or external. The operations to be performed are: creating, writing, reading, listing, and deleting key tokens or key records.

The verbs described in this chapter include:
- “AES Key Record Create (CSNBAKRC)” on page 352
- “AES Key Record Delete (CSNBAKRD)” on page 355
- “AES Key Record List (CSNBAKRL)” on page 357
- “AES Key Record Read (CSNBAKRR)” on page 360
- “AES Key Record Write (CSNBAKRW)” on page 362
- “DES Key Record Create (CSNBKRC)” on page 365
- “DES Key Record Delete (CSNBKRD)” on page 367
- “DES Key Record List (CSNBKRL)” on page 369
- “DES Key Record Read (CSNBKRR)” on page 372
- “DES Key Record Write (CSNBKRW)” on page 374
- “PKA Key Record Create (CSNDKRC)” on page 376
- “PKA Key Record Delete (CSNDKRD)” on page 378
- “PKA Key Record List (CSNDKRL)” on page 380
- “PKA Key Record Read (CSNDKRR)” on page 383
- “PKA Key Record Write (CSNDKRW)” on page 385
- “Retained Key Delete (CSNDRKD)” on page 387
- “Retained Key List (CSNDRKL)” on page 390

Key labels and key-storage management

Use the verbs described in this section to manage AES, DES, and PKA key storage.

The CCA software manages key storage as an indexed repository of key records. Access key storage using a key label with verbs that have a key-label or key-identifier parameter.

An independent key-storage system can be used to manage records for AES key records, DES key records, and PKA key records:

**AES key storage**
- Holds null and internal AES key tokens

**DES key storage**
- Holds null, external, and internal DES key tokens

**PKA key storage**
- Holds null PKA key tokens, and both internal and external public and private PKA key tokens
Private RSA keys are generated and optionally retained within the coprocessor using the PKA Key Generate verb. Depending on the other uses for coprocessor storage, between 75 and 150 keys can normally be retained within the coprocessor.

Key storage must be initialized before any records are created. Before a key token can be stored in key storage, a key-storage record must be created using the AES Key Record Create, DES Key Record Create, or PKA Key Record Create verb.

Use the AES Key Record Delete, DES Key Record Delete, or PKA Key Record Delete verb to delete a key token from a key record, or to entirely delete the key record from key storage.

Use the AES Key Record List, DES Key Record List, or PKA Key Record List verb to determine the existence of key records in key storage. These list verbs create a key-record-list file with information about select key records. The wildcard character, represented by an asterisk (*), is used to obtain information about multiple key records. The file can be read using conventional workstation-data-management services.

Individual key tokens can be read using the AES Key Record Read, DES Key Record Read, and PKA Key Record Read verbs or written using the AES Key Record Write, DES Key Record Write, and PKA Key Record Write verbs.

**Environment variables for the key storage file**

These environment variables contain the name of the key storage file.

There is one for each type: AES, DES, and PKA.

**CSUAESDS**

AES key storage file.

**CSUDESDS**

DES key storage file.

**CSUPKADS**

PKA key storage file.

See also "Dual Support: Key storage interactions" on page 770.

**Key-label content**

Use a key label to identify a record in key storage managed by a CCA implementation.

The key label must be left-aligned in the 64-byte string variable used as input to the verb. Some verbs use a key label while others use a key identifier. Calls that use a key identifier accept either a key token or a key label.

A key-label character string has the following properties:

- It contains 64 bytes of data.
- The first character is within the range X'20' - X'FE'. If the first character is within this range, the input is treated as a key label, even if it is otherwise not valid. Inputs beginning with a byte valued in the range X'00' - X'1F' are considered to be some form of key token. A first byte valued to X'FF' is not valid.
- The first character of the key label cannot be numeric (0 - 9).
• The label is ended by a space character on the right (in ASCII it is X'20', and in EBCDIC it is X'40'). The remainder of the 64-byte field is padded with space characters.

• Construct a label with 1 - 7 name tokens, each separated by a period (.). The key label must not end with a period.

• A name token consists of 1 - 8 characters in the character set A - Z, 0 - 9, and three additional characters relating to different character symbols in the various national language character sets as listed in Table 77.

Table 77. Valid symbols for the name token

<table>
<thead>
<tr>
<th>ASCII systems</th>
<th>EBCDIC systems</th>
<th>USA graphic (for reference)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'23'</td>
<td>X'7B'</td>
<td>#</td>
</tr>
<tr>
<td>X'24'</td>
<td>X'5B'</td>
<td>$</td>
</tr>
<tr>
<td>X'40'</td>
<td>X'7C'</td>
<td>@</td>
</tr>
</tbody>
</table>

The alphabetic and numeric characters and the period should be encoded in the normal character set for the computing platform that is in use, either ASCII or EBCDIC.

Note:
1. Some CCA implementations accept the characters a - z and fold these to their uppercase equivalents, A - Z. For compatibility reasons, only use the uppercase alphabetic characters.
2. Some implementations internally transform the EBCDIC encoding of a key label to an ASCII string. Also, the label might be put in tokenized form by dropping the periods and formatting each name token into 8-byte groups, padded on the right with space characters.

Some verbs accept a key label containing a wild card represented by an asterisk (*). (X'2A' in ASCII; X'5C' in EBCDIC). When a verb permits the use of a wild card, the wild card can appear as the first character, as the last character, or as the only character in a name token. Any of the name tokens can contain a wild card.

Examples of valid key labels include the following:

- A
- ABCD.2.3.4.5555
- ABCDEFGH
- BANKSYS.XXXXX.43*.PDQ

Examples of key labels that are not valid are listed in Table 78.

Table 78. Key labels that are not valid

<table>
<thead>
<tr>
<th>Key label not valid</th>
<th>Problem with key label</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/.B</td>
<td>A slash is an unacceptable character</td>
</tr>
<tr>
<td>ABCDEFGH9</td>
<td>Name token is greater than 8 characters</td>
</tr>
<tr>
<td>1111111.2.3.4.55555</td>
<td>First character cannot be numeric</td>
</tr>
<tr>
<td>A111111.2.3.4.55555.6.7.8</td>
<td>Number of name tokens exceeds 7</td>
</tr>
<tr>
<td>BANKSYS.XXXXX.43*.D</td>
<td>Number of wild cards exceeds 1</td>
</tr>
<tr>
<td>A.B.</td>
<td>Last character cannot be a period</td>
</tr>
</tbody>
</table>
Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z

Key storage for IBM z/OS and for Linux on the IBM platforms other than IBM System z, diverged in design at their very inception.

Background information about master key management

There are four types (or sets) of master keys (Symmetric DES, AES, Asymmetric RSA (PKA), and APKA).

There are three master key registers for each of the four types of master key. In other words, there are a total of twelve master key registers.

The APKA master-key register set, introduced to CCA beginning with Release 4.1.0, is used to encrypt and decrypt the Object Protection Key (OPK) that is itself used to wrap the key material of an Elliptic Curve Cryptography (ECC) key. ECC keys are asymmetric.

For each of the four types, there is a master key register in one of these three categories:

New master-key (NMK) register
This register holds a master key that is not yet usable for decrypting key tokens for normal cryptographic operations.

The NMK register can be in one of these states:

**EMPTY**
No key parts have been loaded yet.

**PARTIALLY FULL**
Some key parts have been loaded, but not the LAST key part. See “Master Key Process (CSNBMKP)” on page 119.

**FULL**
The LAST key part has been loaded, but the SET command has not yet been called. See “Master Key Process (CSNBMKP)” on page 119.

Current master-key (CMK) register
This register holds a master key that can be used to decrypt internal key tokens for keys in use with normal cryptographic operations. Internal key tokens are protected under the master key; the keys are actually stored outside the adapter.

The CMK register can be in one of these states:

**EMPTY**
No valid key has yet been established with the SET command in the life of this adapter, or the adapter has been re-initialized to clear the master key registers.

**VALID**
A master key has been loaded with the SET command.

Old master-key (OMK) register
This is the master key that previously has been the CMK, before the master key that is now in the CMK register. The OMK register can also be used to decrypt internal key tokens, but for these keys a warning with return code 0 and reason code 2 is returned, along with the results from the requested cryptographic operation.
The OMK register can be in one of these states:

**EMPTY**
No valid key is in this register.

**VALID**
A master key that previously was in the CMK register has been shifted to the OMK register by the SET command. The same invocation of the SET command also shifted the contents of the NMK register into the CMK register.

### SET command

The SET command performs the operations described in this topic.

The SET command is invoked with the Master Key Process (CSNBMKP) on page 119, and performs these commands:

1. The master key from the CMK register is copied to the OMK register.
2. The master key from the FULL NMK register is copied to the CMK register.
3. The NMK register status is changed to EMPTY.

### Key Storage on z/OS (RTNMK-focused)

*Design point* - Keys should be re-enciphered to a master key in the NMK register.

This forces the following process to be followed when changing the master key:

- Load all the master key parts for a NMK, such that the LAST key part has been loaded, but the SET command has not been issued. Now the NMK register is in the FULL state.
- Re-encipher all of (for example: CKDS) an existing key storage to a copy of that key storage that is not online, using the RTNMK rule_array keyword of "Key Token Change (CSNBKTC)” on page 218 (for AES or DES) or "PKA Key Token Change (CSNDKTC)” on page 497 (for PKA), creating CKDS-pending. Keys in this copy are enciphered under the NMK register, and so are not usable for normal cryptographic operations.
- Invoke the SET command for the NMK. See "SET command." Now the master keys in the current CKDS are enciphered under the OMK (because of the shift), and are usable. Also, the master keys in the CKDS-pending are also usable because the NMK has now become the CMK.
- Delete the old CKDS and change CKDS-pending to be the normal CKDS, completing the process.

### Key Storage for traditional IBM systems other than IBM System z (RTCMK-focused: Linux, AIX, Windows)

*Design point* - Keys should be re-enciphered to a master key in the CMK register.

This forces the following process to be followed when changing the master key:

- Load all the master key parts for a NMK, such that the LAST key part has been loaded, then issue the SET command. Now the previous OMK is gone, the previous CMK is now the OMK, and the CMK contains the newly-loaded value. See "SET command."
- Re-encipher all of an existing CCA host key storage data file’s key tokens, which are enciphered under the OMK, to be enciphered under the CMK. This is done using the RTCMK rule_array keyword of “Key Token Change (CSNBKTC)” on page 218 or “PKA Key Token Change (CSNDKTC)” on page 497.
- This **immediately replaces operational keys** with the re-enciphered version.
- The CCA key storage file has a data structure with the verification pattern of the most recently SET master key. The key storage implementation also allows writing external tokens into the key storage. This means that external key tokens, and the internal key tokens encrypted under current master key, will be allowed into the key storage.

It is impossible with current implementation to use RTNMK together with CCA key storage.

• During the re-encipherment:
  - Some of the keys in the CCA key storage files are enciphered under the OMK (because of the shift) and are usable
  - Some of the keys in the CCA key storage files are enciphered under the CMK, either because they are new or because they have been re-enciphered.
  - No new key tokens can be created with the key wrapped using the OMK.

Both types are usable for cryptographic operations.

**Changing the master key for two or more adapters that have the same master key, with shared CCA key storage**

Because the verification pattern of the CMK is stored in a header in key storage, changing the master key for a configuration of multiple adapters requires extra care.

The master key verification pattern in key storage has the following properties:

• It is checked once when a process starts.

• It is repopulated when the first CEX*C has its master key changed.

These two properties force the user to use the same process to change the master key for all CEX*Cs after the first CEX*C. If the process exits (such as when the application completes), then the next time that the application starts the key storage header will be checked and the master key verification pattern will reflect the newly SET master key, which will cause a future attempt to set that same master key to a second or third CEX*C to have a conflict with the key storage header. Therefore, using the same process to change the master keys in all the CEX*C adapters is the only way to proceed if CCA key storage is being used.

There are several ways to change the master keys, most of which do not suffer from this limitation:

• A TKE can be used to change the master keys for all the CEX*C adapters in a group.

• An operator can directly change the master keys for a domain on a CEX*C from an IBM System z management interface (physical access is needed).

• A user application built to use the **libcsulccamk.so** library for this purpose, which can be programmed to:
  1. **Allocate a CEX*C by invoking [Cryptographic Resource Allocate (CSUACRA)](page 110)”.**
  2. Change the master key.
  3. **Deallocate each adapter in the group before exiting, by invoking [Cryptographic Resource Deallocate (CSUACRD)](page 113)”.**

• Note that the included utility, named panel.exe, is not designed to change the master keys for all the cards in a group; this is a more sophisticated operation. For details about panel.exe, see [“The panel.exe utility” on page 773].
Key storage file ownership

The last user to access the key storage file owns it, due to the internals of the key storage functions.

The file is recreated after being compressed, and due to the file creation the owner is changed.

Having the set-group-id bit (g+s) on in the directory permission causes the file to be created with the group owner the same as the directory group owner. The group read/write permissions on the file then allow the other members of the group continued access to the file.

The Linux on IBM System z approach

Because the CCA key storage design point for the Linux platform host release has always been CMK-focused, this design point was taken forward for the Linux on IBM System z approach.

At this time, CCA host key storage does not support nor ship with an additional utility to manage the 'store-in-pending' approach to re-enciphering key tokens. This additional utility is necessary to work with use of the RTNPK keyword for "Key Token Change (CSNBKTC)" on page 218 and "PKA Key Token Change (CSNDKTC)" on page 497. Therefore, it is suggested that users wanting to make use of CCA host key storage management follow the 'RTCMK-focused' approach described in "Key Storage for traditional IBM systems other than IBM System z (RTCMK-focused: Linux, AIX, Windows)" on page 349.

However it is also desirable to provide as much host-support equivalence with the z/OS approach as possible, given that the underlying system is running on a System z platform and likely to collaborate with z/OS software. Therefore, the RTNPK keyword is provided for "Key Token Change (CSNBKTC)" on page 218 and "PKA Key Token Change (CSNDKTC)" on page 497 to allow users who have their own utility or key storage management facility to manage key tokens using the method most familiar from z/OS:

- The key tokens to be enciphered should be passed directly (not by label) to "Key Token Change (CSNBKTC)" on page 218 and "PKA Key Token Change (CSNDKTC)" on page 497 for re-encipherment, and stored outside CCA host key storage.
- When re-encipherment is complete and the "Master Key Process (CSNMKP)" on page 119 SET command has been issued, the re-enciphered key tokens can be reintroduced to CCA host key storage if desired, using the standard mechanisms.
AES Key Record Create (CSNBAKRC)

Use the AES Key Record Create verb to create a key-token record in AES key-storage.

The new key record can be a null AES key-token or a valid internal AES key-token. It is identified by the key label specified with the key_label parameter.

After creating an AES key-record, use any of the following verbs to add or update a key token in the record:
- AES Key Record Delete
- AES Key Record Write
- Key Generate
- Key Token Change
- Key Token Change2
- Symmetric Key Generate
- Symmetric Key Import
- Symmetric Key Import2

Note:
1. To delete a key record from AES key-storage, use the AES Key Record Delete verb.
2. AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

Format

The format of CSNBAKRC.

```c
CSNBAKRC (  
    return_code,  
    reason_code,  
    exit_data_length,  
    exit_data,  
    rule_array_count,  
    rule_array_count,  
    key_label,  
    key_token_length,  
    key_token)
```

Parameters

The parameters for CSNBAKRC.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0.

`rule_array`
AES Key Record Create (CSNBAKRC)

Direction: Input
   Type: Array
   This parameter is ignored.

key_label
   Direction: Input
   Type: String
   A pointer to a string variable containing the key label of the AES key-record to be created.

key_token_length
   Direction: Input
   Type: Integer
   A pointer to an integer variable containing the number of bytes of data in the key_token variable. If the value of the key_token_length variable is zero, a record with a null AES key-token is created.

key_token
   Direction: Input
   Type: String
   A pointer to a string variable containing the key token being written to AES key-storage.

Restrictions
   The restrictions for CSNBAKRC.

   The record must have a unique label. Therefore, there cannot be another record in the AES key storage file with the same label and a different key type.

Required commands
   The required commands for CSNBAKRC.

   This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
   The usage notes for CSNBAKRC.

   None

Related information
   Additional information about CSNBAKRC.

   See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

JNI version
   This verb has a Java Native Interface (JNI) version, which is named CSNBAKRCJ.

   See "Building Java applications to use with the CCA JNI" on page 26.
AES Key Record Create (CSNBAKRC)

The parameters for CSNBAKRCJ are shown here.

**Format**

```java
public native void CSNBAKRCJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label,
    hikmNativeInteger key_token_length,
    byte[] key_token);
```
AES Key Record Delete (CSNBAKRD)

Use the AES Key Record Delete verb to perform one of the tasks listed in this topic in the AES key storage file.

- Overwrite (delete) a key token or key tokens in AES key-storage, replacing the key token of each selected record with a null AES key-token.
- Delete an entire key record or key records, including the key label and the key token of each selected record, from AES key-storage.

Identify a task with the `rule_array` keyword, and the key record or records with the `key_label` parameter. To identify multiple records, use a wild card (*) in the key label.

Note: AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

**Format**

The format of CSNBAKRD.

```c
CSNBAKRD (    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_label)
```

**Parameters**

The parameters for CSNBAKRD.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see [Parameters common to all verbs](#) on page 20.

- **rule_array_count**
  
  **Direction:** Input
  
  **Type:** Integer
  
  A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0 or 1.

- **rule_array**
  
  **Direction:** Input
  
  **Type:** String
  
  A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords are described in Table 79.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task (One, optional)</td>
<td>Deletes a key token from a key record in AES key storage. This is the default.</td>
</tr>
</tbody>
</table>

Table 79. Keywords for AES Key Record Delete control information
AES Key Record Delete (CSNBAKRD)

Table 79. Keywords for AES Key Record Delete control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL-DL</td>
<td>Deletes an entire key record, including the key label, from AES key storage.</td>
</tr>
</tbody>
</table>

**key_label**

Direction: Input  
Type: String

A pointer to a string variable containing the key label of a key-token record or records in AES key-storage. Use a wild card (*) in the `key_label` variable to identify multiple records in key storage.

**Restrictions**

The restrictions for CSNBAKRD.

The record defined by the `key_label` must be unique.

**Required commands**

The required commands for CSNBAKRD.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBAKRD.

None

**Related information**

Additional information about CSNBAKRD.

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRDJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBAKRDJ are shown here.

**Format**

```java
public native void CSNBAKRDJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_identifier);
```
The AES Key Record List verb creates a key-record-list file containing information about specified key records in key storage.

Information listed includes whether record validation is correct, the type of key, and the date and time the record was created and last updated.

Specify the key records to be listed using the key-label variable. To identify multiple key records, use the wild card (*) in the key label.

**Note:**
1. To list all the labels in key storage, specify the key_label parameter with *, *, *, and so forth, up to a maximum of seven name tokens (**.***.***).
2. AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

This verb creates the key-record-list file and returns the name of the file and the length of the file name to the calling application. This file has a header record, followed by 0 - n detail records, where n is the number of key records with matching key-labels.

The file is kept in the /opt/IBM/CCA/keys/deslist directory (assuming the directory name was not changed during installation). These list files are created under the ownership of the environment of the user that requests the list service. Make sure the files created kept the same group ID as your installation requires. This can also be achieved by setting the 'set-group-id-on-execution' bit on in this directory. See the g+s flags in the chmod command for full details. Not doing this might cause errors to be returned on key-record-list verbs.

**Format**

The format of CSNBAKRL.

```plaintext
CSNBAKRL(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_label,
    dataset_name_length,
    dataset_name,
    security_server_name)
```

**Parameters**

The parameters for CSNBAKRL.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

Direction: Input
Type: Integer
AES Key Record List (CSNBAKRL)

A pointer to an integer variable containing the number of elements in the
rule_array variable. This value must be 0.

rule_array
Direction: Input
Type: Array
A pointer to a string variable containing an array of keywords. This verb
currently does not use keywords.

key_label
Direction: Input
Type: String
A pointer to a string variable containing the key label of a key-token record in
key storage. In a key label, you can use a wild card (*) to identify multiple
records in key storage.

dataset_name_length
Direction: Output
Type: Integer
A pointer to an integer variable containing the number of bytes of data
returned by the verb in the dataset_name variable. The maximum returned
length is 64 bytes.

dataset_name
Direction: Output
Type: String
A pointer to a string variable containing the name of the file returned by the
verb. The file contains the AES key-record information. When the verb stores a
key-record-list file, it overlays any older file with the same name.

The file name returned by this verb is defined by the CSUAESLD environment
variable.

This verb returns the file name as a fully qualified file specification (for
example, /opt/IBM/CCA/keys/KYRLTnnn.LST), where nnn is the numeric
portion of the name. This value increases by one every time that you use this
verb. When this value reaches 999, it resets to 001.

security_server_name
Direction: Output
Type: String
A pointer to a string variable. The information in this variable is not currently
used, but the variable must be declared.

Restrictions
The restrictions for CSNBAKRL.

None

Required commands
The required commands for CSNBAKRL.
AES Key Record List (CSNBAKRL)

This verb requires the Key Test and Key Test2 command (offset X’001D’) to be enabled in the active role.

Usage notes
The usage notes for CSNBAKRL.

None

Related information
Additional information about CSNBAKRL.

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBAKRLJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBAKRLJ are shown here.

Format
public native void CSNBAKRLJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label,
    hikmNativeInteger data_set_name_length,
    byte[] data_set_name,
    byte[] security_server_name);
AES Key Record Read (CSNBAKRR)

Use the AES Key Record Read verb to read a key-token record from AES key-storage and return a copy of the key token to application storage.

The returned key token can be null. In this event, the key_length variable contains a value of 64 and the key-token variable contains 64 bytes of X'00' beginning at offset 0.

Note: AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

Format
The format of CSNBAKRR.

```c
CSNBAKRR (  
   return_code,  
   reason_code,  
   exit_data_length,  
   exit_data,  
   rule_array_count,  
   rule_array,  
   key_label,  
   key_token_length,  
   key_token)
```

Parameters
The parameters for CSNBAKRR.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

rule_array_count
Direction: Input
Type: Integer
A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0.

rule_array
Direction: Input
Type: Array
This parameter is ignored.

key_label
Direction: Input
Type: String
A pointer to a string variable containing the key label of the record to be read from AES key-storage.

key_token_length
Direction: Input/Output
Type: Integer
A pointer to an integer variable containing the number of bytes of data in the key_token variable. The maximum length is 64.
AES Key Record Read (CSNBAKRR)

**key_token**

*Direction: Output*

*Type: String*

A pointer to a string variable containing the key token read from AES key-storage. This variable must be large enough to hold the AES key token being read. On completion, the `key_token_length` variable contains the actual length of the token being returned.

**Restrictions**

The restrictions for CSNBAKRR.

None.

**Required commands**

The required commands for CSNBAKRR.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBAKRR.

None

**Related information**

Additional information about CSNBAKRR.

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRRJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBAKRRJ are shown here.

**Format**

```java
public native void CSNBAKRRJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label,
    hikmNativeInteger key_token_length,
    byte[] key_token);
```
AES Key Record Write (CSNBAKRW)

Use this verb to write a copy of an AES key-token from application storage into AES key-storage.

This verb can perform the following processing options:
- Write the new key-token only if the old token was null.
- Write the new key-token regardless of content of the old token.

AES key records are stored in the external key-storage file defined by the CSUAESDS environment variable.

**Note:** Before using this verb, use the verb “AES Key Record Create (CSNBAKRC)” on page 352 to create a key record in the key storage file.

**Format**

The format of CSNBAKRW.

```c
CSNBAKRW (
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_label,
    key_token_length,
    key_token)
```

**Parameters**

The parameters for CSNBAKRW.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`rule_array_count`
- Direction: Input
- Type: Integer
- A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0 or 1.

`rule_array`
- Direction: Input
- Type: Array
- A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters.

The `rule_array` keywords are described in Table 80.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing option</td>
<td>(One, optional)</td>
</tr>
</tbody>
</table>
Table 80. Keywords for AES Key Record Write control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECK</td>
<td>Specifies that the record is written only if a record of the same label in AES key-storage contains a null key-token. This is the default.</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>Specifies that the record is overwritten regardless of the current content of the record in AES key-storage.</td>
</tr>
</tbody>
</table>

**key_label**

Direction: Input  
Type: String  
A pointer to a string variable containing the key label that identifies the record in AES key-storage where the key token is to be written.

**key_token_length**

Direction: Input  
Type: Integer  
A pointer to an integer variable containing the number of bytes of data in the key_token variable. This value must be 64.

**key_token**

Direction: Input  
Type: String  
A pointer to a string variable containing the AES key-token to be written into AES key-storage.

**Restrictions**

The restrictions for CSNBAKRW.

The record defined by the key_label parameter must be unique and must already exist in the key storage file.

**Required commands**

The required commands for CSNBAKRW.

This verb requires the Key Test and Key Test2 command (offset X’001D’) to be enabled in the active role.

**Usage notes**

The usage notes for CSNBAKRW.

None

**Related information**

Additional information about CSNBAKRW.

You can use this verb with the key record create verb to write an initial record to key storage. Use it following the Key Import and Key Generate verb to write an operational key imported or generated by these verbs directly to the key storage file.
AES Key Record Write (CSNBAKRW)

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBAKRWJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBAKRWJ are shown here.

Format

```java
public native void CSNBAKRWJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label,
    hikmNativeInteger key_token_length,
    byte[] key_token);
```
DES Key Record Create (CSNBKRC)

Use the DES Key Record Create verb to add a key record to the DES key storage file.

The record contains a key token set to binary zeros and is identified by the label passed in the key_label parameter. The key label must be unique.

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Format
The format of CSNBKRC.

```c
CSNBKRC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_label)
```

Parameters
The parameters for CSNBKRC.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

key_label
Direction: Input  
Type: String

The 64-byte label of a record in the DES key storage file that is the target of this verb. The created record contains a key token set to binary zeros and has a key type of NULL.

Restrictions
The restrictions for CSNBKRC.

The record must have a unique label. Therefore, there cannot be another record in the DES key storage file with the same label and a different key type.

Required commands
The required commands for CSNBKRC.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNBKRC.

None

Related information
Additional information for CSNBKRC.
DES Key Record Create (CSNBKRC)

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRCJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKRCJ are shown here.

Format

```java
public native void CSNBKRCJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_label);
```
Use the DES Key Record Delete verb to perform one of the following tasks in the DES key storage file.

- Replace the token in a key record with a null key token
- Delete an entire key record, including the key label, from the key storage file

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

**Format**

The format of CSNBKRD.

```
CSNBKRD(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_label)
```

**Parameters**

The parameters for CSNBKRD.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

- **rule_array_count**
  - Direction: Input
  - Type: Integer
  - A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1.

- **rule_array**
  - Direction: Input
  - Type: String
  - The 8-byte keyword that defines the action to be performed. The `rule_array` keywords are described in Table 81.

**Table 81. Keywords for DES Key Record Delete control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task (One required)</td>
<td></td>
</tr>
<tr>
<td>TOKEN-DL</td>
<td>Deletes a key token from a key record in DES key storage.</td>
</tr>
<tr>
<td>LABEL-DL</td>
<td>Deletes an entire key record, including the key label, from DES key storage.</td>
</tr>
</tbody>
</table>

- **key_label**
  - Direction: Input
  - Type: String
  - The 64-byte label of a record in the key storage file that is the target of this verb.
Restrictions
The restrictions for CSNBKRD.

The record defined by the key_label must be unique.

Required commands
The required commands for CSNBKRD.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNBKRD.

None

Related information
Additional information about CSNBKRD.

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBKRDJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKRDJ are shown here.

Format
public native void CSNBKRDJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label);
The DES Key Record List verb creates a key-record-list file containing information about specified key records in key storage.

Information listed includes whether record validation is correct, the type of key, and the date and time the record was created and last updated.

Specify the key records to be listed using the key-label variable. To identify multiple key records, use the wild card (*) in the key label.

**Note:** To list all the labels in key storage, specify the key_label parameter with *.*, *.*.*, *.*.*.*, and so forth, up to a maximum of seven name tokens (*.*.*.*.*.*.*).

This verb creates the key-record-list file and returns the name of the file and the length of the file name to the calling application. The file has a header record, followed by 0 - n detail records, where n is the number of key records with matching key-labels. The file is kept in the /opt/IBM/CCA/keys/deslist directory (assuming the directory name was not changed during installation). These list files are created under the ownership of the environment of the user that requests the list service. Make sure the files created keep the same group ID as your installation requires. This can also be achieved by setting the “set-group-id-on-execution” bit on in this directory. See the g+s flags in the `chmod` command for full details. Not doing this might cause errors to be returned on key-record-list verbs.

DES key records are stored in the external key-storage file defined by the CSUDESDDS environment variable.

### Format
The format of CSNBKRL.

```plaintext
CSNBKRL(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_label,
    dataset_name_length,
    dataset_name,
    security_server_name
)
```

### Parameters
The parameters for CSNBKRL.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**key_label**

**Direction:** Input

**Type:** String

The `key_label` parameter is a pointer to a string variable containing the key label of a key-token record in key storage. In a key label, you can use a wild card (*) to identify multiple records in key storage.

**dataset_name_length**
DES Key Record List (CSNBKRL)

Direction: Output
Type: Integer

The dataset_name_length parameter is a pointer to an integer variable containing the number of bytes of data returned by the verb in the dataset_name variable. The maximum returned length is 64 bytes.

dataset_name
Direction: Output
Type: String

The dataset_name parameter is a pointer to a 64-byte string variable containing the name of the file returned by the verb. The file contains the key-record information.

The verb returns the file name as a fully qualified file specification.

Note: When the verb stores a key-record-list file, it overlays any older file with the same name.

security_server_name
Direction: Output
Type: String

The security_server_name parameter is a pointer to a string variable. The information in this variable is not currently used, but the variable must be declared.

Restrictions
The restrictions for CSNBKRL.

None

Required commands
The required commands for CSNBKRL.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNBKRL.

None

Related information
Additional information about CSNBKRL.

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.
JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRLJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKRLJ are shown here.

Format

```java
public native void CSNBKRLJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_label,
    hikmNativeInteger data_set_name_length,
    byte[] data_set_name,
    byte[] security_server_name);
```
DES Key Record Read (CSNBKRR)

Use the DES Key Record Read verb to copy an internal key token from the DES key storage file to application storage.

Other cryptographic services can then use the copied key token directly. The key token can also be used as input to the token copying functions of Key Generate or Key Import verbs to create additional NOCV keys.

DES key records are stored in the external key-storage file defined by the CSUDESDDS environment variable.

Format

The format of CSNBKRR.

```plaintext
CSNBKRR(  
    return_code,  
    reason_code,  
    exit_data_length,  
    exit_data,  
    key_label,  
    key_token)
```

Parameters

The parameters for CSNBKRR.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`key_label`

- **Direction:** Input
- **Type:** String

The 64-byte label of a record in the DES key storage file. The internal key token in this record is returned to the caller.

`key_token`

- **Direction:** Output
- **Type:** String

The 64-byte internal key token retrieved from the DES key storage file.

Restrictions

The restrictions for CSNBKRR.

The record defined by the `key_label` parameter must be unique and must already exist in the key storage file.

Required commands

The required commands for CSNBKRR.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.
Usage notes
The usage notes for CSNBKRR.

None

Related information
Additional information about CSNBKRR.

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 345.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBKRRJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBKRRJ are shown here.

Format
public native void CSNBKRRJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_label,
    byte[] key_token);
DES Key Record Write (CSNBKRW)

Use the DES Key Record Write verb to copy an internal DES key token from application storage into the DES key storage file.

The key label must be unique and the record must already exist in the key storage file.

DES key records are stored in the external key-storage file defined by the CSUDESDFS environment variable.

**Note:** Before you use this verb, use the DES Key Record Create verb (see "DES Key Record Create (CSNBKRC)" on page 365) to create a key record in the key storage file.

**Format**
The format of CSNBKRW.

```plaintext
CSNBKRW(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_token,
    key_label)
```

**Parameters**
The parameters for CSNBKRW.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**key_token**
- **Direction:** Input/Output
- **Type:** String

The 64-byte internal key token that is written to the DES key storage file.

**key_label**
- **Direction:** Input
- **Type:** String

The 64-byte label of a record in the DES key storage file that is the target of this verb. The record is updated with the internal key token supplied in the `key_token` parameter.

**Restrictions**
The restrictions for CSNBKRW.

The record defined by the `key_label` parameter must be unique and must already exist in the key storage file.

**Required commands**
The required commands for CSNBKRW.
DES Key Record Write (CSNBKRW)

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes

The usage notes for CSNBKRW.

None

Related information

Additional information about CSNBKRW.

You can use this verb with the key record create verb to write an initial record to key storage. Use it following the Key Import and Key Generate verb to write an operational key imported or generated by these verbs directly to the key storage file.

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBKRWJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBKRWJ are shown here.

Format

```java
public native void CSNBKRWJ(
    hklnativeInteger return_code,
    hklnativeInteger reason_code,
    hklnativeInteger exit_data_length,
    byte[] exit_data,
    byte[] key_token,
    byte[] key_label);
```
PKA Key Record Create (CSNDKRC)

This verb writes a new record to the PKA key storage file.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRC.

```c
CSNDKRC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label,
    token_length,
    token)
```

Parameters

The parameters for CSNDKRC.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0.

`rule_array`

Direction: Input
Type: String

This parameter is ignored.

`label`

Direction: Input
Type: String

The label of the record to be created, 64-byte character string.

`token_length`

Direction: Input
Type: Integer

The length of the field containing the token to be written to the PKA key storage file. If zero is specified, a null token will be added to the file. The maximum value of `token_length` is the maximum length of a private RSA token.

`token`

Direction: Input
Type: String
PKA Key Record Create (CSNDKRC)

Data to be written to the PKA key storage file if token_length is nonzero. An RSA private token in either external or internal format, or an RSA public token.

Restrictions
The restrictions for CSNDKRC.

None

Required commands
The required commands for CSNDKRC.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNDKRC.

None

Related information
Additional information about CSNDKRC.

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDKRCJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDKRCJ are shown here.

Format
```
public native void CSNDKRCJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label,
    hikmNativeInteger key_token_length,
    byte[] key_token);
```
PKA Key Record Delete (CSNDKRD)

Use PKA Key Record Delete to delete a record from the PKA key storage file.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRD.

```
CSNDKRD(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label)
```

Parameters

The parameters for CSNDKRD.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

  **Direction:** Input  
  **Type:** Integer  

  A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0 or 1.

`rule_array`

  **Direction:** Input  
  **Type:** String  

  Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 82.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deletion mode</td>
<td>(One, optional). Specifies whether the record is to be deleted entirely or whether only its contents are to be erased.</td>
</tr>
<tr>
<td>LABEL-DL</td>
<td>Specifies the record will be deleted from the PKA key storage file entirely. This is the default deletion mode.</td>
</tr>
<tr>
<td>TOKEN-DL</td>
<td>Specifies only the contents of the record are to be deleted. The record will still exist in the PKA key storage file, but will contain only binary zeros.</td>
</tr>
</tbody>
</table>

`label`

  **Direction:** Input  
  **Type:** String  

  The label of the record to be deleted, a 64-byte character string.
Restrictions
The restrictions for CSNDKRD.

None

Required commands
The required commands for CSNDKRD.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNDKRD.

None

Related information
Additional information about CSNDKRD.

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDKRDJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDKRDJ are shown here.

Format
```java
public native void CSNDKRDJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_identifier);
```
The PKA Key Record List verb creates a key-record-list file containing information about specified key records in PKA key-storage.

Information includes whether record validation is correct, the type of key, and the dates and times when the record was created and last updated.

Specify the key records to be listed using the `key_label` parameter. To identify multiple key records, use the wild card (*) in a key label.

**Note:** To list all the labels in key storage, specify the `key_label` parameter with 
*, ***, **.*, and so forth, up to a maximum of seven name tokens (**.*.*.*.*.*.*).  

This verb creates the list file and returns the name of the file and the length of the file name to the calling application. This verb also returns the name of the security server where the file is stored. The PKA Key Record List file has a header record, followed by 0 - n detail records, where n is the number of key records with matching key labels. The file is kept in the `/opt/IBM/CCA/keys/pkalist` directory (assuming the directory name was not changed during installation). These list files are created under the ownership of the environment of the user that requests the list verb. Make sure the files created kept the same group ID as your installation requires. This can also be achieved by setting the “set-group-id-on-execution” bit on in this directory. See the g+s flags in the `chmod` command for full details. Not doing this might cause errors to be returned on key-record-list verbs.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

For information concerning the location of the key-record-list directory, refer to the IBM 4764 PCI-X Cryptographic Coprocessor CCA Support Program Installation Manual.

**Format**

The format of CSNDKRL.

```plaintext
CSNDKRL(
    return_code,
    reason_code,
    exit_data_length,
    edit_data,
    rule_array_count,
    rule_array,
    key_label,
    dataset_name_length,
    dataset_name,
    security_server_name)
```

**Parameters**

The parameters for CSNDKRL.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

Direction: Input  
Type: Integer
A pointer to an integer variable containing the number of elements in the
rule_array variable. This value must be 0.

rule_array
Direction: Input
    Type: Array

This parameter is ignored.

key_label
Direction: Output
    Type: String

The key_label parameter is a pointer to a string variable containing a key record
in PKA key-storage. You can use a wild card (*) to identify multiple records in
key storage.

dataset_name_length
Direction: Input
    Type: Integer

The dataset_name_length parameter is a pointer to an integer variable containing
the number of bytes of data returned in the dataset_name variable. The
maximum returned length is 64 bytes.

dataset_name
Direction: Output
    Type: String

The dataset_name parameter is a pointer to a 64-byte string variable containing
the name of the file returned by the verb. The file contains the key-record
information.

The verb returns the file name as a fully qualified file specification.

Note: When the verb stores a key-record-list file, it overlays any older file with
the same name.

security_server_name
Direction: Output
    Type: String

The security_server_name parameter is a pointer to a string variable. The
information in this variable is not currently used, but the variable must be
declared.

Restrictions
The restrictions for CSNDKRL.

None

Required commands
The required commands for CSNDKRL.
This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNDKRL.
None

Related information
Additional information about CSNDKRL.
See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDKRLJ.
See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDKRLJ are shown here.

Format
```java
public native void CSNDKRLJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label,
    hikmNativeInteger data_set_name_length,
    byte[] data_set_name,
    byte[] security_server_name);
```
PKA Key Record Read (CSNDKRR)

Reads a record from the PKA key storage file and returns the content of the record. This is true even when the record contains a null PKA token.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRR.

```
CSNDKRR(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label,
    token_length,
    token)
```

Parameters

The parameters for CSNDKRR.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0.

`rule_array`

- **Direction:** Input
- **Type:** String

This parameter is ignored.

`label`

- **Direction:** Input
- **Type:** String

The label of the record to be read, a 64-byte character string.

`token_length`

- **Direction:** Input/Output
- **Type:** Integer

The length of the area to which the record is to be returned. On successful completion of this verb, `token_length` will contain the actual length of the record returned.

`token`

- **Direction:** Output
- **Type:** String
PKA Key Record Read (CSNDKRR)

Area into which the returned record will be written. The area should be at least as long as the record.

Restrictions
The restrictions for CSNDKRR.

None

Required commands
The required commands for CSNDKRR.

This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

Usage notes
The usage notes for CSNDKRR.

None

Related information
Additional information about CSNDKRR.

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDKRRJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDKRRJ are shown here.

Format
```java
public native void CSNDKRRJ(
        hikmNativeInteger return_code,
        hikmNativeInteger reason_code,
        hikmNativeInteger exit_data_length,
        byte[] exit_data,
        hikmNativeInteger rule_array_count,
        byte[] rule_array,
        byte[] key_label,
        hikmNativeInteger key_token_length,
        byte[] key_token);```

PKA Key Record Write (CSNDKRW)

Writes over an existing record in the PKA key storage file.

PKA key records are stored in the external key-storage file defined by the CSUPKADS environment variable.

Format

The format of CSNDKRW.

```c
CSNDKRW(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label,
    token_length,
    token)
```

Parameters

The parameters for CSNDKRW.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0 or 1.

`rule_array`

Direction: Input
Type: String

Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 83.

### Table 83. Keywords for PKA Key Record Write control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write mode</td>
<td>(One, optional). Specifies the circumstances under which the record is to be written.</td>
</tr>
<tr>
<td>CHECK</td>
<td>Specifies the record will be written only if a record of type NULL with the same label exists in the PKA key storage file. If such a record exists, it is overwritten. This is the default condition.</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>Specifies the record will be overwritten regardless of the current content of the record. If a record with the same label exists in the PKA key storage file, is overwritten.</td>
</tr>
</tbody>
</table>

`label`

Direction: Input
Type: String

The label of the record to be overwritten, a 64-byte character string.
PKA Key Record Write (CSNDKRW)

**token_length**
- **Direction:** Input
- **Type:** Integer
  - The length of the field containing the token to be written to the PKA key storage file.

**token**
- **Direction:** Input
- **Type:** String
  - The data to be written to the PKA key storage file, which is an RSA private token in either external or internal format, or an RSA public token.

**Restrictions**
- **The restrictions for CSNDKRW.**
  - None

**Required commands**
- **The required commands for CSNDKRW.**
  - This verb requires the Key Test and Key Test2 command (offset X'001D') to be enabled in the active role.

**Usage notes**
- **The usage notes for CSNDKRW.**
  - None

**Related information**
- **Additional information about CSNDKRW.**
  - See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

**JNI version**
- **This verb has a Java Native Interface (JNI) version, which is named CSNDKRWJ.**
  - See "Building Java applications to use with the CCA JNI" on page 26.
  - The parameters for CSNDKRWJ are shown here.

**Format**
```java
public native void CSNDKRWJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label,
    hikmNativeInteger key_token_length,
    byte[] key_token);
```
Retained Key Delete (CSNDRKD)

Use this verb to delete a PKA key-record currently retained within the cryptographic engine.

Both public and private keys can be retained within the cryptographic engine using verbs such as PKA Key Generate. A list of retained keys can be obtained using the Retained Key List verb.

**IMPORTANT**
Before using this verb, see the information about retained keys in "Using retained keys."

Format

The format of CSNDRKD.

```
CSNDRKD(  return_code,  
          reason_code,  
          exit_data_length,  
          exit_data,  
          rule_array_count,  
          rule_array,  
          key_label)
```

Parameters

The parameters for CSNDRKD.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

- **rule_array_count**
  - Direction: Input
  - Type: Integer
  - A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0.

- **rule_array**
  - Direction: Input
  - Type: String
  - This parameter is ignored.

- **key_label**
  - Direction: Input
  - Type: String
  - A pointer to a string variable containing the key label of a PKA key-record that has been retained within the cryptographic engine. The use of a wild card in the `key_label` variable is not permitted.

Using retained keys

Retained key use is discouraged on the IBM System z platform because a retained key can exist only in one CEX*C Cryptographic adapter, by definition.
Retained Key Delete (CSNDRKD)

- This has potential problems:
  - The key cannot be exported, so it cannot be backed up.
  - The key cannot be exported to another card in the same group, so operations concerning the retained key cannot participate in load-balancing.
  - There is an exception to the above points, in that keys generated in a deterministic fashion using externally saved regeneration data (it is possible to save so-called 'regen data' securely) can be recreated from that data or created in multiple cards across a card group.

However, this is a very sophisticated topic, and is beyond the scope of this document. Also, the complexity required to implement this properly, as well as the sophistication involved in its data management, present formidable obstacles.

Retained key support is offered in this release, however. The following verbs work with retained keys:

- "PKA Key Generate (CSNDPKG)" on page 478 generates an RSA retained key. The same restrictions that Integrated Cryptographic Service Facility (ICSF) has for retained key creation are implemented here. These are:
  - Notice that PKA Key Token Build will let you create 'key-mgmt' skeleton key tokens, and this is as designed. You can still pass these to PKA Key Generate and have a key pair created. What is not allowed is specifying that this 'key-mgmt' token is to be generated in PKA Key Generate as a RETAIN key token: a retained key. Such an attempt will fail with error 12 reason code 3046.
  - The maximum modulus size is 2048 bits.
  - The usage flags are restricted to signature generation.
    Specifically, key management usage for retained keys is not allowed because of the dangers of losing your key encrypting key (kek) for important keys, when that kek exists only inside a single adapter.
- "Retained Key List (CSNDRKL)" on page 390 lists the retained keys inside an adapter.
- "Retained Key Delete (CSNDRKD)" on page 387 deletes a retained key from adapter internal storage.

Restrictions
The restrictions for CSNDRKD.
None

Required commands
The required commands for CSNDRKD.

This verb requires the Retained Key Delete command (offset X'0203') to be enabled in the active role.

Usage notes
The usage notes for CSNDRKD.

This verb is not impacted by the AUTOSELECT option. See Table 1 on page 10 for more information.
Retained Key Delete (CSNDRKD)

Related information
Additional information about CSNDRKD.

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 348.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDRKDJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDRKDJ are shown here.

Format
public native void CSNDRKDJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label);
Retained Key List (CSNDRKL)

Use this verb to list the key labels of selected PKA key records that have been retained within the cryptographic engine.

Specify the keys to be listed using the `key_label_mask` variable. To identify multiple keys, use a wild card (*) in the mask. Only labels with matching characters to those in the mask up to the first "*" is returned. To list all retained key labels, specify a mask of an ", followed by 63 space characters. For example, if the cryptographic engine has retained key labels a.a, a.a1, a.b.c.d, and z.a, and you specify the mask `a.*`, the verb returns a.a, a.a1 and a.b.c.d. If you specify a mask of `a.a*`, the verb returns a.a and a.a1.

To retain PKA keys within the coprocessor, use the PKA Key Generate verb. To delete retained keys from the coprocessor, use the Retained Key Delete verb.

**IMPORTANT**

Before using this verb, see the information about retained keys in "Using retained keys" on page 387.

### Format

The format of CSNDRKL.

```plaintext
CSNDRKL(  return_code,  reason_code,  exit_data_length,  exit_data,  rule_array_count,  rule_array,  key_label_mask,  retained_keys_count,  key_labels_count,  key_labels)
```

### Parameters

The parameters for CSNDRKL.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

  Direction: Input
  Type: Integer

  A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0.

**rule_array**

  Direction: Input
  Type: Array

  This parameter is ignored.

**key_label_mask**
Retained Key List (CSNDRKL)

**Direction: Input**
Type: String

A pointer to a string variable containing a key-label mask that is used to filter the list of key names returned by the verb. Use a wild card (*) to identify multiple key records retained within the coprocessor.

**retained_keys_count**
Direction: Input/Output
Type: Integer

A pointer to an integer variable to receive the total number of retained-key records stored within the coprocessor.

**key_labels_count**
Direction: Input/Output
Type: Integer

A pointer to an integer variable which on input defines the maximum number of key labels to be returned, and which on output defines the number of key labels returned by the coprocessor.

**key_labels**
Direction: Output
Type: Array

A pointer to a string array variable containing the returned key labels. The coprocessor returns zero or more 64-byte array elements, each of which contains the key label of a PKA key-record retained within the coprocessor.

**Restrictions**
The restrictions for CSNDRKL.

None

**Required commands**
The required commands for CSNDRKL.

This verb requires the Retained Key List command (offset X'0230') to be enabled in the active role.

**Usage notes**
The usage notes for CSNDRKL.

This verb is not impacted by the AUTOSELECT option. See Table 1 on page 10 for more information.

**Related information**
Related information about CSNDRKL.

See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System Z” on page 348.
JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDRKLJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDRKLJ are shown here.

**Format**

```java
public native void CSNDRKLJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_label_mask,
    hikmNativeInteger retained_keys_count,
    hikmNativeInteger key_labels_count,
    byte[] key_labels);
```
Chapter 10. Financial services

The process of validating personal identities in a financial transaction system is called personal authentication.

The personal identification number (PIN) is the basis for verifying the identity of a customer across financial industry networks. CCA provides verbs to translate, verify, and generate PINs. You can use the verbs to prevent unauthorized disclosures when organizations handle PINs.

The following verbs are described in this chapter:

- "Clear PIN Encrypt (CSNBCPE)" on page 405
- "Clear PIN Generate (CSNBPGN)" on page 408
- "Clear PIN Generate Alternate (CSNCPGA)" on page 412
- "CVV Generate (CSNBCSG)" on page 417
- "CVV Verify (CSNBCSV)" on page 426
- "Encrypted PIN Generate (CSNBEGP)" on page 430
- "Encrypted PIN Translate (CSNBPTT)" on page 435
- "Encrypted PIN Verify (CSNBPTV)" on page 441
- "PIN Change/Unblock (CSNBPCU)" on page 446
- "Secure Messaging for Keys (CSNBSKY)" on page 453
- "Secure Messaging for PINs (CSNBSPN)" on page 457
- "Transaction Validation (CSNBTRV)" on page 462

How personal identification numbers (PINs) are used

CCA allows your applications to generate PINs, to verify supplied PINs, and to translate PINs from one format or encryption key to another.

Many people are familiar with PINs, which are used to access an automated teller machine (ATM). From the system point of view, PINs are used primarily in financial networks to authenticate users. Typically, a user is assigned a PIN and enters the PIN at automated teller machines (ATMs) to gain access to his or her accounts. It is extremely important that the PIN be kept private so no one other than the account owner can use it.

How VISA card verification values are used

The Visa International Service Association (VISA) and MasterCard International, Incorporated have specified a cryptographic method to calculate a value that relates to the personal account number (PAN), the card expiration date, and the service code.

The VISA card-verification value (CVV) and the MasterCard card-verification code (CVC) can be encoded on either track 1 or track 2 of a magnetic striped card and are used to detect forged cards. Because most online transactions use track-2, the CCA verbs generate and verify the CVV^2 by the track-2 method.
The VISA CVV Generate verb calculates a 1-byte to 5-byte value through the DES-encryption of the PAN, the card expiration date, and the service code using two data-encrypting keys or two MAC keys. The VISA CVV Verify verb calculates the CVV by the same method, compares it to the CVV supplied by the application (which reads the credit card’s magnetic stripe) in the CVV_value, and issues a return code that indicates whether the card is authentic.

The CVV Key Combine verb combines two operational DES keys into one operational TDES key. The verb accepts as input two single-length keys that are suitable for use with the CVV (card-verification value) algorithm. The resulting double-length key meets a more recent industry standard of using TDES to support PIN-based transactions. In addition, the double-length key is in a format that can be wrapped using the Key Export to TR31 verb.

### Translating data and PINs in networks

More and more data is being transmitted across networks where, for various reasons, the keys used on one network cannot be used on another network.

Encrypted data and PINs that are transmitted across these boundaries must be translated securely from encryption under one key to encryption under another key. For example, a traveler visiting a foreign city might want to use an ATM to access an account at home. The PIN entered at the ATM might need to be encrypted at the ATM and sent over one or more financial networks to the traveler’s home bank. At the home bank, the PIN must be verified before access is allowed. On intermediate systems (between networks), applications can use the Encrypted PIN Translate verb to re-encrypt a PIN block from one key to another. Running on CCA, such applications can ensure that PINs never appear in the clear and that the PIN-encrypting keys are isolated on their own networks.

### Working with Europay-Mastercard-Visa Smart cards

There are several verbs you can use in secure communications with Europay-Mastercard-Visa (EMV) smart cards.

The processing capabilities are consistent with the specifications provided in these documents:

- EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2
- Integrated Circuit Card Specification (VIS) 1.4.0 Corrections

EMV smart cards include the following processing capabilities:

- The Diversified Key Generate verb with rule-array options TDES-XOR, TDESEMV2, and TDESEMV4 enables you to derive a key used to cipher and authenticate messages, and more particularly message parts, for exchange with an EMV smart card. You use the derived key with verbs such as: Encipher, Decipher, MAC Generate, MAC Verify, Secure Messaging for Keys, and Secure Messaging for PINs. These message parts can be combined with message parts created using the Secure Messaging for Keys and Secure Messaging for PINs verbs.
The Secure Messaging for Keys verb enables secure incorporation of a key into a message part (generally the value portion of a TLV component of a secure message for a card). Similarly, the Secure Messaging for PINs verb enables secure incorporation of a PIN block into a message part.

PIN Change/Unblock verb enables encryption of a new PIN to send to a new EMV card, or to update the PIN value on an initialized EMV card. This verb generates both the required session key (from the master encryption key) and the required authentication code (from the master authentication key).

The ZERO-PAD option of the PKA Encrypt enables validation of a digital signature created according to ISO 9796-2 standard by encrypting information that you format, including a hash value of the message to be validated. You compare the resulting enciphered data to the digital signature accompanying the message to be validated.

The MAC Generate and MAC Verify verbs post-pad a X’80’...X’00’ string to a message as required for authenticating messages exchanged with EMV smart cards.

**PIN verbs**

This section describes the PIN verbs, as well as the various PIN algorithms and PIN block formats supported by CCA. It also explains the use of PIN-encrypting keys.

You use the PIN verbs to generate, verify, and translate PINs.

**Generating a PIN**

To generate personal identification numbers, call the Clear PIN Generate or Encrypted PIN Generate verb.

Using a PIN generation algorithm, data used in the algorithm, and the PIN generation key, the Clear PIN Generate verb generates a clear PIN and a PIN verification value, or offset. Using a PIN generation algorithm, data used in the algorithm, the PIN generation key, and an outbound PIN encrypting key, the Encrypted PIN Generate verb generates and formats a PIN and encrypts the PIN block.

**Encrypting a PIN**

To format a PIN into a supported PIN block format and encrypt the PIN block, call the Clear PIN Encrypt verb.

**Generating a PIN validation value from an encrypted PIN block**

To generate a clear VISA PIN validation value (PVV) from an encrypted PIN block, call the Clear PIN Generate Alternate verb.

The PIN block can be encrypted under an input PIN-encrypting key (IPINENC) or an output PIN encrypting key (OPINENC).

**Verifying a PIN**

To verify a supplied PIN, call the Encrypted PIN Verify verb.

You supply the enciphered PIN, the PIN-encrypting key that enciphers the PIN, and other data. You must also specify the PIN verification key and PIN verification
algorithm. The Encrypted PIN Verify verb generates a verification PIN. This verb compares the two personal identification numbers and if they are the same, it verifies the supplied PIN.

**Translating a PIN**

To translate a PIN block format from one PIN-encrypting key to another or from one PIN block format to another, call the Encrypted PIN Translate verb.

You must identify the input PIN-encrypting key that originally enciphered the PIN. You also need to specify the output PIN-encrypting key that you want the verb to use to encipher the PIN. If you want to change the PIN block format, specify a different output PIN block format from the input PIN block format.

**Algorithms for generating and verifying a PIN**

CCA supports the following algorithms for generating and verifying personal identification numbers.

- IBM 3624 institution-assigned PIN
- IBM 3624 customer-selected PIN (through a PIN offset)
- IBM German Bank Pool PIN (verify through an institution key)
- VISA PIN through a VISA PIN validation value
- Interbank PIN

The algorithms are described in detail in Appendix E, “PIN formats and algorithms,” on page 683.

**Using PINs on different systems**

CCA allows you to translate different PIN block formats, which lets you use personal identification numbers on different systems.

CCA supports the following formats:

- IBM 3624
- IBM 3621 (same as IBM 5906)
- IBM 4704 encrypting PINPAD format
- ISO 0 (same as ANSI 9.8, VISA 1, and ECI 1)
- ISO 1 (same as ECI 4)
- ISO 2
- VISA 2
- VISA 3
- VISA 4
- ECI 2
- ECI 3

The algorithms are described in detail in Appendix E, “PIN formats and algorithms,” on page 683.

**PIN-Encrypting keys**

A unique master key variant enciphers each type of key.

Note that the PIN block variant constant (PBVC) are not supported in this version of CCA.
Derived unique key per transaction algorithms
CCA supports ANSI X9.24 derived unique key per transaction algorithms to
generate PIN-encrypting keys from user data.

CCA supports both single-length and double-length key generation. Keywords for
single-length and double-length key generation cannot be mixed.

Encrypted PIN Translate
The UKPTIPIN, IPKTOPIN, and UKPTBOTH keywords will cause the verb to
generate single-length keys. and DUKPT-IP, DKPT-OP, and DUKPT-BH are the
respective keywords to generate double-length keys.

The input_PIN_profile and output_PIN_profile parameters must supply the current
key serial number when these keywords are specified.

Encrypted PIN Verify
The UKPTIPIN keyword will cause the verb to verify single-length keys.
DUKPT-IP is the keyword for double-length key generation.

The input_PIN_profile parameter must supply the current key serial number when
these keywords are specified.

ANSI X9.8 PIN restrictions
These access control points implement the PIN-block processing restrictions of the
ANSI X9.8 standard implemented in CCA 4.1.0.

These access control points are available on the IBM z196 starting with the CEX3C
feature. These access control points are disabled in the default role. A TKE
Workstation is required to enable them.

These are the access control points:
• ANSI X9.8 PIN - Enforce PIN block restrictions (offset X'0350'). See “ANSI X9.8
  PIN - Enforce PIN block restrictions” on page 398.
• ANSI X9.8 PIN - Allow modification of PAN_01_0350 (offset X'0351') See “ANSI
  X9.8 PIN - Allow modification of PAN” on page 398.
• ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350 (offset X'0352') See
  “ANSI X9.8 PIN - Allow only ANSI PIN blocks” on page 398.
• ANSI X9.8 PIN - Use stored decimalization table only (offset X'0356') See “Use
  Only Valid Decimalization Tables” on page 399.

These verbs are affected by these access control points:
• Clear PIN Generate Alternate (CSNBCPA)
• Encrypted PIN Generate (CSNBEPG)
• Encrypted PIN Translate (CSNBPTR)
• Encrypted PIN Verify (CSNPBVR)
• Secure Messaging for PINs (CSNBSNP)

PIN decimalization tables can be stored in the Coprocessor, starting with CEX3C,
for use by CCA verbs. Only tables that have been activated can be used. A TKE
Workstation is required to manage the tables in the coprocessors.
ANSI X9.8 PIN - Enforce PIN block restrictions

When the ANSI X9.8 PIN - Enforce PIN block restrictions access control point is enabled, restrictions are enforced.

The following restrictions will be enforced:

- The Encrypted PIN Translate and Secure Messaging for PINs verbs will not accept IBM 3624 PIN format in the output profile parameter when the input profile parameter is not IBM 3624.
- The Encrypted PIN Translate verb will not accept ISO-0 or ISO-3 formats in the input PIN profile unless ISO-0 or ISO-3 is in the output PIN profile.
- The Encrypted PIN Translate and Secure Messaging for PINs verbs will not accept ISO-1 or ISO-2 formats in the output profile parameter when the input profile parameter contains ISO-0, ISO-3, or VISA4.
- When the input profile parameter for the Encrypted PIN Translate and Secure Messaging for PINs verbs contains either ISO-0 or ISO-3 formats, the PAN within the decrypted PIN block will be extracted. This PAN must be the same as the PAN that was supplied as the input PAN parameter, and this PAN must be the same as the PAN supplied as the output PAN parameter.
- The input PAN and output PAN parameters for the Encrypted PIN Translate and Secure Messaging for PINs verbs must be equivalent.
- When the rule array for the Clear PIN Generate Alternate verb contains VISA-PVV, the input PIN profile must contain ISO-0 or ISO-3 formats.

ANSI X9.8 PIN - Allow modification of PAN

In order to enable the ANSI X9.8 PIN - Allow modification of PAN access control point, the ANSI X9.8 PIN - Enforce PIN block restrictions must also be enabled.

The ANSI X9.8 PIN - Allow modification of PAN access control point cannot be enabled by itself.

When the ANSI X9.8 PIN - Allow modification of PAN access control point is enabled, the input PAN and output PAN parameters will be tested in the Encrypted PIN Translate and Secure Messaging for PINs verbs. The input PAN will be compared to the portions of the PAN that are recoverable from the decrypted PIN block. If the PANs are the same, the account number will be changed in the output PIN block.

ANSI X9.8 PIN - Allow only ANSI PIN blocks

In order to enable the ANSI X9.8 PIN - Allow only ANSI PIN blocks access control point, the ANSI X9.8 PIN - Enforce PIN block restrictions must also be enabled.

The ANSI X9.8 PIN - Allow only ANSI PIN blocks access control point cannot be enabled by itself.

When this access control point is enabled, the Encrypted PIN Translate verb will allow reformatting of the PIN block as shown in Table 84.

<table>
<thead>
<tr>
<th>Reformat to:</th>
<th>ISO Format 0</th>
<th>ISO Format 1</th>
<th>ISO Format 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reformat from:</td>
<td>ISO Format 0</td>
<td>Not permitted</td>
<td>Reformat permitted. Change of PAN not permitted.</td>
</tr>
<tr>
<td>Reformat to:</td>
<td>ISO Format 0</td>
<td>ISO Format 1</td>
<td>ISO Format 3</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Reformat from:</td>
<td>ISO Format 1</td>
<td>Reformat permitted</td>
<td>Reformat permitted</td>
</tr>
<tr>
<td>ISO Format 1</td>
<td>Reformat permitted</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Use Only Valid Decimalization Tables**

The Use Only Valid Decimalization Tables access control point can be enabled by itself.

When this access control point is enabled, the Secure Messaging for PINs, Clear PIN Generate Alternate, Encrypted PIN Generate, and Encrypted PIN Verify verbs must supply a decimalization table that matches the active decimalization tables stored in the coprocessors. The decimalization table in the `data_array` parameter will be compared against the active decimalization tables in the coprocessor, and if the supplied table matches a stored table, the request will be processed. If the supplied table doesn’t match any of the stored tables or there are no stored tables, the request will fail.

PIN decimalization tables can be stored in the coprocessor, starting with CEX3C, for use by CCA verbs. Only tables that have been activated can be used. A TKE Workstation is required to manage the tables in the coprocessors.

**The PIN profile**

The components of the PIN profile are described in this topic.

The PIN profile consists of the following:

- PIN block format (see "PIN block format")
- Format control (see "Format control" on page 402)
- Pad digit (see "Pad digit" on page 402)
- Current Key Serial Number (for UKPT and DUKPT – see "Current key serial number" on page 403)

Table 85 shows the format of a PIN profile.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 7</td>
<td>PIN block format</td>
</tr>
<tr>
<td>8 - 15</td>
<td>Format control</td>
</tr>
<tr>
<td>16 - 23</td>
<td>Pad digit</td>
</tr>
<tr>
<td>24 - 47</td>
<td>Current Key Serial Number (for UKPT and DUKPT)</td>
</tr>
</tbody>
</table>

**PIN block format**

This keyword specifies the format of the PIN block. The 8-byte value must be left-aligned and padded with blanks.

Refer to [Table 86 on page 400](#) for a list of valid values.
Table 86. Format values of PIN blocks

<table>
<thead>
<tr>
<th>Format value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECI-2</td>
<td>Eurocheque International format 2</td>
</tr>
<tr>
<td>ECI-3</td>
<td>Eurocheque International format 3</td>
</tr>
<tr>
<td>ISO-0</td>
<td>ISO format 0, ANSI X9.8, VISA 1, and ECI 1</td>
</tr>
<tr>
<td>ISO-1</td>
<td>ISO format 1 and ECI 4</td>
</tr>
<tr>
<td>ISO-2</td>
<td>ISO format 2</td>
</tr>
<tr>
<td>ISO-3</td>
<td>ISO format 3</td>
</tr>
<tr>
<td>VISA-2</td>
<td>VISA format 2</td>
</tr>
<tr>
<td>VISA-3</td>
<td>VISA format 3</td>
</tr>
<tr>
<td>VISA-4</td>
<td>VISA format 4</td>
</tr>
<tr>
<td>3621</td>
<td>IBM 3621 and 5906</td>
</tr>
<tr>
<td>3624</td>
<td>IBM 3624</td>
</tr>
<tr>
<td>4704-EPP</td>
<td>IBM 4704 encrypting PIN pad</td>
</tr>
</tbody>
</table>

**PIN block format and PIN extraction method keywords**

In the Clear PIN Generate Alternate, Encrypted PIN Translate, and Encrypted PIN Verify verbs, you can specify a PIN extraction keyword for a given PIN block format.

In the table below, the allowable PIN extraction methods are listed for each PIN block format. The first PIN extraction method keyword listed for a PIN block format is the default. Refer to Table 87 for a list of valid values.

Table 87. PIN block format and PIN extraction method keywords

<table>
<thead>
<tr>
<th>PIN block format</th>
<th>PIN extraction method keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECI-2</td>
<td>PINLEN04</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINLEN04 format.</td>
</tr>
<tr>
<td>ECI-3</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
<tr>
<td>ISO-0</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
<tr>
<td>ISO-1</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
<tr>
<td>ISO-2</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
<tr>
<td>ISO-3</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
<tr>
<td>VISA-2</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
<tr>
<td>VISA-3</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
<tr>
<td>VISA-4</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
</tbody>
</table>
The PIN extraction methods operate as follows:

**PINBLOCK**
- Specifies that the service verb use one of these:
  - The PIN length, if the PIN block contains a PIN length field
  - The PIN delimiter character, if the PIN block contains a PIN delimiter character.

**PADDIGIT**
- Specifies that the verb use the pad value in the PIN profile to identify the end of the PIN.

**HEXDIGIT**
- Specifies that the verb use the first occurrence of a digit in the range from X'A' to X'F' as the pad value to determine the PIN length

**PINLENnn**
- Specifies that the verb use the length specified in the keyword, where nn can range from 04 - 16, the number of digits used to identify the PIN.

**PADEXIST**
- Specifies that the verb use the character in the 16th position of the PIN block as the value of the pad value.

### Enhanced PIN security mode
An enhanced PIN security mode is available. This optional mode is selected by enabling the PTR Enhanced PIN Security (offset X'0313') access control point in the CEX2C or CEX*C default role.

When active, this control point affects all PIN verbs that extract or format a PIN using a PIN-block format of 3621 or 3624 with a PIN-extraction method of PADDIGIT.

Table 88 on page 402 summarizes the verbs affected by the enhanced PIN security mode, and describes the effect that the mode has when the access control point is enabled.

<table>
<thead>
<tr>
<th>PIN block format</th>
<th>PIN extraction method keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3621</td>
<td>PADDIGIT, HEXDIGIT, PINLEN04 - PINLEN12, PADEXIST</td>
<td>The PIN extraction method keywords specify a PIN extraction method for an IBM 3621 PIN block format. The first keyword, PADDIGIT, is the default PIN extraction method for the PIN block format.</td>
</tr>
<tr>
<td>3624</td>
<td>PADDIGIT, HEXDIGIT, PINLEN04 - PINLEN16, PADEXIST</td>
<td>The PIN extraction method keywords specify a PIN extraction method for an IBM 3624 PIN block format. The first keyword, PADDIGIT, is the default PIN extraction method for the PIN block format.</td>
</tr>
<tr>
<td>4704-EPP</td>
<td>PINBLOCK</td>
<td>The PIN extraction method keywords specify a PIN extraction method for a PINBLOCK format.</td>
</tr>
</tbody>
</table>
Table 88. Verbs affected by enhanced PIN security mode

<table>
<thead>
<tr>
<th>PIN-block format and PIN-extraction method</th>
<th>Affected verbs</th>
<th>PIN processing changes when Enhanced PIN Security Mode enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECI-2, 3621, or 3624 formats AND PINLENnn</td>
<td>Clear PIN Generate AlternateEncrypted PIN TranslateEncrypted PIN VerifyPIN</td>
<td>The PINLEN keyword in the rule_array parameter for PIN extraction method is not allowed if the Enhanced PIN Security Mode is enabled. <strong>Note:</strong> The verb will fail with return code 8 and reason code X'7E0'.</td>
</tr>
<tr>
<td>3621 or 3624 format and PADDIGIT</td>
<td>Clear PIN Generate AlternateEncrypted PIN TranslateEncrypted PIN VerifyPIN Change/UnblockPIN</td>
<td>PIN formatting determines the PIN length by scanning from right to left until a digit, not equal to the PAD digit, is found. The minimum PIN length is set at four digits, so scanning ceases one digit past the position of the fourth PIN digit in the block.</td>
</tr>
<tr>
<td>3621 or 3624 format and PADDIGIT</td>
<td>Clear PIN EncryptEncrypted PIN GenerateEncrypted PIN TranslatePIN</td>
<td>Format control This keyword specifies whether there is any control on the user-supplied PIN format. The 8-byte value must be left-aligned and padded with blanks. The only permitted value is NONE, which indicates no format control will be used.</td>
</tr>
<tr>
<td>3621 or 3624 format and PADDIGIT</td>
<td>Encrypted PIN Translate</td>
<td>Restricted to non-decimal digit for PAD digit.</td>
</tr>
</tbody>
</table>

**Format control**

This keyword specifies whether there is any control on the user-supplied PIN format.

The 8-byte value must be left-aligned and padded with blanks. The only permitted value is NONE, which indicates no format control will be used.

**Pad digit**

Some PIN formats require the pad digit parameter.

If the PIN format does not need a pad digit, the verb ignores this parameter.

Table 89 shows the format of a pad digit. The PIN profile pad digit must be specified in upper case.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 - 22</td>
<td>Seven space characters</td>
</tr>
<tr>
<td>23</td>
<td>Character representation of a hexadecimal pad digit or a space if a pad digit is not needed. Characters must be one of the following: digits 0 - 9, letters A - F, or a blank.</td>
</tr>
</tbody>
</table>

Each PIN format supports only a pad digit in a certain range. Table 90 lists the valid pad digits for each PIN block format.

<table>
<thead>
<tr>
<th>PIN Block Format</th>
<th>Output PIN Profile</th>
<th>Input PIN Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECI-2</td>
<td>Pad digit is not used</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>ECI-3</td>
<td>Pad digit is not used</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>ISO-0</td>
<td>F</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>ISO-1</td>
<td>Pad digit is not used</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>ISO-2</td>
<td>Pad digit is not used</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>ISO-3</td>
<td>Pad digit is not used</td>
<td>Pad digit is not used</td>
</tr>
</tbody>
</table>
Table 90. Pad digits for PIN block formats (continued)

<table>
<thead>
<tr>
<th>PIN Block Format</th>
<th>Output PIN Profile</th>
<th>Input PIN Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>VISA-2</td>
<td>0 - 9</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>VISA-3</td>
<td>0 - F</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>VISA-4</td>
<td>F</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>3621</td>
<td>0 - F</td>
<td>0 - F</td>
</tr>
<tr>
<td>3624</td>
<td>0 - F</td>
<td>0 - F</td>
</tr>
<tr>
<td>4704-EPP</td>
<td>F</td>
<td>Pad digit is not used</td>
</tr>
</tbody>
</table>

The verb returns an error indicating that the PAD digit is not valid if all of these conditions are met:
- The PTR Enhanced PIN Security (offset X'0313') access control point is enabled in the active role.
- The output PIN profile specifies 3621 or 3624 as the PIN-block format.
- The output PIN profile specifies a decimal digit (0 - 9) as the PAD digit.

**Recommendations for the pad digit**

IBM recommends you use a non-decimal pad digit in the range of A - F when processing IBM 3624 and IBM 3621 PIN blocks.

If you use a decimal pad digit, the creator of the PIN block must ensure that the calculated PIN does not contain the pad digit, or unpredictable results might occur.

For example, you can exclude a specific decimal digit from being in any calculated PIN by using the IBM 3624 calculation procedure and by specifying a decimalization table that does not contain the desired decimal pad digit.

**Current key serial number**

The current key serial number is the concatenation of the initial key serial number (a 59-bit value) and the encryption counter (a 21-bit value).

The concatenation is an 80-bit (10-byte) value. Table 91 shows the format of the current key serial number.

When UKPT or DUKPT is specified, the PIN profile parameter is extended to a 48-byte field and must contain the current key serial number.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 - 47</td>
<td>Character representation of the current key serial number used to derive the initial PIN encrypting key. It is left-aligned and padded with 4 blanks.</td>
</tr>
</tbody>
</table>

**Decimalization tables**

Decimalization tables can be loaded in the coprocessors to restrict attacks using modified tables.

The management of the tables requires a TKE workstation.

These verbs make use of the stored decimalization tables:
- Clear PIN Generate (CSNBPGN)
• Clear PIN Generate Alternate (CSNBCPA)
• Encrypted PIN Generate (CSNBEPG)
• Encrypted PIN Verify (CSNBPV)

The ANSI X9.8 PIN - Use stored decimalization table only (offset X'0356') access control point is used to restrict the use of the stored decimalization tables. When the access control point is enabled, the table supplied by the verb will be compared against the active tables stored in the coprocessor. If the supplied table does not match any of the active tables, the request will fail.

A TKE workstation (Version 7.1 or later) is required to manage the PIN decimalization tables. The tables must be loaded and then activated. Only active tables are checked when the access control point is enabled.

**Note:** CCA routes work to all active coprocessors based on workload. All coprocessors must have the same set of decimalization tables for the decimalization table access control point to be effective.
Clear PIN Encrypt (CSNBCPE)

The Clear PIN Encrypt verb formats a PIN into one of the following PIN block formats and encrypts the results.

You can use this verb to create an encrypted PIN block for transmission. With the RANDOM keyword, you can have the verb generate random PIN numbers.

**Note:** A clear PIN is a sensitive piece of information. Ensure your application program and system design provide adequate protection for any clear PIN value.

- IBM 3621 format
- IBM 3624 format
- ISO-0 format (same as the ANSI X9.8, VISA-1, and ECI formats)
- ISO-1 format (same as the ECI-4 format)
- ISO-2 format
- ISO-3 format
- IBM 4704 encrypting PINPAD (4704-EPP) format
- VISA 2 format
- VISA 3 format
- VISA 4 format
- ECI2 format
- ECI3 format

**Format**

The format of CSNBCPE.

```
CSNBCPE(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    PIN_encrypting_key_identifier,
    rule_array_count,
    rule_array,
    clear_PIN,
    PIN_profile,
    PAN_data,
    sequence_number
    encrypted_PIN_block)
```

**Parameters**

The parameter definitions for CSNBCPE.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see [Parameters common to all verbs](#) on page 20.

**PIN_encrypting_key_identifier**

- **Direction:** Input/Output
- **Type:** String

The 64-byte string containing an internal key token or a key label of an internal key token. The internal key token contains the key that encrypts the PIN block. The control vector in the internal key token must specify an OPINENC key type and have the CPINENC usage bit set to B'1'.

Chapter 10. Financial services  405
Clear PIN Encrypt (CSNBCPE)

rule_array_count
Direction: Input
Type: Integer
A pointer to an integer variable containing the number of elements in the rule_array variable. Valid values are 0 and 1.

rule_array
Direction: Input
Type: String
Keywords that provide control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The rule_array keywords are described in Table 92.

Table 92. Keywords for Clear PIN Encrypt control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Rule (Optional)</td>
<td></td>
</tr>
<tr>
<td>ENCRYPT</td>
<td>This is the default. Use of this keyword is optional.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>Causes the verb to generate a random PIN value. The length of the PIN is</td>
</tr>
<tr>
<td></td>
<td>based on the value in the clear_PIN variable. Set the value of the clear</td>
</tr>
<tr>
<td></td>
<td>PIN to zero and use as many digits as the desired random PIN. Pad the</td>
</tr>
<tr>
<td></td>
<td>remainder of the clear PIN variable with space characters.</td>
</tr>
</tbody>
</table>

clear_PIN
Direction: Input
Type: String
A 16-character string with the clear PIN. The value in this variable must be left-aligned and padded on the right with space characters.

PIN_profile
Direction: Input
Type: String
A 24-byte string containing three 8-byte elements with a PIN block format keyword, the format control keyword, NONE, and a pad digit as required by certain formats. See “The PIN profile” on page 399 for additional information.

PAN_data
Direction: Input
Type: String
A 12-byte PAN in character format. The verb uses this parameter if the PIN profile specifies the ISO-0, ISO-3, or VISA-4 keyword for the PIN block format. Otherwise, ensure this parameter is a 12-byte variable in application storage. The information in this variable will be ignored, but the variable must be specified.

Note: When using the ISO-0 or ISO-3 keyword, use the 12 rightmost digits of the PAN data, excluding the check digit. When using the VISA-4 keyword, use the 12 leftmost digits of the PAN data, excluding the check digit.
Clear PIN Encrypt (CSNBCPE)

Direction: Input
Type: Integer
The 4-byte character integer. The verb currently ignores the value in this variable. For future compatibility, the suggested value is 99999.

encrypted_PIN_block
Direction: Output
Type: String
The field that receives the 8-byte encrypted PIN block.

Restrictions
The restrictions for CSNBCPE.

The format control specified in the PIN profile must be NONE.

Required commands
The required commands for CSNBCPE.

This verb requires the Clear PIN Encrypt command (offset X'00AF') to be enabled in the active role.

An enhanced PIN security mode is available for formatting an encrypted PIN-block into IBM 3621 or 3624 format using the PADDIGIT PIN-extraction method. This mode limits checking of the PIN to decimal digits. No other PIN-block consistency checking will occur. To activate this mode, enable the PTR Enhanced PIN Security command (offset X'0313') in the active role.

Usage notes
Usage notes for CSNBCPE.

None

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBCPEJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBCPEJ are shown here.

Format
```java
public native void CSNBCPEJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] PADDIGIT_PIN_encrypting_key_identifier,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] clear_PIN,
    byte[] PADDIGIT_PIN_profile,
    byte[] PAN_data,
    hikmNativeInteger sequence_number,
    byte[] encrypted_PIN_block );
```
Clear PIN Generate (CSNBPGN)

Use the Clear PIN Generate verb to generate a clear PIN, a PIN validation value (PVV), or an offset according to an algorithm.

You supply the algorithm or process rule using the rule_array parameter.
- IBM 3624 (IBM-PIN or IBM-PINO)
- VISA PIN validation value (VISA-PVV)
- Interbank PIN (INBK-PIN)

For guidance information about VISA, see their appropriate publications.

Format

The format of CSNBPGN.

```plaintext
CSNBPGN(  
  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  PIN_generating_key_identifier,  
  rule_array_count,  
  rule_array,  
  PIN_length,  
  PIN_check_length,  
  data_array,  
  returned_result)
```

Parameters

The parameters for CSNBPGN.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

**PIN_generating_key_identifier**

Direction: Input/Output  
Type: String

The 64-byte key label or internal key token that identifies the PIN generation (PINGEN) key.

**rule_array_count**

Direction: Input  
Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1.

**rule_array**

Direction: Input  
Type: String

The process rule provides control information to the verb. The keyword is left-aligned in an 8-byte field and padded on the right with blanks. The rule_array keyword is described in Table 93 on page 409.
Table 93. Keywords for Clear PIN Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process Rule</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>GBP-PIN</td>
<td>The IBM German Bank Pool PIN, which uses the institution PINGEN key to generate an institution PIN (IPIN).</td>
</tr>
<tr>
<td>IBM-PIN</td>
<td>The IBM 3624 PIN, which is an institution-assigned PIN. It does not calculate the PIN offset.</td>
</tr>
<tr>
<td>IBM-PINO</td>
<td>The IBM 3624 PIN offset, which is a customer-selected PIN and calculates the PIN offset (the output).</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>The Interbank PIN that is generated.</td>
</tr>
<tr>
<td>VISA-PVV</td>
<td>The VISA PIN validation value. Input is the customer PIN.</td>
</tr>
</tbody>
</table>

**PIN_length**

*Direction: Input*

*Type: Integer*

The length of the PIN used for the IBM algorithms only, **IBM-PIN** or **IBM-PINO**. Otherwise, this parameter is ignored. Specify an integer from 4 - 16.

**PIN_check_length**

*Direction: Input*

*Type: Integer*

The length of the PIN offset used for the **IBM-PINO** process rule only. Otherwise, this parameter is ignored. Specify an integer from 4 - 16.

**Note:** The PIN check length must be less than or equal to the integer specified in the **PIN_length** parameter.

**data_array**

*Direction: Input*

*Type: String*

Three 16-byte data elements required by the corresponding **rule_array** parameter. The data array consists of three 16-byte fields or elements whose specification depends on the process rule. If a process rule only requires one or two 16-byte fields, the rest of the data array is ignored by the verb. **Table 94** describes the array elements.

Table 94. Array elements for the Clear PIN Generate verb

<table>
<thead>
<tr>
<th>Array element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear_PIN</td>
<td>Clear user selected PIN of 4 - 12 digits of 0 - 9. Left-aligned and padded with spaces. For <strong>IBM-PINO</strong>, this is the clear customer PIN (CSPIN).</td>
</tr>
<tr>
<td>Decimalization_table</td>
<td>Decimalization table for IBM and GBP only. Sixteen digits of 0 - 9.</td>
</tr>
<tr>
<td></td>
<td>If the ANSI X9.8 PIN - Use stored decimalization table only access control point (X'0356') is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.</td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td>For VISA only, the leftmost sixteen digits. Eleven digits of the personal account number (PAN). One digit key index. Four digits of customer selected PIN.</td>
</tr>
<tr>
<td></td>
<td>For Interbank only, sixteen digits. Eleven rightmost digits of the personal account number (PAN). A constant of 6. One digit key selector index. Three digits of PIN validation data.</td>
</tr>
</tbody>
</table>
**Clear PIN Generate (CSNBPGN)**

Table 94. Array elements for the Clear PIN Generate verb (continued)

<table>
<thead>
<tr>
<th>Array element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation_data</td>
<td>Validation data for IBM and IBM German Bank Pool padded to 16 bytes. One to sixteen characters of hexadecimal account data left-aligned and padded on the right with blanks.</td>
</tr>
</tbody>
</table>

Table 95 lists the data array elements required by the process rule (rule_array parameter). The numbers refer to the process rule’s position within the array.

Table 95. Array elements for Clear PIN Generate

<table>
<thead>
<tr>
<th>Process Rule</th>
<th>IBM-PIN</th>
<th>IBM-PINO</th>
<th>GBP-PIN</th>
<th>GBP-PINO</th>
<th>VISA-PVV</th>
<th>INBK-PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimalization_table</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation_data</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear_PIN</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Note: Generate offset for GBP algorithm is equivalent to IBM offset generation with PIN_check_length of 4 and PIN_length of 6.

**returned_result**

Direction: Output
Type: String
The 16-byte generated output, left-aligned, and padded on the right with blanks.

**Restrictions**

The restrictions for CSNBPGN.

None

**Required commands**

The required commands for CSNBPGN.

This verb requires the Clear PIN Generate - 3624 command (offset X'00A0') to be enabled in the active role.

Whenever the ANSI X9.8 PIN - Use stored decimalization table only command (offset X'0356') is enabled in the active role, the Decimalization_table element of the data_array value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables.

**Usage notes**

The usage notes for CSNBPGN.

If you are using the IBM 3624 PIN and IBM German Bank Pool PIN algorithms, you can supply an unencrypted customer selected PIN to generate a PIN offset.
Related information

Related information about CSNBPGN.

The algorithms are described in detail in Appendix E, “PIN formats and algorithms,” on page 683.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPGNJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBPGNJ are shown here.

Format

```java
public native void CSNBPGNJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] PIN_generating_key_identifier,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger PIN_length,
    hikmNativeInteger PIN_check_length,
    byte[] data_array,
    byte[] returned_result );
```
Clear PIN Generate Alternate (CSNBCPA)

Use the Clear PIN Generate Alternate verb to generate a clear VISA PVV (PIN validation value) from an input encrypted PIN block or to produce a 3624 offset from a customer-selected encrypted PIN.

The PIN block can be encrypted under either an input PIN-encrypting key (IPINENC) or an output PIN-encrypting key (OPINENC).

Format

The format of CSNBCPA.

```c
CSNBCPA(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    PIN_encryption_key_identifier,
    PIN_generation_key_identifier,
    PIN_profile,
    PAN_data,
    encrypted_PIN_block,
    rule_array_count,
    rule_array,
    PIN_check_length,
    data_array,
    returned_PVV)
```

Parameters

The parameters for CSNBCPA.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**PIN_encryption_key_identifier**

Direction: Input/Output
Type: String

A 64-byte string consisting of an internal token that contains an IPINENC or OPINENC key or the label of an IPINENC or OPINENC key that is used to encrypt the PIN block. If you specify a label, it must resolve uniquely to either an IPINENC or OPINENC key.

**PIN_generation_key_identifier**

Direction: Input/Output
Type: String

A 64-byte string that consists of an internal token that contains a PIN generation (PINGEN) key or the label of a PINGEN key.

**PIN_profile**

Direction: Input
Type: String

The three 8-byte character elements that contain information necessary to extract a PIN from a formatted PIN block. The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the Clear PIN Generate Alternate verb. See "The PIN profile" on page 399 for additional information.
Clear PIN Generate Alternate (CSNBCPA)

**PAN_data**
- **Direction:** Input
- **Type:** String

A 12-byte field that contains 12 characters of PAN data. The personal account number recovers the PIN from the PIN block if the PIN profile specifies ISO-0 or VISA-4 block formats. Otherwise it is ignored, but you must specify this parameter.

For ISO-0 or ISO-3, use the rightmost 12 digits of the PAN, excluding the check digit. For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

**encrypted_PIN_block**
- **Direction:** Input
- **Type:** String

An 8-byte field that contains the encrypted PIN that is input to the VISA PVV generation algorithm. The verb uses the IPINENC or OPINENC key that is specified in the PIN_encryption_key_identifier parameter to encrypt the block.

**rule_array_count**
- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1 or 2. If the default extraction method for a PIN block format is desired, specify the rule_array_count value as 1.

**rule_array**
- **Direction:** Input
- **Type:** String

The process rule for the PIN generation algorithm. Specify IBM-PINO or VISA-PVV (the VISA PIN verification value) in an 8-byte field, left-aligned, and padded with blanks. The rule_array points to an array of one or two 8-byte elements. The rule_array keywords are described in Table 96.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIN calculation method</td>
<td>(One required)</td>
</tr>
<tr>
<td>IBM-PINO</td>
<td>This keyword specifies use of the IBM 3624 PIN Offset calculation method.</td>
</tr>
<tr>
<td>VISA-PVV</td>
<td>This keyword specifies use of the VISA PVV calculation method.</td>
</tr>
</tbody>
</table>

**PIN extraction method** (One optional) See the text following this table.

If the PIN extraction method is provided, one of the PIN extraction method keywords shown in Table 87 on page 400 can be specified for the given PIN block format. See “PIN block format and PIN extraction method keywords” on page 400 for additional information. If the default extraction method for a PIN block format is desired, specify the rule_array_count value as 1.

The PIN extraction methods operate as follows:

**PINBLOCK**
- Specifies that the verb use one of the following:
Clear PIN Generate Alternate (CSNBCPA)

- The PIN length, if the PIN block contains a PIN length field
- The PIN delimiter character, if the PIN block contains a PIN delimiter character.

PADDIGIT
Specifies that the verb use the pad value in the PIN profile to identify the end of the PIN.

HEXDIGIT
Specifies that the verb use the first occurrence of a digit in the range from X'A' to X'F' as the pad value to determine the PIN length.

PINLEN
Specifies that the verb use the length specified in the keyword, where nn can range from 04 - 16, to identify the PIN.

The PINLENnn keywords are disabled for this verb by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the PTR Enhanced PIN Security command (bit X'0313') must be enabled using a TKE.

PADEXIST
Specifies that the verb use the character in the 16th position of the PIN block as the value of the pad value.

PIN_check_length
Direction: Input
Type: Integer
The length of the PIN offset used only for the IBM-PINO process rule. Otherwise, this parameter is ignored. Specify an integer from 4 - 16.

Note: The PIN check length must be less than or equal to the integer specified in the PIN_length parameter.

data_array
Direction: Input
Type: String
Three 16-byte elements, table 97 describes the format when IBM-PINO is specified. Table 98 on page 415 describes the format when VISA-PVV is specified.

Table 97. Array elements for Clear PIN Generate Alternate, data_array (IBM-PINO)

<table>
<thead>
<tr>
<th>Array element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimalization_table</td>
<td>This element contains the decimalization table of 16 characters (0 - 9) that are used to convert hexadecimal digits (X'0' - X'F') of the enciphered validation data to the decimal digits (X'0' - X'9'). If the ANSI X9.8 PIN - Use stored decimalization table only access control point (X'0356') is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.</td>
</tr>
<tr>
<td>validation_data</td>
<td>This element contains 1 - 16 characters of account data. The data must be left-aligned and padded on the right with space characters.</td>
</tr>
<tr>
<td>Reserved-3</td>
<td>This field is ignored, but you must specify it.</td>
</tr>
</tbody>
</table>
Table 98. Array elements for Clear PIN Generate Alternate, data_array (VISA-PVV)

<table>
<thead>
<tr>
<th>Array element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans_sec_parm</td>
<td>For VISA-PVV only, the leftmost twelve digits. Eleven digits of the personal account number (PAN). One digit key index. The rest of the field is ignored.</td>
</tr>
<tr>
<td>Reserved-2</td>
<td>This field is ignored, but you must specify it.</td>
</tr>
<tr>
<td>Reserved-3</td>
<td>This field is ignored, but you must specify it.</td>
</tr>
</tbody>
</table>

---

**returned_PVV**

**Direction:** Output  
**Type:** Character

A 16-byte area that contains the 4-byte PVV left-aligned and padded with blanks.

---

**Restrictions**

The restrictions for CSNBCPA.

None

---

**Required commands**

The required commands for CSNBCPA.

This verb requires the commands shown in the following table to be enabled in the active role based on the keyword specified for the PIN-calculation method:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-PINO</td>
<td>X'00A4'</td>
<td>Clear PIN Generate Alternate - 3624 Offset</td>
</tr>
<tr>
<td>VISA-PVV</td>
<td>X'00BB'</td>
<td>Clear PIN Generate Alternate - VISA PVV</td>
</tr>
</tbody>
</table>

---

An enhanced PIN security mode, on the CEX2C or CEX*C is available for extracting PINs from encrypted PIN blocks. This mode only applies when specifying a PIN-extraction method for an IBM 3621 or an IBM 3624 PIN-block. To do this, you must enable the PTR Enhanced PIN Security (offset X'0313') access control point in the default role. When activated, this mode limits checking of the PIN to decimal digits and a PIN length minimum of 4 is enforced. No other PIN-block consistency checking will occur.

An enhanced PIN security mode starting with CEX3C is available beginning with Release 4.1.0, to implement restrictions required by the ANSI X9.8 PIN standard. The restrictions are to accept only a PIN_profile variable that contains a PIN-block format of ISO-0 or ISO-3. To enforce these restrictions, you must enable the following access control points in the default role:

- ANSI X9.8 PIN - Enforce PIN block restrictions (X'0350')

For more information, see "ANSI X9.8 PIN restrictions" on page 397.

**Note:** A role with offset X'0350' enabled also affects access control of the Encrypted PIN Translate and the Secure Messaging for PINs verbs.

Whenever the ANSI X9.8 PIN - Use stored decimalization table only command (offset X'0356') is enabled in the active role, the Decimalization_table element of the data_array value must match one of the PIN decimalization tables that are in the
Clear PIN Generate Alternate (CSNBCPA)

active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables. The VISA-PVV PIN-calculation method does not have a Decimalization_table element and is therefore not affected by this command.

Usage notes

The usage notes for CSNBCPA.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCPAJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBCPAJ are shown here.

Format

```java
public native void CSNBCPAJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] inbound_PIN_encrypting_key_identifier,
    byte[] PIN_generating_key_identifier,
    byte[] input_PIN_profile,
    byte[] PAN_data,
    byte[] encrypted_PIN_block,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger PIN_check_length,
    byte[] data_array,
    byte[] returned_result);
```
CVV Generate (CSNBCSG)

Use the CVV Generate verb to generate a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2.

This verb generates a CVV that is based on the information that the PAN_data, the expiration_date, and the service_code parameters provide. This verb uses the Key-A and the Key-B keys to cryptographically process this information. Key-A and Key-B can be single-length DATA or MAC keys, or a combined Key-A, Key-B double length DATA or MAC key. If the requested CVV is shorter than 5 characters, the CVV is padded on the right by space characters. The CVV is returned in the 5-byte variable that the CVV_value parameter identifies. When you verify a CVV, compare the result to the value that the CVV_value supplies.

See CVV Key Combine (CSNBCKC) for information on combining two single-length MAC-capable keys into one double-length key.

Format

The format of CSNBCSG.

```
CSNBCSG(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    PAN_data,
    expiration_date,
    service_code,
    CVV_key_A_Identifier,
    CVV_key_B_Identifier,
    CVV_value)
```

Parameters

The parameters for CSNBCSG.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

*rule_array_count*

| Direction: Input  
| Type: Integer  
| A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0, 1, or 2.

*rule_array*

| Direction: Input  
| Type: String  
| Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The rule_array keywords are described in Table 99 on page 418
Table 99. Keywords for CVV Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAN data length (One, optional)</td>
<td></td>
</tr>
<tr>
<td>PAN-13</td>
<td>Specifies that the length of the PAN data is 13 bytes. <strong>PAN-13 is the default value.</strong></td>
</tr>
<tr>
<td>PAN-14</td>
<td>Specifies that the length of the PAN data is 14 bytes.</td>
</tr>
<tr>
<td>PAN-15</td>
<td>Specifies that the length of the PAN data is 15 bytes.</td>
</tr>
<tr>
<td>PAN-16</td>
<td>Specifies that the length of the PAN data is 16 bytes.</td>
</tr>
<tr>
<td>PAN-17</td>
<td>Specifies that the length of the PAN data is 17 bytes.</td>
</tr>
<tr>
<td>PAN-18</td>
<td>Specifies that the length of the PAN data is 18 bytes.</td>
</tr>
<tr>
<td>PAN-19</td>
<td>Specifies that the length of the PAN data is 19 bytes.</td>
</tr>
<tr>
<td>CVV length (One, optional)</td>
<td></td>
</tr>
<tr>
<td>CVV-1</td>
<td>Specifies that the CVV is to be computed as one byte, followed by four blanks. <strong>CVV-1 is the default value.</strong></td>
</tr>
<tr>
<td>CVV-2</td>
<td>Specifies that the CVV is to be computed as two bytes, followed by three blanks.</td>
</tr>
<tr>
<td>CVV-3</td>
<td>Specifies that the CVV is to be computed as three bytes, followed by two blanks.</td>
</tr>
<tr>
<td>CVV-4</td>
<td>Specifies that the CVV is to be computed as four bytes, followed by one blank.</td>
</tr>
<tr>
<td>CVV-5</td>
<td>Specifies that the CVV is to be computed as five bytes.</td>
</tr>
</tbody>
</table>

**PAN data**

**Direction:** Input  
**Type:** String

The `PAN_data` parameter specifies an address that points to the place in application data storage that contains personal account number (PAN) information in character form. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If the `PAN-nn` keyword is specified in the `rule_array`, where `nn` is a value between 13 and 19, then `nn` number of characters are processed.

If you specify the `PAN-nn` keyword in the `rule_array` where `nn` is less than 16, the server might copy 16 bytes to a work area. Therefore, ensure that the verb can address 16 bytes of storage.

**expiration_date**

**Direction:** Input  
**Type:** String

The `expiration_date` parameter specifies an address that points to the place in application data storage that contains the card expiration date in numeric character form in a 4-byte field. The application programmer must determine whether the CVV will be calculated with the date form of YYMM or MMYY.

**service_code**

**Direction:** Input  
**Type:** String

The `service_code` parameter specifies an address that points to the place in application data storage that contains the service code in numeric character form.
form in a 3-byte field. The service code is the number that the track-2 magnetic-stripe standards define. The service code of X'000' is supported.

**CVV_key_A_Identifier**

**Direction:** Input/Output  
**Type:** String

A 64-byte string that is the internal key token containing a single or double-length DATA or MAC key, or the label of a key storage record containing a single or double-length DATA or MAC key.

When this key is a double-length key, `CVV_key_B_identifier` must be 64 byte of binary zero. When a double-length MAC key is used, the CV bits 0 - 3 must indicate a CVVKEY-A key (B'0010').

A single-length key contains the key-A key that encrypts information in the CVV process. The left half of a double-length key contains the key-A key that encrypts information in the CVV process and the right half contains the key-B key that decrypts information.

**CVV_key_B_Identifier**

**Direction:** Input/Output  
**Type:** String

A pointer to a 64-byte internal key token or a key label of a single-length DATA or MAC key that decrypts information in the CCV process. The internal key token contains the Key-B key that decrypts information in the CVV process. When `CVV_key_A_identifier` is a double-length key, this parameter must be 64 byte of binary zero.

**CVV_value**

**Direction:** Output  
**Type:** String

A pointer to the location in application data storage that will be used to store the computed 5-byte character output value.

**Restrictions**

The restrictions for CSNBCSG.

None

**Required commands**

The required commands for CSNBCSG.

This verb requires the VISA CVV Generate command (offset X'00DF') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBCSG.

None
CVV Generate (CSNBCSG)

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCSGJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBCSGJ are shown here.

Format

```java
public native void CSNBCSGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] PAN_data,
    byte[] expiration_date,
    byte[] service_code,
    byte[] CVV_key_A_Identifier,
    byte[] CVV_key_B_Identifier,
    byte[] CVV_value);
```
Use the CVV Key Combine verb to combine two operational DES keys into one operational TDES key.

The verb accepts as input two single-length keys that are suitable for use with the CVV (card-verification value) algorithm. The resulting double-length key meets a more recent industry standard of using TDES to support PIN-based transactions. In addition, the double-length key is in a format that can be wrapped using the Key Export to TR31 verb.

The CVV Generate and CVV Verify verbs use the CVV algorithm to generate and verify card security codes required by Visa (CVV) and MasterCard (CVC). Previously, these verbs only accepted as input two single-length MAC-capable keys. These verbs will additionally accept as input a double-length MAC or MAC-capable DATA key that contains key-A as the left half of the key, and key-B as the right half of the key. The double-length key must be usable with either the CVV Generate verb, the CVV Verify verb, or both.

The CVV Key Combine verb allows combining most pairs of single-length DES keys that formerly functioned as a separate key-A and key-B into one double-length CVVKEY-A key. The CVVKEY-A attribute in the control vector is now changed to mean single-length CVV key containing key-A or double-length CVV key containing key-A and key-B.

To use this verb, specify the following:

- Up to two optional rule-array keywords:
  1. A key wrapping method keyword that specifies whether to use the new enhanced wrapping method, the original wrapping method, or the wrapping method defined as the default according to a configuration setting.
  2. A translation control keyword that restricts the translation method to the enhanced method.
- A single-length operational DES key for **key-A**
  Identify a single-length operational DES key that has a key type of MAC or DATA. The key identifier length must be 64, which is the length of a DES key-token or a key label. This parameter identifies the key-A key used with the CVV algorithm. It is placed in the left half of the double-length output key.
  When a MAC key is identified, it must have as its subtype extension ANY-MAC (CV bits 0 - 3 = B'0000') or CVVKEY-A (CV bits 0 - 3 = B'0010'). If a DATA key is identified, it must have its MAC generate bit on (CV bit 20), its MAC verify bit on (CV bit 21), or both bits on.
- A single-length operational DES key for **key-B**
  Identify a single-length operational DES key that has a key type of MAC or DATA. The key identifier length must be 64, which is the length of a DES key-token or a key label. This parameter identifies the key-B key used with the CVV algorithm. It is placed in the right half of the double-length output key.
  When a MAC key is identified, it must have as its subtype extension ANY-MAC (CV bits 0 - 3 = B'0000') or CVVKEY-B (CV bits 0 - 3 = B'0011'). If a DATA key is identified, it must have its MAC generate bit on (CV bit 20), its MAC verify bit on (CV bit 21), or both bits on.
- An output key identifier
  Identify a null key-token in a 64-byte buffer, or the key label of a DES null key-token. If the input parameter identifies a key label, the output key is placed in DES key-storage. otherwise, the output is returned in the buffer provided.
CVV Key Combine (CSNBCKC)

The following table shows the various output combinations that are returned for the MAC generate and MAC verify attributes. These results are based on the three possible MAC generate and MAC verify control-vector-bit combinations (bits 20 - 21) that the pair of input keys can have.

<table>
<thead>
<tr>
<th>CV bits 20 - 21 of input key key-A, single length</th>
<th>CV bits 20 - 21 of input key key-B, single length</th>
<th>MAC generate and MAC verify</th>
<th>MAC generate only, single length</th>
<th>MAC verify only</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC generate and MAC verify</td>
<td>MAC generate and MAC verify</td>
<td>MAC generate only, single length</td>
<td>MAC verify only, single length</td>
<td>MAC verify only</td>
</tr>
<tr>
<td>MAC generate only</td>
<td>MAC generate only</td>
<td>MAC generate only, single length</td>
<td>MAC verify only, single length</td>
<td>MAC verify only</td>
</tr>
<tr>
<td>MAC verify only</td>
<td>MAC verify only</td>
<td>MAC generate only, single length</td>
<td>MAC verify only, single length</td>
<td>MAC verify only</td>
</tr>
</tbody>
</table>

Invalid combination, control vector conflict

Format

The format of CSNBCKC.

```c
CSNBCKC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_a_identifier_length,
    key_a_identifier,
    key_b_identifier_length,
    key_b_identifier,
    output_key_identifier_length
    output_key_identifier)
```

Parameters

The parameters for CSNBCKC.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

**Direction:** Input  
**Type:** Integer  
A pointer to an integer variable containing the number of elements in the `rule_array` variable. The value must be 0, 1, or 2.

**rule_array**

**Direction:** Input  
**Type:** String array  
A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords are:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key wrapping method (one, optional)</td>
<td></td>
</tr>
</tbody>
</table>
CVV Key Combine (CSNBCKC)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>USECONFG</td>
<td>Specifies to wrap the key using the configuration setting for the default wrapping method. This keyword is ignored for AES keys. This is the default.</td>
</tr>
<tr>
<td>WRAP-ECB</td>
<td>Specifies to wrap the key using the legacy wrapping method.</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>Specifies to wrap the key using the enhanced wrapping method.</td>
</tr>
</tbody>
</table>

*Translation control* (optional). This is valid only with wrapping method WRAP-ENH or with USECONFG when the default wrapping method is WRAP-ENH. This option cannot be used on a key with a control vector valued to binary zeros.

| ENH-ONLY      | Specifies to restrict the key from being wrapped with the legacy wrapping method after it has been wrapped with the enhanced wrapping method. Sets bit 56 (ENH-ONLY) of the control vector to B'1'. |

There are restrictions on the available wrapping methods for the output key derived from the wrapping methods employed and control vector restrictions of the input keys. These are detailed in Table 100.

**Table 100. Key-wrapping matrix for the CVV Key Combine verb**

<table>
<thead>
<tr>
<th>key-A or key-B wrapped using WRAP-ENH method</th>
<th>key-A or key-B has ENH-ONLY bit on (CV bit 56 = B'1')</th>
<th>WRAP-ENH (by keyword or by default)</th>
<th>ENH-ONLY keyword</th>
<th>Resulting form of output key or error</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No to both</td>
<td>No</td>
<td>ECB wrapped</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No to both</td>
<td>No</td>
<td>Wrap type conflict, 8/2161 (X'871')</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes to either</td>
<td>No</td>
<td>WRAP-ENH, CV bit 56 = B'0' (NOT set)</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes to either</td>
<td>No</td>
<td>WRAP-ENH, CV bit 56 = B'1' (IS set)</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes to either</td>
<td>Yes</td>
<td>WRAP-ENH, CV bit 56 = B'1' (IS set)</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes to either</td>
<td>Yes</td>
<td>CV bit 56 conflict, 8/2111 (X'83F')</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No to both</td>
<td>Yes</td>
<td>CV bit 56 conflict, 8/2111 (X'83F')</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No to both</td>
<td>Yes</td>
<td>CV bit 56 conflict, 8/2111 (X'83F')</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No to both</td>
<td>Yes</td>
<td>CV bit 56 conflict, 8/2111 (X'83F')</td>
</tr>
</tbody>
</table>

**key_a_identifier_length**

*Direction: Input*

*Type: Integer*

A pointer to an integer variable containing the number of bytes in the key_a_identifier variable. This value must be 64.

**key_a_identifier**

*Direction: Input*

*Type: String*

A pointer to a string variable containing either the operational single-length DES key-token of the key-A key, or the label of such a key token. This key must be a MAC key or a DATA key that can perform MAC operations.
CVV Key Combine (CSNBCKC)

**key_b_identifier_length**
- **Direction:** Input
- **Type:** Integer
  - A pointer to an integer variable containing the number of bytes in the key_b_identifier variable. This value must be 64.

**key_b_identifier**
- **Direction:** Input
- **Type:** String
  - A pointer to a string variable containing either the operational single-length DES key-token of the key-B key, or the label of such a key token. This key must be a MAC key or a DATA key that can perform MAC operations.

**output_key_identifier_length**
- **Direction:** Input
- **Type:** Integer
  - A pointer to an integer variable containing the number of bytes in the output_key_identifier variable. This value must be 64.

**output_key_identifier**
- **Direction:** Input/Output
- **Type:** String
  - A pointer to a string variable containing a NULL key token, or the key label of a null DES key-token.

### Restrictions

The restrictions for CSNBCKC.

Input key-A and input key-B cannot have different export control bits (CV bit 17 and 57); these bits must match. Use the Prohibit Export verb to change XPORT-OK to NO-XPORT (CV bit 17), or the Restrict Key Attribute verb to change T31XPTOK to NOT31XPT (CV bit 57). These two bits are propagated to the output key-token.

### Required commands

The required commands for CSNBCKC.

The CVV Key Combine verb requires the CVV Key Combine command (offset X'0155') to be enabled in the active role.

In addition, these commands are required to be enabled in the active role, depending on the key-wrapping method keyword:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAP-ECB</td>
<td>X'0156'</td>
<td>CVV Key Combine - Allow Wrapping Override Keywords</td>
</tr>
<tr>
<td>WRAP-ENH</td>
<td>X'0156'</td>
<td>CVV Key Combine - Allow Wrapping Override Keywords</td>
</tr>
</tbody>
</table>

One additional restriction is related to combining the key-A and key-B pair of keys when there are mixed types. To permit the combination of mixed key types into a single set of types (ANY-MAC, CVVKEY-A, CVVKEY-B, and DATA), enable the CVV Key Combine - Permit Mixed Key Types command (offset X'0157') in the active role. See the table below for when this command is required:
### CVV Key Combine (CSNBCKC)

<table>
<thead>
<tr>
<th>Input key-A</th>
<th>ANY-MAC</th>
<th>CVVKEY-A</th>
<th>CVVKEY-B</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY-MAC</td>
<td>Always returns double-length CVVKEY-A key</td>
<td>Invalid combination, control vector conflict</td>
<td>Only returns double-length CVVKEY-A key if X'0157' enabled</td>
<td>Only returns double-length CVVKEY-A key if X'0157' enabled</td>
</tr>
<tr>
<td>CVVKEY-A</td>
<td>Only returns double-length CVVKEY-A key if X'0157' enabled, else access error</td>
<td>Invalid combination, control vector conflict</td>
<td>Always returns double-length CVVKEY-A key</td>
<td>Only returns double-length CVVKEY-A key if X'0157' enabled</td>
</tr>
<tr>
<td>CVVKEY-B</td>
<td>Invalid combination</td>
<td>Invalid combination</td>
<td>Invalid combination</td>
<td>Invalid combination</td>
</tr>
<tr>
<td>DATA</td>
<td>Only returns double-length CVVKEY-A key if X'0157' enabled, else access error</td>
<td>Invalid combination, control vector conflict</td>
<td>Only returns double-length CVVKEY-A key if X'0157' enabled, else access error</td>
<td>Always returns double-length CVVKEY-A key</td>
</tr>
</tbody>
</table>

### JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBCKCJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBCKCJ are shown here.

### Format

```java
public native void CSNBCKCJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_a_identifier_length,
    byte[] key_a_identifier,
    hikmNativeInteger key_b_identifier_length,
    byte[] key_b_identifier,
    hikmNativeInteger output_key_identifier_length,
    byte[] output_key_identifier);
```
Use the CVV Verify verb to verify a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2.

This verb generates a CVV based on the information the PAN_data, the expiration_date, and the service_code parameters provide. This verb uses the Key-A and the Key-B keys to cryptographically process this information. If the requested CVV is shorter than 5 characters, the CVV is padded on the right by space characters. The generated CVV is then compared to the value that the CVV_value supplies for verification.

See CVV Key Combine (CSNBCKC) for information on combining two single-length MAC-capable keys into one double-length key.

Format

The format of CSNBCSV.

```c
CSNBCSV(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    PAN_data,
    expiration_date,
    service_code,
    CVV_key_A_Identifier,
    CVV_key_B_Identifier,
    CVV_value)
```

Parameters

The parameters for CSNBCSV.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

```c
rule_array_count
    Direction: Input
    Type: Integer
    A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0, 1, or 2.

rule_array
    Direction: Input
    Type: String
    Keywords that provide control information to the verb. Each keyword is left-aligned in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. The rule_array keywords are described in Table 101.

Table 101. Keywords for CVV Verify control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAN data length</td>
<td>(One, optional)</td>
</tr>
</tbody>
</table>
Table 101. Keywords for CVV Verify control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAN-13</td>
<td>Specifies that the length of the PAN data is 13 bytes. <strong>PAN-13 is the default value.</strong></td>
</tr>
<tr>
<td>PAN-14</td>
<td>Specifies that the length of the PAN data is 14 bytes.</td>
</tr>
<tr>
<td>PAN-15</td>
<td>Specifies that the length of the PAN data is 15 bytes.</td>
</tr>
<tr>
<td>PAN-16</td>
<td>Specifies that the length of the PAN data is 16 bytes.</td>
</tr>
<tr>
<td>PAN-17</td>
<td>Specifies that the length of the PAN data is 17 bytes.</td>
</tr>
<tr>
<td>PAN-18</td>
<td>Specifies that the length of the PAN data is 18 bytes.</td>
</tr>
<tr>
<td>PAN-19</td>
<td>Specifies that the length of the PAN data is 19 bytes.</td>
</tr>
</tbody>
</table>

**CVV length (One, optional)**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVV-1</td>
<td>Specifies that the CVV is to be computed as one byte, followed by four blanks. <strong>CVV-1 is the default value.</strong></td>
</tr>
<tr>
<td>CVV-2</td>
<td>Specifies that the CVV is to be computed as two bytes, followed by three blanks.</td>
</tr>
<tr>
<td>CVV-3</td>
<td>Specifies that the CVV is to be computed as three bytes, followed by two blanks.</td>
</tr>
<tr>
<td>CVV-4</td>
<td>Specifies that the CVV is to be computed as four bytes, followed by one blank.</td>
</tr>
<tr>
<td>CVV-5</td>
<td>Specifies that the CVV is to be computed as five bytes.</td>
</tr>
</tbody>
</table>

**PAN_data**

**Direction:** Input  
**Type:** String

The **PAN_data** parameter specifies an address that points to the place in application data storage that contains personal account number (PAN) information in character form. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If the **PAN-nn** keyword is specified in the **rule_array**, where **nn** is a value between 13 and 19, then **nn** number of characters are processed.

If you specify the **PAN-nn** keyword in the **rule_array** where **nn** is less than 16, the server might copy 16 bytes to a work area. Therefore, ensure that the verb can address 16 bytes of storage.

**expiration_date**

**Direction:** Input  
**Type:** String

The **expiration_date** parameter specifies an address that points to the place in application data storage that contains the card expiration date in numeric character form in a 4-byte field. The application programmer must determine whether the CVV will be calculated with the date form of YYMM or MMYY.

**service_code**

**Direction:** Input  
**Type:** String

The **service_code** parameter specifies an address that points to the place in application data storage that contains the service code in numeric character form in a 3-byte field. The service code is the number that the track-2 magnetic-stripe standards define. The service code of X'000' is supported.
CVV Verify (CSNBCSV)

**CVV_key_A_Identifier**

*Direction: Input/Output*

*Type: String*

A 64-byte string that is the internal key token containing a single or double-length DATA or MAC key, or the label of a key storage record containing a single or double-length DATA or MAC key.

When this key is a double-length key, `CVV_key_B_Identifier` must be 64 byte of binary zero. When a double-length MAC key is used, the CV bits 0 - 3 must indicate a CVVKEY-A key (B’0010’).

A single-length key contains the key-A key that encrypts information in the CVV process. The left half of a double-length key contains the key-A key that encrypts information in the CVV process and the right half contains the key-B key that decrypts information.

**CVV_key_B_Identifier**

*Direction: Input/Output*

*Type: String*

A pointer to a 64-byte internal key token or a key label of a single-length DATA or MAC key that decrypts information in the CCV process. The internal key token contains the Key-B key that decrypts information in the CVV process.

When `CVV_key_A_Identifier` is a double-length key, this parameter must be 64 byte of binary zero.

**CVV_value**

*Direction: Input*

*Type: String*

The `CVV_value` parameter specifies an address that contains the CVV value which will be compared to the computed CVV value. This is a 5-byte field.

**Restrictions**

The restrictions for CSNBCSV.

None

**Required commands**

The required commands for CSNBCSV.

This verb requires the VISA CVV Verify command (offset X'00E0') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBCSV.

None

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBCSVJ.

See "Building Java applications to use with the CCA JNI" on page 26.
The parameters for CSNBCSVJ are shown here.

**Format**

```java
public native void CSNBCSVJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] PAN_data,
    byte[] expiration_date,
    byte[] service_code,
    byte[] CVV_key_A_Identifier,
    byte[] CVV_key_B_Identifier,
    byte[] CVV_value);
```

**CVV Verify (CSNBCCSV)**
Encrypted PIN Generate (CSNBEPG)

The Encrypted PIN Generate verb formats a PIN and encrypts the PIN block.

To generate the PIN, the verb uses one of the following PIN calculation methods:
- IBM 3624 PIN
- IBM German Bank Pool Institution PIN
- Interbank PIN

To format the PIN, the verb uses one of the following PIN block formats:
- IBM 3621 format
- IBM 3624 format
- ISO-0 format (same as the ANSI X9.8, VISA-1, and ECI-1 formats)
- ISO-1 format (same as the ECI-4 format)
- ISO-2 format
- ISO-3 format
- IBM 4704 encrypting PINPAD (4704-EPP) format
- VISA 2 format
- VISA 3 format
- VISA 4 format
- ECI-2 format
- ECI-3 format

Format

The format of CSNBEPG.

```
CSNBEPG(  return_code,  
         reason_code,  
         exit_data_length,  
         exit_data,  
         PIN_generating_key_identifier,  
         outbound_PIN_encrypting_key_identifier  
         rule_array_count,  
         rule_array,  
         PIN_length,  
         data_array,  
         PIN_profile,  
         PAN_data,  
         sequence_number  
         encrypted_PIN_block)
```

Parameters

The parameters for CSNBEPG.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**PIN_generating_key_identifier**

Direction: Input/Output
Type: String
Encrypted PIN Generate (CSNBEPG)

The 64-byte internal key token or a key label of an internal key token in the DES key storage file. The internal key token contains the PIN-generating key. The control vector must specify the PINGEN key type and have the EPINGEN usage bit set to B'1'.

**outbound_PIN_encrypting_key_identifier**

Direction: Input  
Type: String  
A 64-byte internal key token or a key label of an internal key token in the DES key storage file. The internal key token contains the key to be used to encrypt the formatted PIN and must contain a control vector that specifies the OPINENC key type and has the EPINGEN usage bit set to B'1'.

**rule_array_count**

Direction: Input  
Type: Integer  
A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1.

**rule_array**

Direction: Input  
Type: String  
Keywords that provide control information to the verb. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The rule_array keywords are described in Table 102.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GBP-PIN</td>
<td>This keyword specifies the IBM German Bank Pool Institution PIN calculation method is to be used to generate a PIN.</td>
</tr>
<tr>
<td>IBM-PIN</td>
<td>This keyword specifies the IBM 3624 PIN calculation method is to be used to generate a PIN.</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>This keyword specifies the Interbank PIN calculation method is to be used to generate a PIN.</td>
</tr>
</tbody>
</table>

**PIN_length**

Direction: Input  
Type: String  
A integer defining the PIN length for those PIN calculation methods with variable length PINs. Otherwise, the variable should be set to zero.

**data_array**

Direction: Input  
Type: Integer  
Three 16-byte character strings, which are equivalent to a single 48-byte string. The values in the data array depend on the keyword for the PIN calculation method. Each element is not always used, but you must always declare a complete data array. The numeric characters in each 16-byte string must be from 1 - 16 bytes in length, uppercase, left-aligned, and padded on the right with space characters. Table 103 on page 432 describes the array elements.
Table 103. Array elements for Encrypted PIN Generate data_array parameter

<table>
<thead>
<tr>
<th>Array element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimalization_table</td>
<td>Decimalization table for IBM and GBP only. Sixteen characters that are used to map the hexadecimal digits (X'0' - X'F') of the encrypted validation data to decimal digits (X'0' - X'9'). If the ANSI X9.8 PIN - Use stored decimalization table only command (offset X'0356') access control point is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.</td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td>For Interbank only, sixteen digits. Eleven rightmost digits of the personal account number (PAN). A constant of 6. One digit key selector index. Three digits of PIN validation data.</td>
</tr>
<tr>
<td>Validation_data</td>
<td>Validation data for IBM and IBM German Bank Pool padded to 16 bytes. 1-16 characters of hexadecimal account data left-aligned and padded on the right with blanks.</td>
</tr>
</tbody>
</table>

Table 104 lists the data array elements required by the process rule (rule_array parameter). The numbers refer to the process rule's position within the array.

Table 104. Keywords for Encrypted PIN Generate control information

<table>
<thead>
<tr>
<th>Process rule</th>
<th>IBM-PIN</th>
<th>GBP-PIN</th>
<th>INBK-PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimalization_table</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Validation_data</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

PIN_profile

Direction: Input
Type: Array

A 24-byte string containing the PIN profile including the PIN block format. See "The PIN profile" on page 399 for additional information.

PAN_data

Direction: Input
Type: String

A 12-byte string that contains 12 digits of Personal Account Number (PAN) data. The verb uses this parameter if the PIN profile specifies the ISO-0, ISO-3, or VISA-4 keyword for the PIN block format. Otherwise, ensure that this parameter is a 4-byte variable in application storage. The information in this variable will be ignored, but the variable must be specified.

Note: When using the ISO-0 or ISO-3 keywords, use the 12 rightmost digits of the PAN data, excluding the check digit. When using the VISA-4 keyword, use the 12 leftmost digits of the PAN data, excluding the check digit.

sequence_number

Direction: Input
Type: Integer

The 4-byte string that contains the sequence number used by certain PIN block formats. The verb uses this parameter if the PIN profile specifies the 3621 or 4704-EPP keyword for the PIN block format. Otherwise, ensure this parameter is a 4-byte variable in application storage. The information in the variable will be ignored, but the variable must be declared. To enter a sequence number, do the following:
Encrypted PIN Generate (CSNBEPG)

- Enter 99999 to use a random sequence number that the service generates.
- For the 3621 PIN block format, enter a value in the range from 0 - 65,535.
- For the 4704-EPP PIN block format, enter a value in the range from 0 - 255.

encrypted_PIN_block

Direction: Output
Type: String

The field where the verb returns the 8-byte encrypted PIN.

Restrictions

The restrictions for CSNBEPG.

The format control specified in the PIN profile must be NONE.

Required commands

The required commands for CSNBEPG.

This verb requires the commands, as shown in the following table, to be enabled in the active role based on the keyword specified for the PIN-calculation methods.

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-PIN</td>
<td>X'00B0'</td>
<td>Encrypted PIN Generate - 3624</td>
</tr>
<tr>
<td>GBP-PIN</td>
<td>X'00B1'</td>
<td>Encrypted PIN Generate - GBP</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>X'00B2'</td>
<td>Encrypted PIN Generate - Interbank</td>
</tr>
</tbody>
</table>

An enhanced PIN security mode is available for formatting an encrypted PIN block into IBM 3621 or 3624 format using the PADDIGIT PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of 4 is enforced; no other PIN-block consistency checking will occur. To activate this mode, enable the PTR Enhanced PIN Security Mode command (offset X'0313') in the active role.

Whenever the ANSI X9.8 PIN - Use stored decimalization table only command (offset X'0356') is enabled in the active role, the Decimalization_table element of the data_array value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables. The INBK-PIN PIN-calculation method does not have a Decimalization_table element and is therefore not affected by this command.

Usage notes

The usage notes for CSNBEPG.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBEPGJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBEPGJ are shown here.
Encrypted PIN Generate (CSNBEPG)

Format

```java
public native void CSNBEPGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] PIN_generating_key_identifier,
    byte[] outbound_PIN_encrypting_key_identifier,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger PIN_length,
    byte[] data_array,
    byte[] PIN_profile,
    byte[] PAN_data,
    hikmNativeInteger sequence_number,
    byte[] encrypted_PIN_block);
```
Encrypted PIN Translate (CSNBPTR)

Use the Encrypted PIN Translate verb to re-encipher a PIN block from one
PIN-encrypting key to another and, optionally, to change the PIN block format,
such as the pad digit or sequence number.

The unique-key-per-transaction key derivation for single and double-length keys is
available for the Encrypted PIN Translate verb. This support is available for the
input_PIN_encrypting_key_identifier and the output_PIN_encrypting_key_identifier
parameters for both REFORMAT and TRANSLAT process rules. The rule_array
keyword determines which PIN keys are derived keys.

The Encrypted PIN Translate verb can be used for unique-key-per-transaction key
derivation.

Format

The format of CSNBPTR.

```
CSNBPTR(  
  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  input_PIN_encrypting_key_identifier,  
  output_PIN_encrypting_key_identifier,  
  input_PIN_profile,  
  PAN_data_in,  
  PIN_block_in,  
  rule_array_count,  
  rule_array,  
  output_PIN_profile,  
  PAN_data_out,  
  sequence_number,  
  PIN_block_out)
```

Parameters

The parameters for CSNBPTR.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data
parameters, see "Parameters common to all verbs" on page 20.

**input_PIN_encrypting_key_identifier**

Direction: Input/Output
Type: String

The input PIN-encrypting key (IPINENC) for the PIN_block_in parameter
specified as a 64-byte internal key token or a key label. If keyword
UKPTOPIN, UKPTBOTH, DUKPT-IP, or DUKPT-BH is specified in the
rule_array parameter, the input_PIN_encrypting_key_identifier must specify a key
token or key label of a KEYGENKY with the UKPT usage bit enabled.

**output_PIN_encrypting_key_identifier**

Direction: Input/Output
Type: String

The output PIN-encrypting key (OPINENC) for the PIN_block_out parameter
specified as a 64-byte internal key token or a key label. If keyword
UKPTOPIN, UKPTBOTH, DUKPT-IP, or DUKPT-BH is specified in the
Encrypted PIN Translate (CSNBPTR)

The output_PIN_encrypting_key_identifier must specify a key token or key label of a KEYGENKY with the UKPT usage bit enabled.

**input_PIN_profile**

Direction: Input  
Type: String

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile depending on whether the PIN block is being enciphered or deciphered by the verb. See "The PIN profile" on page 399 for additional information.

If you choose the TRANSLAT processing rule or the REFORMAT processing rule in the rule_array parameter, the input PIN profile and output PIN profile can have different PIN block formats. If you specify UKPTIPIN with DUKPT-IP or UKPTBOTH with DUKPT-BH in the rule_array parameter, the input_PIN_profile is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 399 for additional information.

The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the Encrypted PIN Translate verb with a process rule (rule_array parameter) of REFORMAT. If the process rule is TRANSLAT, the pad digit is ignored.

The PINLENnn keywords are disabled for this verb by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the PTR Enhanced PIN Security access control point (bit X'0313') must be enabled using a TKE.

**PAN_data_in**

Direction: Input  
Type: String

The personal account number (PAN) if the process rule (rule_array parameter) is REFORMAT and the input PIN format is ISO-0, ISO-3, or VISA-4 only. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For ISO-0 or ISO-3, use the rightmost 12 digits of the PAN, excluding the check digit.

For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

**PIN_block_in**

Direction: Input  
Type: String

The 8-byte enciphered PIN block that contains the PIN to be translated.

**rule_array_count**

Direction: Input  
Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1, 2, or 3.

**rule_array**

Direction: Input  
Type: String
Encrypted PIN Translate (CSNBPTR)

The process rule for the verb is described in Table 105.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing rule</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>REFORMAT</td>
<td>Changes the PIN format, the contents of the PIN block, and the PIN-encrypting key.</td>
</tr>
<tr>
<td>TRANSLAT</td>
<td>Changes the PIN-encrypting key only. It does not change the PIN format and the contents of the PIN block.</td>
</tr>
<tr>
<td>PIN block format and PIN extraction method (Optional)</td>
<td>See “PIN block format and PIN extraction method keywords” on page 400 for additional information and a list of PIN block formats and PIN extraction method keywords. Note: If a PIN extraction method is not specified, the first one listed in Table 87 on page 400 for the PIN block format will be the default.</td>
</tr>
</tbody>
</table>

**DUKPT keywords - Single length key derivation** (One, optional)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UKPTIPIN</td>
<td>The input_PIN_encrypting_key_identifier is derived as a single length key. The input_PIN_encrypting_key_identifier must be a KEYGENKY key with the UKPT usage bit enabled. The input_PIN_profile must be 48 bytes and contain the key serial number.</td>
</tr>
<tr>
<td>UKPTOPIN</td>
<td>The output_PIN_encrypting_key_identifier is derived as a single length key. The output_PIN_encrypting_key_identifier must be a KEYGENKY key with the UKPT usage bit enabled. The output_PIN_profile must be 48 bytes and contain the key serial number.</td>
</tr>
<tr>
<td>UKPTBOTH</td>
<td>Both the input_PIN_encrypting_key_identifier and the output_PIN_encrypting_key_identifier are derived as a single length key. Both the input_PIN_encrypting_key_identifier and the output_PIN_encrypting_key_identifier must be KEYGENKY keys with the UKPT usage bit enabled. Both the input_PIN_profile and the output_PIN_profile must be 48 bytes and contain the respective key serial number.</td>
</tr>
</tbody>
</table>

**DUKPT keywords - double length key derivation** (One, optional)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUKPT-IP</td>
<td>The input_PIN_encrypting_key_identifier is derived as a double length key. The input_PIN_encrypting_key_identifier must be a KEYGENKY key with the UKPT usage bit enabled. The input_PIN_profile must be 48 bytes and contain the key serial number.</td>
</tr>
<tr>
<td>DUKPT-OP</td>
<td>The output_PIN_encrypting_key_identifier is derived as a double length key. The output_PIN_encrypting_key_identifier must be a KEYGENKY key with the UKPT usage bit enabled. The output_PIN_profile must be 48 bytes and contain the key serial number.</td>
</tr>
<tr>
<td>DUKPT-BH</td>
<td>Both the input_PIN_encrypting_key_identifier and the output_PIN_encrypting_key_identifier are derived as a double length key. Both the input_PIN_encrypting_key_identifier and the output_PIN_encrypting_key_identifier must be KEYGENKY keys with the UKPT usage bit enabled. Both the input_PIN_profile and the output_PIN_profile must be 48 bytes and contain the respective key serial number.</td>
</tr>
</tbody>
</table>

**output_PIN_profile**

**Direction:** Input  
**Type:** String  

The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile, depending on whether the PIN block is being enciphered or deciphered by the verb.

- If you choose the TRANSLAT processing rule in the rule_array parameter, the input_PIN_profile and the output_PIN_profile must specify the same PIN block format.
- If you choose the REFORMAT processing rule in the rule_array parameter, the input PIN profile and output PIN profile can have different PIN block formats.
Encrypted PIN Translate (CSNBPTR)

- If you specify UKPTOPIN or UKPTBOTH in the rule_array parameter, the output_PIN_profile is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 399 for additional information.
- If you specify DUKPT-OP or DUKPT-BH in the rule_array parameter, the output_PIN_profile is extended to a 48-byte field and must contain the current key serial number. See "The PIN profile" on page 399 for additional information.

**PAN_data_out**

**Direction:** Input  
**Type:** String

The personal account number (PAN) if the process rule (rule_array parameter) is **REFORMAT** and the output PIN format is **ISO-0**, **ISO-3**, or **VISA-4** only. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

For **ISO-0** or **ISO-3**, use the rightmost 12 digits of the PAN, excluding the check digit.

For **VISA-4**, use the leftmost 12 digits of the PAN, excluding the check digit.

**sequence_number**

**Direction:** Output  
**Type:** Integer

The sequence number if the process rule (rule_array parameter) is **REFORMAT** and the output PIN block format is 3621 or 4704-EPP only. Specify the integer value 99999. Otherwise, this parameter is ignored.

**PIN_block_out**

**Direction:** Input  
**Type:** String

The 8-byte output PIN block that is re-enciphered.

**Restrictions**

The restrictions for CSNBPTR.

None

**Required commands**

The required commands for CSNBPTR.

This verb requires the commands, as shown in the following table, to be enabled in the active role based on the keyword specified for the PIN-calculation methods.

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Input profile format control keyword</th>
<th>Output profile format control keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSLAT</td>
<td>NONE</td>
<td>NONE</td>
<td>X'00B3'</td>
<td>Encrypted PIN Translate - Translate</td>
</tr>
<tr>
<td>REFORMAT</td>
<td>NONE</td>
<td>NONE</td>
<td>X'00B7'</td>
<td>Encrypted PIN Translate - Reformat</td>
</tr>
</tbody>
</table>

This verb also requires the UKPT - PIN Verify_ PIN Translate command (offset X'00E1') to be enabled if you employ UKPT processing.
Encrypted PIN Translate (CSNBPTR)

Note: A role with offset X'00E1' enabled can also use the Encrypted PIN Verify verb with UKPT processing.

An enhanced PIN security mode is available for extracting PINs from a 3621 or 3624 encrypted PIN-block and formatting an encrypted PIN block into IBM 3621 or 3624 format using the PADDIGIT PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of 4 is enforced; no other PIN-block consistency checking will occur. To activate this mode, enable the PTR Enhanced PIN Security command (offset X'0313') in the active role.

The verb returns an error indicating that the PAD digit is not valid if all of these conditions are met:
1. The Enhanced PIN security mode command is enabled in the active role.
2. The output PIN profile specifies 3621 or 3624 as the PIN-block format.
3. The output PIN profile specifies a decimal digit (0 - 9) as the PAD digit.

Beginning with Release 4.1.0, three new commands are added (offsets X'0350', X'0351', and X'0352'). These three commands affect how PIN processing is performed as described below:
1. Enable the ANSI X9.8 PIN - Enforce PIN block restrictions command (offset X'0350') in the active role to apply additional restrictions to PIN processing implemented in CCA 4.1.0, as follows:
   • Do not translate or reformat a non-ISO PIN block into an ISO PIN block. Specifically, do not allow an IBM 3624 PIN-block format in the output_PIN_profile variable when the PIN-block format in the input_PIN_profile variable is not IBM 3624.
   • Constrain use of ISO-2 PIN blocks to offline PIN verification and PIN change operations in integrated circuit card environments only. Specifically, do not allow ISO-2 input or output PIN blocks.
   • Do not translate or reformat a PIN-block format that includes a PAN into a PIN-block format that does not include a PAN. Specifically, do not allow an ISO-1 PIN-block format in the output_PIN_profile variable when the PIN-block format in the input_PIN_profile variable is ISO-0 or ISO-3.
   • Do not allow a change of PAN data. Specifically, when performing translations between PIN block formats that both include PAN data, do not allow the input_PAN_data and output_PAN_data variables to be different from the PAN data enciphered in the input PIN block.

Note: A role with offset X'0350' enabled also affects access control of the Clear PIN Generate Alternate and the Secure Messaging for PINs verbs.

2. Enable the ANSI X9.8 PIN - Allow modification of PAN_01_0350 command (offset X'0351') in the active role to override the restriction to not allow a change of PAN data. This override is applicable only when either the ANSI X9.8 PIN - Enforce PIN block restrictions command (offset X'0350') or the ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350 command (offset X'0352') or both are enabled in the active role. This override is to support account number changes in issuing environments. Offset X'0351' has no effect if neither offset X'0350' nor offset X'0352' is enabled in the active role.

Note: A role with offset X'0351' enabled also affects access control of the Secure Messaging for PINs verbs.

3. Enable the ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350 command (offset X'0352') in the active role to apply a more restrictive variation of the
**Encrypted PIN Translate (CSNBPTR)**

ANSI X9.8 PIN - Enforce PIN block restrictions command (offset X'0350'). In addition to the previously described restrictions of offset X'0350', this command also restricts the `input_PIN_profile` and the `output_PIN_profile` to contain only ISO-0, ISO-1, and ISO-3 PIN block formats. Specifically, the IBM 3624 PIN-block format is not allowed with this command. Offset X'0352' overrides offset X'0350'.

**Note:** A role with offset X'0352' enabled also affects access control of the Secure Messaging for PINs verbs.

For more information, see “ANSI X9.8 PIN restrictions” on page 397.

**Usage notes**

The usage notes for CSNBPTR.

Some PIN block formats are known by several names. The following table shows the additional names.

<table>
<thead>
<tr>
<th>PIN format</th>
<th>Additional name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO-0</td>
<td>ANSI X9.8, VISA format 1, ECI format 1</td>
</tr>
<tr>
<td>ISO-1</td>
<td>ECI format 4</td>
</tr>
</tbody>
</table>

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBPTRJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBPTRJ are shown here.

**Format**

```java
public native void CSNBPTRJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] input_PIN_encrypting_key_identifier,
    byte[] output_PIN_encrypting_key_identifier,
    byte[] input_PIN_profile,
    byte[] input_PIN_block,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] output_PIN_profile,
    byte[] output_PAN_data,
    hikmNativeInteger sequence_number,
    byte[] output_PIN_block);
```
Encrypted PIN Verify (CSNPBVR)

Use the Encrypted PIN Verify verb to verify that one of the customer selected trial PINs is valid.

Use the verb to verify that one of the following PINs is valid:

- IBM 3624 (IBM-PIN)
- IBM 3624 PIN offset (IBM-PINO)
- IBM German Bank Pool (GBP-PIN)
- VISA PIN validation value (VISA-PVV)
- VISA PIN validation value (VISAPVV4)
- Interbank PIN (INBK-PIN)

The unique-key-par-transaction key derivation for single and double-length keys is available for the `input_PIN_encrypting_key_identifier` parameter.

Format

The format of CSNPBVR.

```
CSNPBVR(  
  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  input_PIN_encrypting_key_identifier,  
  PIN_verifying_key_identifier,  
  input_PIN_profile,  
  PAN_data,  
  encrypted_PIN_block,  
  rule_array_count,  
  rule_array,  
  PIN_check_length,  
  data_array)
```

Parameters

The parameters for CSNPBVR.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`input_PIN_encrypting_key_identifier`

Direction: Input/Output  
Type: String  

The 64-byte key label or internal key token containing the PIN-encrypting key (IPINENC) that enciphers the PIN block. If keyword UKPTIPIN or DUKPT-IP is specified in the `rule_array`, the `input_PIN_encrypting_key_identifier` must specify a key token or key label of a KEYGENKY with the UKPT usage bit enabled.

`PIN_verifying_key_identifier`

Direction: Input/Output  
Type: String  

The 64-byte key label or internal key token that identifies the PIN verify (PINVER) key.
input_PIN_profile

   Direction: Input
   Type: String

   The three 8-byte character elements that contain information necessary to either create a formatted PIN block or extract a PIN from a formatted PIN block. A particular PIN profile can be either an input PIN profile or an output PIN profile depending on whether the PIN block is being enciphered or deciphered by the verb. If you specify UKPTIPIN in the rule_array parameter, the input_PIN_profile is extended to a 48-byte field and must contain the current key serial number. See “The PIN profile” on page 399 for additional information.

   If you specify DUKPT-IP in the rule_array parameter, the input_PIN_profile is extended to a 48-byte field and must contain the current key serial number. See “The PIN profile” on page 399 for additional information.

   The pad digit is needed to extract the PIN from a 3624 or 3621 PIN block in the Encrypted PIN Verify verb.

   The PINLENnn keywords are disabled for this verb by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the PTR Enhanced PIN Security access control point (bit X'0313') must be enabled using a TKE workstation.

PAN_data

   Direction: Input
   Type: String

   The personal account number (PAN) is required for ISO-0, ISO-3, and VISA-4. Otherwise, this parameter is ignored. Specify 12 digits of account data in character format.

   For ISO-0 or ISO-3, use the rightmost 12 digits of the PAN, excluding the check digit.

   For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

encrypted_PIN_block

   Direction: Input
   Type: String

   The 8-byte enciphered PIN block that contains the PIN to be verified.

rule_array_count

   Direction: Input
   Type: Integer

   A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1, 2, or 3.

rule_array

   Direction: Input
   Type: String

   The process rule for the PIN verify algorithm, described in Table 107 on page 443.
### Encrypted PIN Verify (CSNBPVR)

**Table 107. Keywords for Encrypted PIN Verify control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm value</strong> (One, required)</td>
<td></td>
</tr>
<tr>
<td>GBP-PIN</td>
<td>The IBM German Bank Pool PIN. It verifies the PIN entered by the customer and compares that PIN with the institution generated PIN by using an institution key.</td>
</tr>
<tr>
<td>IBM-PIN</td>
<td>The IBM 3624 PIN, which is an institution-assigned PIN. It does not calculate the PIN offset.</td>
</tr>
<tr>
<td>IBM-PINO</td>
<td>The IBM 3624 PIN offset, which is a customer-selected PIN and calculates the PIN offset.</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>The Interbank PIN verify algorithm.</td>
</tr>
<tr>
<td>VISA-PVV</td>
<td>The VISA PIN verify value.</td>
</tr>
<tr>
<td>VISAPVV4</td>
<td>The VISA PIN verify value. If the length is 4 digits, normal processing for VISA-PVV will occur.</td>
</tr>
<tr>
<td><strong>PIN block format and PIN extraction method</strong> (Optional)</td>
<td></td>
</tr>
<tr>
<td>PIN block format and PIN extraction method keywords on page 400 for additional information and a list of PIN block formats and PIN extraction method keywords.</td>
<td></td>
</tr>
<tr>
<td>Note: If a PIN extraction method is not specified, the first one listed in Table 87 on page 400 for the PIN block format will be the default.</td>
<td></td>
</tr>
<tr>
<td>DUKPT keyword - single length key derivation (Optional)</td>
<td></td>
</tr>
<tr>
<td>UKPTIPIN</td>
<td>The input_PIN_encrypting_key_identifier is derived as a single length key. The input_PIN_encrypting_key_identifier must be a KEYGENKY key with the UKPT usage bit enabled. The input_PIN_profile must be 48 bytes and contain the key serial number.</td>
</tr>
<tr>
<td>DUKPT keyword - double length key derivation (Optional)</td>
<td></td>
</tr>
<tr>
<td>DUKPT-IP</td>
<td>The input_PIN_encrypting_key_identifier is to be derived using the DUKPT algorithm. The input_PIN_encrypting_key_identifier must be a KEYGENKY key with the DUKPT usage bit enabled. The input_PIN_profile must be 48 bytes and contain the key serial number.</td>
</tr>
</tbody>
</table>

**PIN_check_length**

**Direction:** Input  
**Type:** String  

The PIN check length for the IBM-PIN or IBM-PINO process rules only. Otherwise, it is ignored. Specify the rightmost digits, 4 - 16, for the PIN to be verified.

**data_array**

**Direction:** Input  
**Type:** Integer  

Three 16-byte elements required by the corresponding rule_array parameter. The data array consists of three 16-byte fields whose specification depend on the process rule. If a process rule requires only one or two 16-byte fields, the rest of the data array is ignored by the verb. Table 108 on page 444 describes the array elements.
Encrypted PIN Verify (CSNBPVR)

Table 108. Array elements for Encrypted PIN Verify data_array parameter

<table>
<thead>
<tr>
<th>Array element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimalization_table</td>
<td>Decimalization table for IBM and GBP only. Sixteen decimal digits of 0 - 9. If the ANSI X9.8 PIN - Use stored decimalization table only command (offset X'0356') access control point is enabled in the active role, this table must match one of the active decimalization tables in the coprocessors.</td>
</tr>
<tr>
<td>PIN_offset</td>
<td>Offset data for IBM-PINO. One to twelve numeric characters, 0 - 9, left-aligned and padded on the right with blanks. For IBM-PINO, the PIN offset length is specified in the PIN_check_length parameter. For IBM-PIN and GBP-PIN, the field is ignored.</td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td>For VISA, only the leftmost twelve digits of the 16-byte field are used. These consist of the rightmost eleven digits of the personal account number (PAN) and a one-digit key index. The remaining four characters are ignored. For Interbank only, all 16 bytes are used. These consist of the rightmost eleven digits of the PAN, a constant of X'6', a one-digit key index, and three numeric digits of PIN validation data.</td>
</tr>
<tr>
<td>RPVV</td>
<td>For VISA-PVV only, referenced PVV (four bytes) that is left-aligned. The rest of the field is ignored.</td>
</tr>
<tr>
<td>Validation_data</td>
<td>Validation data for IBM and GBP padded to 16 bytes. 1 - 16 characters of hexadecimal account data left-aligned and padded on the right with blanks.</td>
</tr>
</tbody>
</table>

Table 109 lists the data array elements required by the process rule (rule_array parameter). The numbers refer to the process rule’s position within the array.

Table 109. Array elements required by the process rule

<table>
<thead>
<tr>
<th>Process rule</th>
<th>IBM-PIN</th>
<th>IBM-PINO</th>
<th>GBP-PIN</th>
<th>VISA-PVV</th>
<th>INBK-PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimalization_table</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation_data</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIN_offset</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RPVV</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Restrictions
The restrictions for CSNBPVR.
None

Required commands
The required commands for CSNBPVR.

This verb requires the following commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-PIN, IBM-PINO</td>
<td>X'00AB'</td>
<td>Encrypted PIN Verify - 3624</td>
</tr>
<tr>
<td>GBP-PIN</td>
<td>X'00AC'</td>
<td>Encrypted PIN Verify - GBP</td>
</tr>
<tr>
<td>VISA-PVV, VISAPVV4</td>
<td>X'00AD'</td>
<td>Encrypted PIN Verify - VISA PVV</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>X'00AE'</td>
<td>Encrypted PIN Verify - Interbank</td>
</tr>
</tbody>
</table>
Encrypted PIN Verify (CSNBPVVR)

This verb also requires the UKPT - PIN Verify_PIN Translate command (offset X'00E1') to be enabled in the active role if you employ UKPT processing.

**Note:** A role with offset X'00E1' enabled can also use the Encrypted PIN Translate verb with UKPT processing.

An enhanced PIN security mode is available for extracting PINs from a 3621 or 3624 encrypted PIN-block using the **PADDIGIT** PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of four is enforced. No other PIN-block consistency checking will occur. To activate this mode, enable the PTR Enhanced PIN Security command (offset X'0313') in the active role.

Whenever the ANSI X9.8 PIN - Use stored decimmalization table only command (offset X'0356') command is enabled in the active role, the **Decimalization_table** element of the **data_array** value must match one of the PIN decimalization tables that are in the active state on the coprocessor. Use of this command provides improved security and control for PIN decimalization tables. The **VISA-PVV**, **VISAPVV4**, and **INBK-PIN** PIN calculation methods do not have a **Decimalization_table** element and are therefore not affected by this command.

### Usage notes

The usage notes for CSNBPVVR.

None

### Related information

Additional information about CSNBPVVR.

The algorithms are described in detail in Appendix E, “PIN formats and algorithms,” on page 683.

### JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPVVRJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBPVVRJ are shown here.

### Format

```java
public native void CSNBPVVRJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] PIN_encrypting_key_identifier,
    byte[] PIN_verifying_key_identifier,
    byte[] PIN_profile,
    byte[] PAN_data,
    byte[] encrypted_PIN_block,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger PIN_check_length,
    byte[] data_array);
```
The PIN Change/Unblock verb is used to generate a special PIN block to change the PIN accepted by an integrated circuit card (smartcard).

The special PIN block is based on the new PIN and the card-specific diversified key and, optionally, on the current PIN of the smartcard. The new PIN block is encrypted with a session key. The session key is derived in a two-step process. First, the card-specific diversified key (ICC Master Key) is derived using the TDES-ENC algorithm of the Diversified Key Generate verb. The session key is then generated according to the rule_array algorithm:

- **TDES-XOR** - XOR ICC Master Key with the Application Transaction Counter (ATC).
- **TDESEMV2** - use the EMV2000 algorithm with a branch factor of 2.
- **TDESEMV4** - use the EMV2000 algorithm with a branch factor of 4.

The generating DKYGENKY cannot have replicated halves. The encryption_issuer_master_key_identifier is a DKYGENKY that permits generation of a SMPIN key. The authentication_issuer_master_key_identifier is also a DKYGENKY that permits generation of a double length MAC key.

The PIN block format is specified by the VISA ICC Card specification: two mutually exclusive rule_array keywords, **VISAPCU1** and **VISAPCU2**. They refer to whether the current PIN is used in the generation of the new PIN. For **VISAPCU1**, it is not used, for **VISAPCU2** it is used.

**Format**

The format of CSNBPCU.

```c
CSNBPCU(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    authentication_issuer_master_key_length,
    authentication_issuer_master_key_identifier,
    encryption_issuer_master_key_length,
    encryption_issuer_master_key_identifier,
    key_generation_data_length,
    key_generation_data,
    new_reference_PIN_key_length,
    new_reference_PIN_key_identifier,
    new_reference_PIN_block,
    new_reference_PIN_profile,
    new_reference_PIN_PAN_data,
    current_reference_PIN_key_length,
    current_reference_PIN_key_identifier,
    current_reference_PIN_block,
    current_reference_PIN_profile,
    current_reference_PIN_PAN_data,
    output_PIN_data_length,
    output_PIN_data,
    output_PIN_profile,
    output_PIN_message_length,
    output_PIN_message)
```
Parameters

The parameters for CSNBPCU.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

**rule_array_count**

**Direction**: Input

**Type**: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1 or 2.

**rule_array**

**Direction**: Input

**Type**: String

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The keywords must be in contiguous storage. The rule_array keywords are described in Table 110.

Table 110. Keywords for PIN Change/Unblock control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>TDES-XOR</td>
<td>TDES encipher clear data to generate the intermediate (card-unique) key, followed by XOR of the final two bytes of each key with the ATC counter. This is the default.</td>
</tr>
<tr>
<td>TDESEMV2</td>
<td>Same processing as in the Diversified Key Generate verb.</td>
</tr>
<tr>
<td>TDESEMV4</td>
<td>Same processing as in the Diversified Key Generate verb.</td>
</tr>
<tr>
<td>PIN processing method</td>
<td>(One, required)</td>
</tr>
<tr>
<td>VISAPCU1</td>
<td>Form the new PIN from the new reference PIN and the intermediate (card-unique) key only.</td>
</tr>
<tr>
<td>VISAPCU2</td>
<td>Form the new PIN from the new reference PIN, the intermediate (card-unique) key and the current reference PIN.</td>
</tr>
</tbody>
</table>

**authentication_issuer_master_key_length**

**Direction**: Input

**Type**: Integer

The length of the authentication_issuer_master_key_identifier parameter. Currently, the value must be 64.

**authentication_issuer_master_key_identifier**

**Direction**: Input/Output

**Type**: String

The label name or internal token of a DKYGENKY key type that is to be used to generate the card-unique diversified key. The control vector of this key must be a DKYLO key that permits the generation of a double-length MAC key (DMAC). This DKYGENKY might not have replicated key halves.

**encryption_issuer_master_key_length**

**Direction**: Input

**Type**: Integer
PIN Change/Unblock (CSNBPCU)

The length of the encryption_issuer_master_key_identifier parameter. Currently, the value must be 64.

**encryption_issuer_master_key_identifier**

Direction: Input/Output  
Type: String

The label name or internal token of a DKYGENKY key type that is to be used to generate the card-unique diversified key and the secure messaging session key for the protection of the output PIN block. The control vector of this key must be a DKYL0 key that permits the generation of a SMPIN key type. This DKYGENKY might not have replicated key halves.

**key_generation_data_length**

Direction: Input  
Type: Integer

The length of the key_generation_data parameter. This value must be 10, 18, 26, or 34 bytes.

**key_generation_data**

Direction: Input  
Type: String

The data provided to generate the card-unique session key. For TDES-XOR, this consists of 8 or 16 bytes of data to be processed by TDES to generate the card-unique diversified key followed by a 16-bit ATC counter to offset the card-unique diversified key to form the session key. For TDESEMV2 and TDESEMV4, this can be 10, 18, 26, or 34 bytes. See “Diversified Key Generate (CSNBDKG)” on page 148 for more information.

**new_reference_PIN_key_length**

Direction: Input  
Type: Integer

The length of the new_reference_PIN_key_identifier parameter. Currently, the value must be 64.

**new_reference_PIN_key_identifier**

Direction: Input/Output  
Type: String

The label name or internal token of a PIN encrypting key that is to be used to decrypt the new_reference_PIN_block. This must be an IPINENC or OPINENC key. If the label name is supplied, the name must be unique in the DES key storage file.

**new_reference_PIN_block**

Direction: Input  
Type: String

This is an 8-byte field that contains the enciphered PIN block of the new PIN.

**new_reference_PIN_profile**

Direction: Input  
Type: String
PIN Change/Unblock (CSNBPCU)

This is a 24-byte field that contains three 8-byte elements with a PIN block format keyword, a format control keyword (NONE), and a pad digit as required by certain formats.

new_reference_PIN_PAN_data
Direction: Input
Type: String
This is a 12-byte field containing PAN in character format. This data might be needed to recover the new reference PIN if the format is ISO-0, ISO-3, or VISA-4. If neither is used, this parameter might be blanks.

For ISO-0 or ISO-3, use the rightmost 12 digits of the PAN, excluding the check digit. For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

current_reference_PIN_key_length
Direction: Input
Type: Integer
The length of the current_reference_PIN_key_identifier parameter. For the current implementation, the value must be 64. If the rule_array contains VISAPCU1, this value must be 0.

current_reference_PIN_key_identifier
Direction: Input/Output
Type: String
The label name or internal token of a PIN encrypting key that is to be used to decrypt the current_reference_PIN_block. This must be an IPINENC or OPINENC key. If the label name is supplied, the name must be unique in the key storage. If the rule_array contains VISAPCU1, this value is ignored.

current_reference_PIN_block
Direction: Input
Type: String
This is an 8-byte field that contains the enciphered PIN block of the new PIN. If the rule_array contains VISAPCU1, this value is ignored.

current_reference_PIN_profile
Direction: Input
Type: String
This is a 24-byte field that contains three 8-byte elements with a PIN block format keyword, a format control keyword (NONE), and a pad digit as required by certain formats. If the rule_array contains VISAPCU1, this value is ignored.

current_reference_PIN_PAN_data
Direction: Input
Type: String
This is a 12-byte field containing PAN in character format. This data might be needed to recover the new reference PIN if the format is ISO-0, ISO-3, or VISA-4. If neither is used, this parameter might be blanks. If the rule_array contains VISAPCU1, this value is ignored.
PIN Change/Unblock (CSNBPCU)

For ISO-0 or ISO-3, use the rightmost 12 digits of the PAN, excluding the check digit. For VISA-4, use the leftmost 12 digits of the PAN, excluding the check digit.

output_PIN_data_length
Direction: Input
Type: Integer
Currently this field is reserved. This value must be 0.

output_PIN_data
Direction: Input
Type: String
This parameter is ignored.

output_PIN_profile
Direction: Input
Type: String
This is a 24-byte field that contains three 8-byte elements with a PIN block format keyword (VISAPCU1 or VISCPU2), a format control keyword (NONE), and eight bytes of spaces.

output_PIN_message_length
Direction: Input/Output
Type: Integer
The length of the output_PIN_message field. Currently the value must be a minimum of 16.

output_PIN_message
Direction: Output
Type: String
The reformatted PIN block with the new reference PIN enciphered under the SMPIN session key.

Restrictions
The restrictions for CSNBPCU.
None

Required commands
The required commands for CSNBPCU.

This verb requires the following commands to be enabled in the active role based on the permissible key-type, IPINENC or OPINENC, used in the decryption of the input PIN blocks.
PIN Change/Unblock (CSNBPCU)

<table>
<thead>
<tr>
<th>PIN-block encrypting key-type</th>
<th>Offset</th>
<th>Command</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPINENC</td>
<td>X'00BC'</td>
<td>PIN Change/Unblock - change EMV PIN with OPINENC</td>
<td>Required if either the new_reference_PIN_key or the current_reference_PIN_key are permitted to be an OPINENC key type.</td>
</tr>
<tr>
<td>IPINENC</td>
<td>X'00BD'</td>
<td>PIN Change/Unblock - change EMV PIN with IPINENC</td>
<td>Required if either the new_reference_PIN_key or the current_reference_PIN_key are permitted to be an IPINENC key type.</td>
</tr>
</tbody>
</table>

When a MAC-MDK or an ENC-MDK of key type DKYGENKY is specified with control vector bits (19 - 22) of B’1111’, the Diversified Key Generate - DKYGENKY - DALL command (offset X’0290’) must also be enabled in the active role.

Note: A role with offset X’0290’ enabled can also use the Diversified Key Generate verb with a DALL key.

An enhanced PIN security mode is available for extracting PINs from a 3621 or 3624 encrypted PIN-block using the PADDIGIT PIN-extraction method. This mode limits checking of the PIN to decimal digits, and a minimum PIN length of 4 is enforced; no other PIN-block consistency checking will occur. To activate this mode, enable the PTR Enhanced PIN Security command (offset X’0313’) in the active role.

Usage notes

The usage notes for CSNBPCU.

There are additional access points for this verb.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBPCUJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBPCUJ are shown here.

Format

```java
public native void CSNBPCUJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger authenticationMasterKeyLength,
    byte[] authenticationMasterKey,
    hikmNativeInteger issuerMasterKeyLength,
    byte[] issuerMasterKey,
    hikmNativeInteger keyGenerationDataLength,
    byte[] keyGenerationData,
    hikmNativeInteger newRefPinKeyLength,
    byte[] newRefPinKey,
) { }
```
PIN Change/Unblock (CSNBPCU)

byte[] newRefPinBlock,
byte[] newRefPinProfile,
byte[] newRefPanData,
hikmNativeInteger currentRefPinKeyLength,
byte[] currentRefPinKey,
byte[] currentRefPinBlock,
byte[] currentRefPinProfile,
byte[] currentRefPanData,
hikmNativeInteger outputPinDataLength,
byte[] outputPinData,
byte[] outputPinProfile,
hikmNativeInteger outputPinMessageLength,
byte[] outputPinMessage);
Secure Messaging for Keys (CSNBSKY)

The Secure Messaging for Keys verb will encrypt a text block including a clear key value decrypted from an internal or external DES token.

The text block is normally a "Value" field of a secure message TLV (Tag/Length/Value) element of a secure message. TLV is defined in ISO/IEC 7816-4.

Format

The format of CSNBSKY.

```c
CSNBSKY(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    input_key_identifier,
    key_encrypting_key_identifier,
    secmsg_key_identifier,
    text_length,
    clear_text,
    initialization_vector,
    key_offset,
    key_offset_field_length,
    enciphered_text,
    output_chaining_vector)
```

Parameters

The parameters for CSNBSKY.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0 or 1.

`rule_array`

Direction: Input
Type: String

Keywords that provide control information to the verb. The processing method is the encryption mode used to encrypt the message. The `rule_array` keywords are described in [Table 111](#).

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enciphering mode</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>TDES-CBC</td>
<td>Use CBC mode to encipher the message. This is the default.</td>
</tr>
<tr>
<td>TDES-ECB</td>
<td>Use EBC mode to encipher the message.</td>
</tr>
</tbody>
</table>
Secure Messaging for Keys (CSNBSKY)

**input_key_identifier**
- **Direction:** Input/Output
- **Type:** String

  The internal token, external token, or key label of an internal token of a double length DES key. The key is recovered in the clear and placed in the text to be encrypted. The control vector of the DES key must not prohibit export.

**key_encrypting_key_identifier**
- **Direction:** Input/Output
- **Type:** String

  If the `input_key_identifier` is an external token, this parameter is the internal token or the key label of the internal token of IMPORTER or EXPORTER. If it is not, it is a null token. If a key label is specified, the key label must be unique.

**secmsg_key_identifier**
- **Direction:** Input/Output
- **Type:** String

  The internal token or key label of a secure message key for encrypting keys. This key is used to encrypt the updated `clear_text` containing the recovered DES key.

**text_length**
- **Direction:** Input
- **Type:** Integer

  The length of the `clear_text` parameter. Length must be a multiple of eight. Maximum length is 4096.

**clear_text**
- **Direction:** Input
- **Type:** String

  Clear text that contains the recovered DES key at the offset specified and is then encrypted. Any padding or formatting of the message must be done by the caller on input.

**initialization_vector**
- **Direction:** Input
- **Type:** String

  The 8-byte supplied string for the TDES-CBC mode of encryption. The `initialization_vector` is XORed with the first eight bytes of `clear_text` before encryption. This field is ignored for TDES-ECB mode.

**key_offset**
- **Direction:** Input
- **Type:** Integer

  The offset within the `clear_text` parameter at `key_offset` where the recovered clear `input_key_identifier` value is to be placed. The first byte of the `clear_text` field is offset 0.

**key_offset_field_length**
- **Direction:** Input
- **Type:** Integer
Secure Messaging for Keys (CSNBSKY)

The length of the field within `clear_text` parameter at `key_offset` where the recovered clear `input_key_identifier` value is to be placed. Length must be a multiple of eight and is equal to the key length of the recovered key. The key must fit entirely within the `clear_text`.

**enciphered_text**

Direction: Output  
Type: String  
The field where the enciphered text is returned. The length of this field must be at least as long as the `clear_text` field.

**output_chaining_vector**

Direction: Output  
Type: String  
This field contains the last eight bytes of enciphered text and is used as the `initialization_vector` for the next encryption call if data needs to be chained for TDES-CBC mode. No data is returned for TDES-ECB.

**Restrictions**

The restrictions for CSNBSKY.

None

**Required commands**

The required commands for CSNBSKY.

This verb requires the Secure Messaging for Keys command (offset X'0273') to be enabled in the active role.

**Usage notes**

The usage notes for CSNBSKY.

Keys appear in the clear only within the secure boundary of the cryptographic coprocessor, and never in host storage.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBSKYJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBSKYJ are shown here.

**Format**

```
public native void CSNBSKYJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] input_key_identifier,
    byte[] key_encrypting_key,
    byte[] session_key,
    hikmNativeInteger text_length,
    byte[] clear_text,
```
Secure Messaging for Keys (CSNBSKY)

```c
byte[] initialization_vector,
hikmNativeInteger key_offset,
hikmNativeInteger key_offset_field_length,
byte[] cipher_text,
byte[] output_chaining_value);
```
The Secure Messaging for PINs verb will encrypt a text block including a clear PIN block recovered from an encrypted PIN block.

The input PIN block will be reformatted if the block format in the input_PIN_profile is different from the block format in the output_PIN_profile. The clear PIN block will only be self encrypted if the SELFENC keyword is specified in the rule_array. The text block is normally a "Value" field of a secure message TLV (Tag/Length/Value) element of a secure message. TLV is defined in ISO/IEC 7816-4.

Format

The format of CSNBSPN.

```c
CSNBSPN(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    input_PIN_block,
    PIN_encrypting_key_identifier,
    input_PIN_profile,
    input_PAN_data,
    secmsg_key_identifier,
    output_PIN_profile,
    output_PAN_data,
    text_length,
    clear_text,
    initialization_vector,
    PIN_offset,
    PIN_offset_field_length,
    enciphered_text,
    output_chaining_vector)
```

Parameters

The parameters for CSNBSPN.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

rule_array_count

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0, 1, or 2.

rule_array

Direction: Input
Type: String

Keywords that provide control information to the verb. The processing method is the algorithm used to create the generated key. The keywords are left-aligned and padded on the right with blanks. The rule_array keywords are described in Table 112 on page 458.
Table 112: Keywords for Secure Messaging for PINs control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enciphering mode</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>TDES-CBC</td>
<td>Use CBC mode to encipher the message. This is the default.</td>
</tr>
<tr>
<td>TDES-ECB</td>
<td>Use EBC mode to encipher the message.</td>
</tr>
<tr>
<td><strong>PIN encryption</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>CLEARPIN</td>
<td>Recovered clear input PIN block (might be reformatted) is placed in the clear in the message for encryption with the secure message key. This is the default.</td>
</tr>
<tr>
<td>SELFENC</td>
<td>Recovered clear input PIN block (might be reformatted) is self-encrypted and then placed in the message for encryption with the secure message key.</td>
</tr>
</tbody>
</table>

**input_PIN_block**
- **Direction:** Input
- **Type:** String

The 8-byte input PIN block that is to be recovered in the clear and, perhaps, reformatted and then placed in the clear_text to be encrypted.

**PIN_encrypting_key_identifier**
- **Direction:** Input/Output
- **Type:** String

The internal token or key label of the internal token of the PIN encrypting key used in encrypting the input_PIN_block. The key must be an IPINENC key.

**input_PIN_profile**
- **Direction:** Input
- **Type:** String

The three 8-byte character elements that contain information necessary to extract the PIN from a formatted PIN block. The valid input PIN formats are ISO-0, ISO-1, ISO-2, and ISO-3. See "The PIN profile" on page 399 for additional information.

**input_PAN_data**
- **Direction:** Input
- **Type:** String

The 12 digit personal account number (PAN) if the input PIN format is ISO-0 or ISO-3. Otherwise, the parameter is ignored.

For ISO-0 or ISO-3, use the rightmost 12 digits of the PAN, excluding the check digit.

**secmsg_key_identifier**
- **Direction:** Input/Output
- **Type:** String

The internal token or key label of an internal token of a secure message key for encrypting PINs. This key is used to encrypt the updated clear_text.

**output_PIN_profile**
- **Direction:** Input
- **Type:** String
The three 8-byte character elements that contain information necessary to create a formatted PIN block. If reformatting is not required, the input_PIN_profile and the output_PIN_profile must specify the same PIN block format. Output PIN block formats supported are ISO-0, ISO-1, ISO-2, and ISO-3.

**output_PAN_data**

**Direction:** Input  
**Type:** String

The 12 digit personal account number (PAN) if the output PIN format is ISO-0 or ISO-3. Otherwise, this parameter is ignored.

For ISO-0 or ISO-3, use the rightmost 12 digits of the PAN, excluding the check digit.

**text_length**

**Direction:** Input  
**Type:** Integer

The length of the clear_text parameter that follows. Length must be a multiple of eight. Maximum length is 4096.

**clear_text**

**Direction:** Input  
**Type:** String

Clear text that contains the recovered and/or reformatted/encrypted PIN at offset specified and then encrypted. Any padding or formatting of the message must be done by the caller on input.

**initialization_vector**

**Direction:** Input  
**Type:** String

The 8-byte supplied string for the TDES-CBC mode of encryption. The initialization_vector is XORed with the first eight bytes of clear_text before encryption. This field is ignored for TDES-ECB mode.

**PIN_offset**

**Direction:** Input  
**Type:** Integer

The offset within the clear_text parameter where the reformatted PIN block is to be placed. The first byte of the clear_text field is offset 0.

**PIN_offset_field_length**

**Direction:** Input  
**Type:** Integer

The length of the field within clear_text parameter at PIN_offset where the recovered clear input_PIN_block value is to be placed. The PIN block might be self-encrypted if requested by the rule_array. Length must be eight. The PIN block must fit entirely within the clear_text.

**enciphered_text**

**Direction:** Output  
**Type:** String
Secure Messaging for PINs (CSNBSPN)

The field where the enciphered text is returned. The length of this field must be at least as long as the clear_text field.

output_chaining_vector
  Direction: Output
  Type: String
  This field contains the last eight bytes of enciphered text and is used as the initialization_vector for the next encryption call if data needs to be chained for TDES-CBC mode. No data is returned for TDES-ECB.

Restrictions
The restrictions for CSNBSPN.

None

Required commands
The required commands for CSNBSPN.

This verb requires the Secure Messaging for PINs command (offset X'0274') to be enabled in the active role.

Beginning with Release 4.1.0, three new commands are added (offsets X'0350', X'0351', and X'0352'). These three commands affect how PIN processing is performed as described below:

1. Enable the ANSI X9.8 PIN - Enforce PIN block restrictions command (offset X'0350') in the active role to apply additional restrictions to PIN processing implemented in CCA 4.1.0, as follows:
   - Constrain use of ISO-2 PIN blocks to offline PIN verification and PIN change operations in integrated circuit card environments only. Specifically, do not allow ISO-2 input or output PIN blocks.
   - Do not reformat a PIN-block format that includes a PAN into a PIN-block format that does not include a PAN.
   - Do not allow a change of PAN data. Specifically, when performing translations between PIN block formats that both include PAN data, do not allow the input_PAN_data and output_PAN_data variables to be different from the PAN data enciphered in the input PIN block.

   Note: A role with offset X'0350' enabled also affects access control of the Clear PIN Generate Alternate and the Encrypted PIN Translate verbs.

2. Enable the ANSI X9.8 PIN - Allow modification of PAN_01_0350 command (offset X'0351') in the active role to override the restriction to not allow a change of PAN data. This override is applicable only when either the ANSI X9.8 PIN - Enforce PIN block restrictions command (offset X'0350') or the ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350 command (offset X'0352') or both are enabled in the active role. This override is to support account number changes in issuing environments. Offset X'0351' has no effect if neither offset X'0350' nor offset X'0352' is enabled in the active role.

   Note: A role with offset X'0351' enabled also affects access control of the Encrypted PIN Translate verbs.

3. Enable the ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350 command (offset X'0352') in the active role to apply a more restrictive variation of the ANSI X9.8 PIN - Enforce PIN block restrictions command (offset X'0350').
Secure Messaging for PINs (CSNBSPN)

addition to the previously described restrictions of offset X'0350', this command also restricts the input_PIN_profile and the output_PIN_profile to contain only ISO-0, ISO-1, and ISO-3 PIN block formats. Specifically, the IBM 3624 PIN-block format is not allowed with this command. Offset X'0352' overrides offset X'0350'.

Note: A role with offset X'0352' enabled also affects access control of the Encrypted PIN Translate verbs.

For more information, see "ANSI X9.8 PIN restrictions" on page 397.

Usage notes
The usage notes for CSNBSPN.

Keys appear in the clear only within the secure boundary of the cryptographic coprocessors, and never in host storage.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBSPNJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNBSPNJ are shown here.

Format
public native void CSNBSPNJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] in_PIN_blk,
    byte[] in_PIN_enc_key_id,
    byte[] in_PIN_profile,
    byte[] in_PAN_data,
    byte[] secmsg_key,
    byte[] out_PIN_profile,
    byte[] out_PAN_data,
    hikmNativeInteger text_length,
    byte[] clear_text,
    byte[] initialization_vector,
    hikmNativeInteger PIN_offset,
    hikmNativeInteger PIN_offset_field_length,
    byte[] cipher_text,
    byte[] output_chaining_value);
Transaction Validation (CSNBTRV)

The Transaction Validation verb supports the generation and validation of American Express card security codes (CSC).

This verb generates and verifies transaction values based on information from the transaction and a cryptographic key. You select the validation method, and either the generate or verify mode, through rule_array keywords.

For the American Express process, the control vector supplied with the cryptographic key must indicate a MAC or MACVER class key. The key can be single or double length. DATAM and DATAMV keys are not supported. The MAC generate control vector bit must be on (bit 20) if you request CSC generation and MAC verify bit (bit 21) must be on if you request verification.

Format

The format of CSNBTRV.

```
CSNBTRV(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    transaction_key_identifier_length,
    transaction_key_identifier,
    transaction_info_length,
    transaction_info,
    validation_values_length,
    validation_values)
```

Parameters

The parameters for CSNBTRV.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

```
rule_array_count
Direction: Input
    Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1 or 2.
```

```
rule_array
Direction: Input
    Type: String

Keywords that provide control information to the verb. The keywords are left-aligned in an 8-byte field and padded on the right with blanks. The keywords must be in contiguous storage. The rule_array keywords are described in Table 113.
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>American Express card security codes</td>
<td>(One, required)</td>
</tr>
</tbody>
</table>
Table 113. Keywords for Transaction Validation control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSC-3</td>
<td>3-digit card security code (CSC) located on the signature panel. <strong>VERIFY</strong> implied. This is the default.</td>
</tr>
<tr>
<td>CSC-4</td>
<td>4-digit card security code (CSC) located on the signature panel. <strong>VERIFY</strong> implied.</td>
</tr>
<tr>
<td>CSC-5</td>
<td>5-digit card security code (CSC) located on the signature panel. <strong>VERIFY</strong> implied.</td>
</tr>
<tr>
<td>CSC-345</td>
<td>Generate 5-byte, 4-byte, or 3-byte values when given an account number and an expiration date. <strong>GENERATE</strong> implied.</td>
</tr>
<tr>
<td><strong>Operation</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td><strong>VERIFY</strong></td>
<td>Specifies verification of the value presented in the validation values variable.</td>
</tr>
<tr>
<td><strong>GENERATE</strong></td>
<td>Specifies generation of the value presented in the validation values variable.</td>
</tr>
</tbody>
</table>

**transaction_key_identifier_length**
- **Direction:** Input
- **Type:** Integer

The length of the **transaction_key_identifier** parameter.

**transaction_key_identifier**
- **Direction:** Input
- **Type:** String

The label name or internal token of a **MAC** or **MACVER** class key. The key can be single or double length.

**transaction_info_length**
- **Direction:** Input
- **Type:** Integer

The length of the **transaction_info** parameter. For the American Express CSC codes, the length must be 19.

**transaction_info**
- **Direction:** Input
- **Type:** String

For American Express, this is a 19-byte field containing the concatenation of the 4-byte expiration data (in the format YYMM) and the 15-byte American Express account number. Provide the information in character format.

**validation_values_length**
- **Direction:** Input/Output
- **Type:** Integer

The length of the **validation_values** parameter. Maximum value for this field is 64.

**validation_values**
- **Direction:** Input
- **Type:** String

This variable contains American Express CSC values. The data is output for **GENERATE** and input for **VERIFY**. See Table 114 on page 464.
Transaction Validation (CSNBTRV)

Table 114. Values for Transaction Validation validation_values parameter

<table>
<thead>
<tr>
<th>Operation</th>
<th>Element description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATE and CSC-345</td>
<td>5555544444333 where: 55555 = CSC 5 value 4444 = CSC 4 value 333 = CSC 3 value</td>
</tr>
<tr>
<td>VERIFY and CSC-3</td>
<td>333 = CSC 3 value</td>
</tr>
<tr>
<td>VERIFY and CSC-4</td>
<td>4444 = CSC 4 value</td>
</tr>
<tr>
<td>VERIFY and CSC-5</td>
<td>55555 = CSC 5 value</td>
</tr>
</tbody>
</table>

Restrictions
The restrictions for CSNBTRV.

None

Required commands
The required commands for CSNBTRV.

This verb requires the listed commands to be enabled in the active role, depending on the operation and card security code specified:

<table>
<thead>
<tr>
<th>Operation keyword</th>
<th>Card security code keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATE</td>
<td>CSC-345</td>
<td>X'0291'</td>
<td>Transaction Validation - Generate</td>
</tr>
<tr>
<td>VERIFY</td>
<td>CSC-3</td>
<td>X'0292'</td>
<td>Transaction Validation - Verify CSC-3</td>
</tr>
<tr>
<td></td>
<td>CSC-4</td>
<td>X'0293'</td>
<td>Transaction Validation - Verify CSC-4</td>
</tr>
<tr>
<td></td>
<td>CSC-5</td>
<td>X'0294'</td>
<td>Transaction Validation - Verify CSC-5</td>
</tr>
</tbody>
</table>

Usage notes
The usage notes for CSNBTRV.

There are additional access control points for this verb.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNBTRVJ.

See [“Building Java applications to use with the CCA JNI” on page 26.](#)

The parameters for CSNBTRVJJ are shown here.

Format
```java
public native void CSNBTRVJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
```

464 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Transaction Validation (CSNBTRV)

byte[] rule_array,
hkmNativeInteger transaction_key_length,
byte[] transaction_key,
hkmNativeInteger transaction_info_length,
byte[] transaction_info,
hkmNativeInteger validation_values_length,
byte[] validation_values);
Transaction Validation (CSNBTRV)
Chapter 11. Using digital signatures

This chapter describes the verbs that support using digital signatures to authenticate messages.

- “Digital Signature Generate (CSNDDSG)” on page 468
- “Digital Signature Verify (CSNDDSV)” on page 473
Digital Signature Generate (CSNDDSG)

This verb generates a digital signature using an RSA or ECC private key.

This verb supports the following methods:
- ANSI X9.30 (ECDSA)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

Note: The maximum signature length is 512 bytes (4096 bits).

The input text should have been previously hashed using either the One-Way Hash verb or the MDC Generate verb. If the signature formatting algorithm specifies ANSI X9.31, you must specify the hash algorithm used to hash the text (SHA-1 or RPMD-160). See “Formatting hashes and keys in public-key cryptography” on page 720.

You select the method of formatting the text through the rule_array parameter.

If the PKA_private_key_identifier specifies an RSA private key, you select the method of formatting the text through the rule_array parameter. If the PKA_private_key_identifier specifies an ECC private key, the ECC signature generated is according to ANSI X9.30.

Note: For PKCS the message digest and the message-digest algorithm identifier are combined into an ASN.1 value of type DigestInfo, which is BER-encoded to give an octet string D (see Table 115 on page 469). D is the text string supplied in the hash variable.

Format

The format of CSNDDSG.

```plaintext
CSNDDSG(    return_code,    reason_code,    exit_data_length,    exit_data,    rule_array_count,    rule_array,    PKA_private_key_identifier_length,    PKA_private_key_identifier,    hash_length,    hash,    signature_field_length,    signature_bit_length,    signature_field)
```

Parameters

The parameter definitions for CSNDDSG.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

rule_array_count
Digital Signature Generate (CSNDDSG)

Direction: Input  
Type: Integer  
A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 0, 1, 2, or 3.

rule_array  
Direction: Input  
Type: String  
Keywords that provide control information to the verb. A keyword specifies the method for calculating the digital signature. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The rule_array keywords are described in Table 115.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Digital signature formatting method</strong> (One, optional and not valid with ECDSA keyword.)</td>
<td></td>
</tr>
<tr>
<td>ISO-9796</td>
<td>Calculate the digital signature on the hash according to ISO-9796-1. Any hash method is allowed. This is the default.</td>
</tr>
<tr>
<td>PKCS-1.0</td>
<td>Calculate the digital signature on the BER-encoded ASN.1 value of the type DigestInfo containing the hash according to the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 00. The text must have been hashed and BER-encoded before input to this service.</td>
</tr>
<tr>
<td>PKCS-1.1</td>
<td>Calculate the digital signature on the BER-encoded ASN.1 value of the type DigestInfo containing the hash according to the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 01. The text must have been hashed and BER-encoded before input to this service.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>Format the hash by padding it on the left with binary zeros to the length of the RSA key modulus. Any supported hash function is allowed.</td>
</tr>
<tr>
<td>X9.31</td>
<td>Format according to the ANSI X9.31 standard. The input text must have been previously hashed with one of the hash algorithms specified below.</td>
</tr>
<tr>
<td><strong>Hash method specification</strong> (One, optional. Valid only with X9.31 digital-signature hash formatting method.)</td>
<td></td>
</tr>
<tr>
<td>RPMD-160</td>
<td>Hash the input text using the RIPEMD-160 hash method.</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Hash the input text using the SHA-1 hash method.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Hash the input text using the SHA-256 hash method.</td>
</tr>
<tr>
<td>SHA-384</td>
<td>Hash the input text using the SHA-384 hash method.</td>
</tr>
<tr>
<td>SHA-512</td>
<td>Hash the input text using the SHA-512 hash method.</td>
</tr>
<tr>
<td><strong>Token algorithm</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>ECDSA</td>
<td>Generate an ECC digital signature. This keyword was introduced with CCA 4.1.0. When specified, this is the only keyword permitted in the rule_array.</td>
</tr>
<tr>
<td>RSA</td>
<td>Generate an RSA digital signature. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
</tbody>
</table>

PKA_private_key_identifier_length  
Direction: Input  
Type: Integer  
The length of the PKA_private_key_identifier field. The maximum size is 3500 bytes.

PKA_private_key_identifier
Digital Signature Generate (CSNDDSG)

**Direction:** Input  
**Type:** String

An internal token or label of the RSA private key or retained key. If the signature format is **X9.31**, the modulus of the RSA key must have a minimum length of 1024 bits or greater. If the signature algorithm is **ECDSA**, this parameter must be a token or label of an ECC private key.

**hash_length**

**Direction:** Input  
**Type:** Integer

The length of the `hash` parameter in bytes. It must be the exact length of the text to sign. The maximum size is 512 bytes. If you specify **ZERO-PAD** in the `rule_array` parameter, the length is restricted to 36 bytes unless the RSA key is a signature only key, then the maximum length is 512 bytes.

On the IBM eServer zSeries 990 and subsequent releases, the hash length limit is controlled by a new access control point. Only RSA key management keys are affected by this access control point. The limit for RSA signature use only keys is 512 bytes. This new access control point is always disabled in the default role. You must have a TKE workstation to enable it.

**hash**

**Direction:** Input  
**Type:** String

The application-supplied text on which to generate the signature. The input text must have been previously hashed, and for PKCS formatting, it must be BER-encoded as previously described. For **X9.31**, the hash algorithms must have been either **SHA-1** or **RPMD-160**. See the `rule_array` parameter for more information.

**signature_field_length**

**Direction:** Input/Output  
**Type:** Integer

The length in bytes of the `signature_field` to contain the generated digital signature. The maximum size is 512 bytes.

For RSA, this must be at least the RSA modulus size (rounded up to a multiple of 32 bytes for the **X9.31** signature format, or one byte for all other signature formats).

For RSA, this field is updated with the minimum byte length of the digital signature.

For the **ECDSA** signature algorithm, R concatenated with S is the digital signature. The maximum output value will be 1042 bits (131 bytes). The size of the signature is determined by the size of P. Both R and S will have size P. For prime curves, the maximum size is 2 * 521 bits. For Brainpool curves, the maximum size is 2 * 512 bits.

**signature_bit_length**

**Direction:** Output  
**Type:** Integer

The bit length of the digital signature generated. For **ISO-9796** this is 1 less than the modulus length. For other RSA processing methods, this is the modulus length.
Digital Signature Generate (CSNDDSG)

signature_field

Direction: Output
Type: String

The digital signature generated is returned in this field. The digital signature is in the low-order bits (right-aligned) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-aligned within the signature_field. Any unused bytes to the right are undefined.

Restrictions

The restrictions for CSNDDSG.

Although ISO-9796 does not require the input hash to be an integral number of bytes in length, this verb requires you to specify the hash_length in bytes.

X9.31 requires the RSA token to have a minimum modulus bit length of 1024 bits, and the length must also be a multiple of 256 bits (or 32 bytes).

The length of the hash parameter in bytes must be the exact length of the text to sign. The maximum size is 256 bytes. If you specify ZERO-PAD in the rule_array parameter, the length is restricted to 36 bytes unless the RSA key is a signature only key, then the maximum length is 256 bytes.

The hash length limit is controlled by an access control point. If OFF (disabled), the maximum hash length limit for ZERO-PAD is the modulus length of the PKA private key. If ON (enabled), the maximum hash length limit for ZERO-PAD is 36 bytes. Only RSA key management keys are affected by this access control point. The limit for RSA signature use only keys is 256 bytes. This new access control point is always disabled in the default role. You must have a TKE workstation to enable it.

Required commands

The required commands for CSNDDSG.

This verb requires the Digital Signature Generate command (offset X'0100') to be enabled in the active role.

With the use of the DSG ZERO-PAD unrestricted hash length command (offset X'030C'), the hash-length restriction does not apply when using ZERO-PAD formatting.

Usage notes

The usage notes for CSNDDSG.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDDSGJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDDSGJ are shown here.
Digital Signature Generate (CSNDDSG)

Format

```java
public native void CSNDDSGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger PKA_private_key_identifier_length,
    byte[] PKA_private_key_identifier,
    hikmNativeInteger hash_length,
    byte[] hash,
    hikmNativeInteger signature_field_length,
    hikmNativeInteger signature_bit_length,
    byte[] signature_field);
```
Digital Signature Verify (CSNDDSV)

This verb verifies a digital signature using an RSA or ECC public key.

This verb verifies digital signatures generated with these methods:
- ANSI X9.30 (ECDSA)
- ANSI X9.31 (RSA)
- ISO 9796-1 (RSA)
- RSA DSI PKCS 1.0 and 1.1 (RSA)
- Padding on the left with zeros (RSA)

This verb can use the RSA or ECC public key, depending on the digital signature algorithm used to generate the signature.

This verb can also use the public keys that are contained in trusted blocks, regardless of whether the block also contains rules to govern its use when generating or exporting keys with the Remote Key Export verb. The format of the trusted block enables Digital Signature Verify to distinguish it from other RSA key tokens, and therefore no special rule array keyword or other parameters are required in order to indicate that the trusted block is being used. However, if the Digital Signature Generate verb is used with the TPK-ONLY keyword in the rule_array, an error will occur if the PKA_public_key_identifier does not contain a trusted block.

Input text should have been previously hashed. You can use the One-Way Hash verb. See also “Formatting hashes and keys in public-key cryptography” on page 720.

Note: The maximum signature length is 256 bytes (2048 bits).

Format

The format of CSNDDSV.

```c
CSNDDSV(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    PKA_public_key_identifier_length,
    PKA_public_key_identifier,
    hash_length,
    hash,
    signature_field_length,
    signature_field)
```

Parameters

The parameters for CSNDDSV.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

`rule_array_count`
Digital Signature Verify (CSNDDSV)

**Direction:** Input  
**Type:** Integer  
A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0, 1, 2, or 3.

**rule_array**  
**Direction:** Input  
**Type:** String  
Keywords that provide control information to the verb. A keyword specifies the method to use to verify the digital signature. Each keyword is left-aligned in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The `rule_array` keywords are described in Table 116.

### Table 116. Keywords for Digital Signature Verify control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Digital signature formatting method</strong> (Optional and not valid with ECDSA keyword.)</td>
<td></td>
</tr>
<tr>
<td>ISO-9796</td>
<td>Verify the digital signature on the hash according to ISO-9796-1. Any hash method is allowed. This is the default.</td>
</tr>
<tr>
<td>PKCS-1.0</td>
<td>Verify the digital signature on the BER-encoded ASN.1 value of the type DigestInfo as specified in the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 00. The text must specify BER encoded hash text.</td>
</tr>
<tr>
<td>PKCS-1.1</td>
<td>Verify the digital signature on the BER-encoded ASN.1 value of the type DigestInfo as specified in the RSA Data Security, Inc. Public Key Cryptography Standards #1 block type 01. The text must specify BER encoded hash text.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>Format the hash by padding it on the left with binary zeros to the length of the PKA key modulus. Any supported hash function is allowed.</td>
</tr>
<tr>
<td>X9.31</td>
<td>Format according to ANSI X9.31 standard.</td>
</tr>
<tr>
<td><strong>Trusted public key restriction</strong> (Optional. Not valid with ECDSA keyword. Valid only with trusted blocks. See <a href="#">“Trusted blocks” on page 647</a>)</td>
<td></td>
</tr>
<tr>
<td>TPK-ONLY</td>
<td>Permits the use of only public keys contained in trusted blocks. By specifying this keyword, the use of regular CCA RSA key tokens is rejected and only the use of a (trusted) public key supplied by the <code>PKA_public_key_identifier</code> parameter can be used to verify the digital signature, thus assuring a sensitive signature verification operation is limited to trusted public keys. If TPK-ONLY is specified, the <code>PKA_public_key_identifier</code> parameter must identify a trusted block that contains two sections after the trusted block token header: (1) trusted block trusted RSA public key (section X'11'), and (2) trusted block information (section X'14'). Section X'14' is required for all trusted blocks. Section X'11' contains the trusted public key, and its usage rules must indicate it can be used in digital signature operations.</td>
</tr>
<tr>
<td><strong>Token algorithm</strong> (One, optional)</td>
<td></td>
</tr>
<tr>
<td>ECDSA</td>
<td>Verify an ECC digital signature. This keyword was introduced with CCA 4.1.0. When specified, this is the only keyword permitted in the <code>rule_array</code>.</td>
</tr>
<tr>
<td>RSA</td>
<td>Verify an RSA digital signature. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
</tbody>
</table>

**PKA_public_key_identifier_length**  
**Direction:** Input  
**Type:** Integer  
The length of the `PKA_public_key_identifier` field containing the public key token or label. The maximum size is 3500 bytes.
Digital Signature Verify (CSNDDSV)

**Direction:** Input
**Type:** String

A token or label of the RSA public key or internal trusted block. If this parameter contains a token or the label of an internal trusted block, the `rule_array` parameter must specify `TPK-ONLY`. If the signature algorithm is `ECDSA`, this must be a token label or an ECC public key.

**hash_length**
**Direction:** Input
**Type:** Integer

The length of the `hash` parameter in bytes. It must be the exact length of the text that was signed. The maximum size is 512 bytes.

**hash**
**Direction:** Input
**Type:** String

The application-supplied text on which the supplied signature was generated. The text must have been previously hashed and, for PKCS formatting, BER-encoded as previously described.

**signature_field_length**
**Direction:** Input
**Type:** Integer

The length in bytes of the `signature_field` parameter. The maximum size is 512 bytes.

**signature_field**
**Direction:** Input
**Type:** String

This field contains the digital signature to verify. The digital signature is in the low-order bits (right-aligned) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-aligned within the `signature_field`.

**Restrictions**

The restrictions for CSNDDSV.

The ability to recover a message from a signature (which ISO-9796 allows but does not require) is **not** supported.

The exponent of the RSA public key must be odd.

Although ISO-9796 does not require the input hash to be an integral number of bytes in length, this service requires you to specify the `hash_length` in bytes.

X9.31 requires the RSA token to have a minimum modulus bit length of 1024, and the length must also be a multiple of 256 bits (or 32 bytes).

**Required commands**

The required commands for CSNDDSV.
Digital Signature Verify (CSNDDSV)

This verb requires the Digital Signature Verify command (offset X’0101’) to be enabled in the active role.

Usage notes
The usage notes for CSNDDSV.

None

Related information
Additional information for CSNDDSV.

Trusted Block Create (CSNDTBC), Remote Key Export (CSNDRKX)

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDDSVJ.

See "Building Java applications to use with the CCA JNI on page 26."

The parameters for CSNDDSVJ are shown here.

Format
```
public native void CSNDDSVJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger PKA_public_key_identifier_length,
    byte[] PKA_public_key_identifier,
    hikmNativeInteger hash_length,
    byte[] hash,
    hikmNativeInteger signature_field_length,
    byte[] signature_field);
```
Chapter 12. Managing PKA cryptographic keys

This topic describes the verbs that generate and manage PKA keys.

- “PKA Key Generate (CSNDPKG)” on page 478
- “PKA Key Import (CSNDPKI)” on page 484
- “PKA Key Token Build (CSNDPKB)” on page 488
- “PKA Key Token Change (CSNDKTC)” on page 497
- “PKA Key Translate (CSNDPKT)” on page 500
- “PKA Public Key Extract (CSNDPKX)” on page 504
- “Remote Key Export (CSNDRKX)” on page 506
- “Trusted Block Create (CSNDTBC)” on page 517
PKA Key Generate (CSNDPKG)

Use the PKA Key Generate verb to generate RSA keys for use on the cryptographic coprocessor or other CCA systems, or ECC keys for use starting with CEX3C.

Input to the PKA Key Generate verb is either a skeleton key token that has been built by the PKA Key Token Build verb, or a valid internal token. In the case of a valid internal token, the verb will generate a key with the same modulus length and the same exponent. In the case of a valid internal ECC token, PKA Key Generate will generate a key based on the curve type and size. Internal tokens with a X'09' section are not supported.

RSA key generation requires the following information in the input skeleton token:

- Size of the modulus in bits. The modulus for Modulus-Exponent format keys is between 512 and 1024 bits in length. The CRT modulus is between 512 and 4096 bits in length. The modulus for the variable-length Modulus-Exponent format is between 512 and 4096 bits in length.

RSA key generation has the following restrictions:

- For Modulus-Exponent, there are restrictions on the modulus, public exponent, and private exponent.
- For CRT, there are restrictions on $dp$, $dq$, $U$, and the public exponent.

See the Key value structure in “PKA Key Token Build (CSNDPKB)” on page 488 for a summary of restrictions.

ECC key generation requires this information in the skeleton token:

- The key type: ECC
- The type of curve: Prime or Brainpool
- The size of $p$ in bits: 192, 224, 256, 384 or 521 for Prime curves and 160, 192, 224, 256, 320, 384, or 512 for Brainpool curves
- Key usage information
- Optionally, application associated data

The generated ECC private key will be returned in one of the following forms:

- Clear key
- Encrypted key enciphered under the APKA-MK
- Encrypted key enciphered by an AES transport key
PKA Key Generate (CSNDPKG)

Format

The format of CSNDPKG.

```c
CSNDPKG( 
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    regeneration_data_length,
    regeneration_data,
    skeleton_key_identifier_length,
    skeleton_key_identifier,
    transport_key_identifier,
    generated_key_token_length,
    generated_key_token)
```

Parameters

The parameter definitions for CSNDPKG.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1, 2, or 3.

`rule_array`

**Direction:** Input  
**Type:** String

A keyword that provides control information to the verb. A keyword is left-aligned in an 8-byte field and padded on the right with blanks. The `rule_array` keywords are described in Table 117.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Private key encryption (One, required)</strong></td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td>Return the private key in clear text. The private key in clear text is an external token. This keyword is valid only for RSA and ECC keys.</td>
</tr>
<tr>
<td>MASTER</td>
<td>Encipher the private key or OPK using the PKA master-key for an RSA key, or the OPK using the APKA master-key for an ECC key. The <code>transport_key_identifier</code> parameter should specify a null key-token. The keyword is not supported if a skeleton token with a 09 section is provided.</td>
</tr>
</tbody>
</table>

Table 117. Keywords for PKA Key Generate control information
### Keywords for PKA Key Generate control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETAIN</td>
<td>Retains the private key within the cryptographic engine and returns the public key. This is only valid for RSA signature keys. Because of this, the RETAIN keyword is not supported for:</td>
</tr>
<tr>
<td></td>
<td>• A skeleton token with a X'09' section provided.</td>
</tr>
<tr>
<td></td>
<td>• An ECC token.</td>
</tr>
<tr>
<td></td>
<td>Before using this keyword, see the information about retained keys in “Using retained keys” on page 387.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Take special notice on the types of skeleton key tokens that can be passed. The PKA Key Token Build verb will, of course, let you create many more types of skeleton key tokens than can be used to generate retained keys, because this is the minority of supported function.</td>
</tr>
<tr>
<td>XPORT</td>
<td>Enciphers the private key under the IMPORTER or EXPORTER key-encrypting-key identified by the transport_key_identifier parameter. For an RSA key, this is an EXPORTER or IMPORTER transport key in a fixed-length operational DES key-token. For an ECC key, this is an EXPORTER or IMPORTER transport key in an operational variable-length AES key-token. This keyword is valid only for RSA and ECC keys.</td>
</tr>
<tr>
<td>RETAIN option</td>
<td>(one, optional). Valid only with the RETAIN keyword.</td>
</tr>
<tr>
<td>CLONE</td>
<td>Mark a generated and retained private key as usable in cryptographic engine cloning process. This keyword is supported only if RETAIN is also specified. Only valid for RSA keys. The keyword is not supported for:</td>
</tr>
<tr>
<td></td>
<td>• A skeleton token with a X'09' section provided</td>
</tr>
<tr>
<td></td>
<td>• An ECC token</td>
</tr>
<tr>
<td>Regeneration data option</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>ITER-38</td>
<td>Force 38 iterations of tests for primality, as required by ANSI X9.31 for the Miller-Rabin primality tests. This option produces a more secure key, but it is labor intensive. This keyword is invalid for ECC key generation. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>Transport key-type</td>
<td>(one, optional; one required if transport_key_identifier is a label). If this keyword is specified, it must match the type of key to be transported, whether the identifier is a label or not.</td>
</tr>
<tr>
<td>OKEK-AES</td>
<td>The outbound key-encrypting key represents an AES key-token.</td>
</tr>
<tr>
<td>OKEK-DES</td>
<td>The outbound key-encrypting key represents a DES key-token. This is the default.</td>
</tr>
</tbody>
</table>

**regeneration_data_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the regeneration_data variable. This parameter must be 0 for ECC tokens. For RSA tokens, the value must be 8 - 512.

If the value is 0, the generated keys are based on a random-seed value. If this value is between 8 - 256, the regeneration data is hashed to form a seed value used in the key generation process to provide a means for recreating a public-private key pair.

**regeneration_data**

**Direction:** Input  
**Type:** String
This field points to a string variable containing a string used as the basis for creating a particular public-private key pair in a repeatable manner. The regeneration data is hashed to form a seed value used in the key generation process and provides a means for recreating a public-private key pair.

**skeleton_key_identifier_length**

**Direction:** Input  
**Type:** Integer

The length of the `skeleton_key_identifier` parameter in bytes. The maximum allowed value is 3500 bytes.

**skeleton_key_identifier**

**Direction:** Input  
**Type:** String

A pointer to the application-supplied skeleton key token generated by PKA Key Token Build, or the label of the token that contains the required modulus length and public exponent for RSA key generation, or the required curve type and bit length for ECC key generation.

If RETAIN was specified and the `skeleton_key_identifier` is a label, the label must match the private key name of the key. For RSA keys, the `skeleton_key_identifier` parameter must contain a token that specifies a modulus length in the range 512 - 4096 bits.

**transport_key_identifier**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing an operational AES or DES key-encrypting-key token, a null key-token, or a key label of such a key. Use an **IMPORTER** key to encipher a private key to be used at this node. Use an **EXPORTER** key to encipher a private key to be used at another node. Choose one of the following:

- When generating an ECC key with the **XPORT** rule-array keyword, provide the variable-length symmetric **IMPORTER** or **EXPORTER** key-token to be used to wrap the generated ECC key. Key bit lengths of 128, 192, and 256 are supported. If this parameter points to a key label, specify rule-array keyword **OKEK-AES** to indicate that the AES key-storage dataset contains the key token.

- When generating an RSA key with the **XPORT** rule-array keyword, provide the fixed-length DES **IMPORTER** or **EXPORTER** key-token to be used to wrap the generated RSA key. If this parameter points to a key label, specify rule-array keyword **OKEK-DES** to indicate that the DES key-storage dataset contains the key token.

- If the **XPORT** rule-array keyword is not specified, specify a null key-token. If this parameter points to a key label, specify keyword **OKEK-AES** for an ECC key or keyword **OKEK-DES** for an RSA key.

**generated_key_token_length**

**Direction:** Input/Output  
**Type:** Integer

The length of the generated key token. The field is checked to ensure that it is at least equal to the size of the token being returned. The maximum size is 3500 bytes. On output, this field is updated with the actual token length.
PKA Key Generate (CSNDPKG)

**generated_key_token**

*Direction: Input/Output*

*Type: String*

The internal token or label of the generated RSA or ECC key. When generating an RSA retained key, on output the verb returns the public token in this variable.

If the key label identifies a key record in PKA key-storage:

- A record must already exist in the PKA key storage file with this same label or the verb will fail.
- The generated key token replaces any key token associated with the label.
- The `generated_key_token_length` returned to the application will be the same as the input length.

If the first byte of the identified string does not indicate a key label (that is, not in the range X'20' - X'FE'), and the variable is of sufficient length to receive the result, then the generated key token is returned in the identified variable.

**Restrictions**

The restrictions for CSNDPKG.

- The maximum public exponent is 17 bits for any key that has a modulus greater than 2048 bits.
- Not all IBM implementations of CCA support a CRT form of the RSA private key; check the product-specific literature. The IBM implementations support an optimized RSA private key (a key in Chinese Remainder Theorem format). The formats vary between versions.
- See “PKA key tokens” on page 65 for the formats used when generating the various forms of key tokens.
- When generating a key for use with ANSI X9.31 digital signatures, the modulus length must be: 1024, 1280, 1536, 1792, 2048, or 4096 bits.
- The key label used for a retained key must not exist in the external PKA key-storage held on the hard disk drive.
- Due to potential loss of a retained private key within the cryptographic engine, retained keys should be avoided for key management purposes.
- 2048-bit RSA keys may have a public exponent in the range of 1 - 256 bytes.
- 4096-bit RSA key public exponents are restricted to the values 3 and 65537.

**Required commands**

The required commands for CSNDPKG.

- This verb requires the PKA Key Generate command (offset X'0103') to be enabled in the active role.
- With the CLONE rule-array keyword, enable the PKA Clone Key Generate command (offset X'0204').
- With the CLEAR rule-array keyword, enable the PKA Key Generate - Clear command (offset X'0205') in the hardware.
- To generate ECC keys with the CLEAR rule-array keyword, this verb requires the Generate ECC keys in the clear command (offset X'0326') to be enabled in the active role.
- To generate keys based on the value supplied in the `regeneration_data` variable, you must enable one of these commands:
PKA Key Generate (CSNDPKG)

- When not using the RETAIN keyword, enable the PKA Key Generate - Permit Regeneration Data command (offset X'027D').
- When using the RETAIN keyword, enable the PKA Key Generate - Permit Regeneration Data Retain command (offset X'027E').

- To disallow the wrapping of a key with a weaker key-encrypting key, enable the Disallow Weak Key Wrap command (offset X'0328') in the active role. This command affects multiple verbs. See Appendix G, "Access control points and verbs," on page 723.

- To receive a warning when wrapping a key with a weaker key-encrypting key, enable the Warn when Wrapping Weak Keys command (offset X'032C') in the active role. The Disallow Weak Key Wrap command (offset X'0328') overrides this command.

Usage notes

Usage notes for CSNDPKG.

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKGJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDPKGJ are shown here.

Format

```java
public native void CSNDPKGJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger regeneration_data_length,
    byte[] regeneration_data,
    hikmNativeInteger skeleton_key_token_length,
    byte[] skeleton_key_token,
    byte[] transport_key_identifier,
    hikmNativeInteger generated_key_identifier_length,
    byte[] generated_key_identifier);
```
PKA Key Import (CSNDPKI)

This verb imports an external PKA or ECC private key token. (This consists of a
PKA or ECC private key and public key.)

The secret values of the key can be:
- Clear
- Encrypted under a limited-authority DES importer key if the source_key_identifier
  is an RSA token
- Encrypted under an AES Key Encryption Key if the source_key_identifier is an
  ECC token

This verb can also import a clear PKA key. The PKA Key Token Build verb creates
a clear PKA key token.

This verb can also import an external trusted block token for use with the Remote
Key Export verb.

Output of this verb is a CCA internal token of the RSA or ECC private key or
trusted block.

Format
The format of CSNDPKI.

```c
CSNDPKI(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    source_key_identifier_length,
    source_key_identifier,
    importer_key_identifier,
    target_key_identifier_length,
    target_key_identifier)
```

Parameters
The parameter definitions for CSNDPKI.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data
parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`
  Direction: Input
  Type: Integer
  
  A pointer to an integer variable containing the number of elements in the
  rule_array variable. This value must be 0 or 1.

`rule_array`
  Direction: Input
  Type: String
PKA Key Import (CSNDPKI)

The rule_array parameter is a pointer to a string variable containing a keyword. The keyword is 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are described in Table 118.

Table 118. Keywords for PKA Key Import control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token type</td>
<td>(One, optional)</td>
</tr>
<tr>
<td>ECC</td>
<td>Specifies that the key being imported is an ECC key. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>RSA</td>
<td>Specifies that the key being imported is an RSA key or a trusted block. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
</tbody>
</table>

**source_key_identifier_length**

Direction: Input
Type: Integer

The length of the source_key_identifier parameter. The maximum size is 3500 bytes.

**source_key_identifier**

Direction: Input
Type: String

Contains an external token or label of a PKA private key, without section identifier X’14’ (Trusted Block Information), or the trusted block in external form as produced by the Trusted Block Create verb with the ACTIVATE keyword.

If a PKA private key without the section identifier X’14’ is passed in:

- There are no qualifiers. A retained key can not be used.
- The key token must contain both public-key and private-key information. The private key can be in cleartext or it can be enciphered. ECC tokens must contain a private key in cleartext.
- This is the output of the PKA Key Generate (CSNDPKG) verb or the PKA Key Token Build (CSNDPKB) verb.
- If encrypted, the key was created on another platform.

If a PKA private key with the section identifier X’14’ is passed in:

- This verb will be used to encipher the MAC key within the trusted block under the PKA master key instead of the IMP-PKA key-encrypting key.
- The importer_key_identifier must contain an IMP-PKA KEK.

**importer_key_identifier**

Direction: Input/Output
Type: String

A variable-length field containing an AES or DES key identifier used to wrap the imported key. For RSA keys and trusted blocks, this must be a DES limited authority transport key (IMP-PKA). For ECC keys, this must be an AES transport key.

This parameter contains one of the following:

- 64-byte label of a key storage record that contains the transport key.
- 64-byte DES internal key token containing the transport key.
PKA Key Import (CSNDPKI)

- A variable-length AES internal key token containing the transport key.
  This parameter is ignored for clear tokens.

target_key_identifier_length
  Direction: Input/Output
  Type: Integer
  The length of the target_key_identifier parameter. The maximum size is 3500 bytes. On output, and if the size is of sufficient length, the variable is updated with the actual length of the target_key_identifier field.

target_key_identifier
  Direction: Input/Output
  Type: String
  This field contains the internal token or label of the imported PKA private key or a trusted block. If a label is specified on input, a PKA key storage record with this label must exist. The PKA key storage record with this label will be overwritten with the imported key unless the existing record is a retained key. If the record is a retained key, the import will fail. A retained key record cannot be overwritten. If no label is specified on input, this field is ignored and should be set to binary zeros on input.

Restrictions
The restrictions for CSNDPKI.

This verb imports RSA keys of up to 4096 bits. However, the hardware configuration sets the limits on the modulus size of keys for digital signatures and key management; thus, the key can be successfully imported but fail when used if the limits are exceeded.

The importer_key_identifier parameter is a limited-authority key-encrypting key.

CRT form tokens with a private section ID of X'05' cannot be imported.

Required commands
The required commands for CSNDPKI.

This verb requires the PKA Key Import command (offset X'0104') to be enabled in the active role. If the source_key_token parameter points to a trusted block, also enable the PKA Key Import - Import an External Trusted Key Block to internal form command (offset X'0311').

Usage notes
The usage notes for CSNDPKI.

This verb imports keys of any modulus size up to 2048 bits. However, the hardware configuration sets the limits on the modulus size of keys for digital signatures and key management; thus, the key can be successfully imported but fail when used if the limits are exceeded.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDPKIJ.
PKA Key Import (CSNDPKI)

This verb has a Java Native Interface (JNI) version, which is named CSNDPKIJ. See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDPKIJ are shown here.

**Format**

```java
public native void CSNDPKIJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger source_key_token_length,
    byte[] source_key_token,
    byte[] transport_key_identifier,
    hikmNativeInteger target_key_identifier_length,
    byte[] target_key_identifier);
```
PKA Key Token Build (CSNDPKB)

Use this verb to build external PKA key tokens containing unenciphered private RSA or ECC keys.

You can use this token as input to the PKA Key Import verb to obtain an operational internal token containing an enciphered private key. This verb builds a skeleton token that you can use as input to the PKA Key Generate verb (see Table 117 on page 479). You can also input to this verb a clear unenciphered public RSA or ECC key and return the public key in a token format that other PKA verbs can use directly.

This verb is used to create the following:
- A skeleton_key_token for use with the PKA Key Generate verb.
- A key token with a public key that has been obtained from another source.
- A key token with a clear private-key and the associated public key.
- A key token for an RSA private key in optimized Chinese Remainder Theorem (CRT) format.
- An RSA token with X'09' section identifier using the RSAMEVAR keyword to obtain a token for a key in Modulus-Exponent format that is variable length.

ECC key generation requires this information in the skeleton token:
- The key type: ECC
- The type of curve: Prime or Brainpool
- The size of p in bits: 192, 224, 256, 384 or 521 for Prime curves and 160, 192, 224, 256, 320, 384, or 521 for Brainpool curves
- Key usage information
- Optionally, application associated data

Format

The format of CSNDPKB.

```plaintext
CSNDPKB(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  rule_array_count,
  rule_array,
  key_value_structure_length,
  key_value_structure,
  private_key_name_length,
  private_key_name,
  user_definable_associated_data_length,
  user_definable_associated_data,
  reserved_2_length,
  reserved_2,
  reserved_3_length,
  reserved_3,
  reserved_4_length,
  reserved_4,
  reserved_5_length,
  reserved_5,
  key_token_length,
  key_token)
```
PKA Key Token Build (CSNDPKB)

Parameters

The parameters for CSNDPKB.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see “Parameters common to all verbs” on page 20.

rule_array_count

Direction: Input
Type: Integer
A pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1, 2, or 3.

rule_array

Direction: Input
Type: String
One or two keywords that provide control information to the verb. The keywords must be in contiguous storage with each of the keywords left-aligned in its own 8-byte location and padded on the right with blanks. The rule_array keywords are described in Table 119.

Table 119. Keywords for PKA Key Token Build control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key type (One, required)</td>
<td></td>
</tr>
<tr>
<td>ECC-PAIR</td>
<td>This keyword indicates building a token containing both public and private ECC key information. The parameter key_value_structure identifies the input key values, if supplied.</td>
</tr>
<tr>
<td>ECC-PUBL</td>
<td>This keyword indicates building a token containing public ECC key information. The parameter key_value_structure identifies the input values, if supplied.</td>
</tr>
<tr>
<td>RSA-CRT</td>
<td>This keyword indicates building a token containing an RSA private key in the optimized Chinese Remainder Theorem (CRT) format. The parameter key_value_structure identifies the input key values, if supplied.</td>
</tr>
<tr>
<td>RSA-PRIV</td>
<td>This keyword indicates building a token containing both public and private RSA key information. The parameter key_value_structure identifies the input key values, if supplied.</td>
</tr>
<tr>
<td>RSA-PUBL</td>
<td>This keyword indicates building a token containing public RSA key information. The parameter key_value_structure identifies the input values, if supplied.</td>
</tr>
<tr>
<td>RSAMEVAR</td>
<td>This keyword indicates RSA-Modulus Exponent-Variant (RSAMEVAR), a type X'09' key token for RSA, named VAR_OPK. Note: Key tokens created with this key type cannot be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.</td>
</tr>
<tr>
<td>Key usage control (One, optional)</td>
<td></td>
</tr>
<tr>
<td>KEY-MGMT</td>
<td>Indicates that an RSA or ECC private key can be used in both the Symmetric Key Import and the Digital Signature Generate verbs. Note: Key tokens created with this key usage cannot be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.</td>
</tr>
<tr>
<td>KM-ONLY</td>
<td>Indicates that an RSA or ECC private key can be used only in symmetric key distribution. Note: Key tokens created with this key usage cannot be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.</td>
</tr>
<tr>
<td>SIG-ONLY</td>
<td>Indicates that an RSA or ECC private key cannot be used in symmetric key distribution. This is the default. Note: Only a skeleton token created from PKA Key Token Build with this key usage type can be passed to PKA Key Generate to create a RETAIN (retained) key.</td>
</tr>
<tr>
<td>Translate control (One, optional)</td>
<td></td>
</tr>
</tbody>
</table>
Table 119. Keywords for PKA Key Token Build control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO-XLATE</td>
<td>The RSA or ECC key cannot be used as a key-encrypting-key for “PKA Key Translate (CSNDPKT)” on page 500.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use of this keyword does not matter when creating a skeleton key token for a later retained key generation operation. It is redundant to the necessary SIG-ONLY keyword.</td>
</tr>
<tr>
<td>XLATE-OK</td>
<td>The RSA or ECC key can be used as a key-encrypting-key for “PKA Key Translate (CSNDPKT)” on page 500.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Key tokens created with this keyword cannot be passed to the PKA Key Generate verb for creating RETAIN (retained) keys.</td>
</tr>
</tbody>
</table>

**key_value_structure_length**

**Direction:** Input  
**Type:** Integer

This is a segment of contiguous storage containing a variable number of input clear key values. The length depends on the key type parameter in the rule_array and on the actual values input. The length is in bytes. For maximum values, see Table 120.

Table 120. PKA Key Token Build - Key value structure length maximum values

<table>
<thead>
<tr>
<th>Key type</th>
<th>Key value structure maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECC-PAIR</td>
<td>207</td>
</tr>
<tr>
<td>ECC-PUBL</td>
<td>139</td>
</tr>
<tr>
<td>RSA-CRT, RSAMEVAR</td>
<td>3500</td>
</tr>
<tr>
<td>RSA-PRIV</td>
<td>648</td>
</tr>
<tr>
<td>RSA-PUBL</td>
<td>520</td>
</tr>
</tbody>
</table>

**key_value_structure**

**Direction:** Input  
**Type:** String

This is a segment of contiguous storage containing a variable number of input clear key values and the lengths of these values in bits or bytes, as specified. The structure elements are ordered, of variable length, and the input key values must be right-aligned within their respective structure elements and padded on the left with binary zeros. If the leading bits of the modulus are zeros, do not count them in the length. Table 121 defines the structure and contents as a function of key type.

Table 121. PKA Key Token Build - Key value structure elements

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Key value structure (ECC-PAIR)</strong></td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>Curve type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’00’ Prime curve</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’01’ Brainpool curve</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Reserved X’00’</td>
</tr>
</tbody>
</table>
Table 121. PKA Key Token Build - Key value structure elements (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>002</td>
<td>002</td>
<td>Length of $p$ in bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'00A0'$ Brainpool p-160</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'00C0'$ Prime P-192, Brainpool P-192</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'00E0'$ Prime P-224, Brainpool P-224</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0100'$ Prime P-256, Brainpool P-256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0140'$ Brainpool P-320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0180'$ Prime P-384, Brainpool P-384</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0200'$ Brainpool P-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0209'$ Prime P-521</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>$ddd$ - this field is the length of the private key $d$ in bytes. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. The maximum value is 66 bytes.</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>$xxx$ - this field is the length of the public key $Q$ in bytes. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. The maximum value is 133 bytes, which includes one byte to indicate if the value is compressed.</td>
</tr>
<tr>
<td>008</td>
<td>$ddd$</td>
<td>Private key, $d$</td>
</tr>
<tr>
<td>008 + $ddd$</td>
<td>$xxx$</td>
<td>Public key, $Q$</td>
</tr>
</tbody>
</table>

Key value structure (ECC-PUBL)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Curve type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'00'$ Prime curve</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'01'$ Brainpool curve</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Reserved $X'00'$</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of $p$ in bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'00A0'$ Brainpool p-160</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'00C0'$ Prime P-192, Brainpool P-192</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'00E0'$ Prime P-224, Brainpool P-224</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0100'$ Prime P-256, Brainpool P-256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0140'$ Brainpool P-320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0180'$ Prime P-384, Brainpool P-384</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0200'$ Brainpool P-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$X'0209'$ Prime P-521</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>$xxx$ - this field is the length of the public key $Q$ in bytes. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. The maximum value is 133 bytes, which includes one byte to indicate if the value is compressed.</td>
</tr>
<tr>
<td>006</td>
<td>$xxx$</td>
<td>Public key, $Q$</td>
</tr>
</tbody>
</table>

Key value structure (Optimized RSA, Chinese Remainder Theorem format, RSA-CRT)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Modulus length in bits (512 - 2048). This is required.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Modulus field length in bytes, $nnn$. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. This value must not exceed 256.</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Public exponent field length in bytes, $eee$. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb.</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>008</td>
<td>002</td>
<td>Length of the prime number, $p$, in bytes, $ppp$. This value can be zero if the key token is used as a skeleton key token in the PKA Key Generate verb. Maximum size of $p + q$ is 256 bytes.</td>
</tr>
</tbody>
</table>
### Table 121. PKA Key Token Build - Key value structure elements (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>002</td>
<td>Length of the prime number, ( q ), in bytes, ( qqq ). This value can be zero if the key token is used as a \textit{skeleton_key_token} in the PKA Key Generate verb. Maximum size of ( p + q ) is 256 bytes.</td>
</tr>
<tr>
<td>012</td>
<td>002</td>
<td>Length of ( d_p ), in bytes, ( rrr ). This value can be zero if the key token is used as a \textit{skeleton_key_token} in the PKA Key Generate verb. Maximum size of ( d_p + d_q ) is 256 bytes.</td>
</tr>
<tr>
<td>014</td>
<td>002</td>
<td>Length of ( d_q ), in bytes, ( sss ). This value can be zero if the key token is used as a \textit{skeleton_key_token} in the PKA Key Generate verb. Maximum size of ( d_p + d_q ) is 256 bytes.</td>
</tr>
<tr>
<td>016</td>
<td>002</td>
<td>Length of ( U ), in bytes, ( uuu ). This value can be zero if the key token is used as a \textit{skeleton_key_token} in the PKA Key Generate verb. Maximum size of ( U ) is 256 bytes.</td>
</tr>
<tr>
<td>018</td>
<td>nnn</td>
<td>Modulus, ( n ).</td>
</tr>
<tr>
<td>018 + nnn</td>
<td>eee</td>
<td>Public exponent, ( e ). This is an integer such that ( 1 &lt; e &lt; n ). ( e ) must be odd. When you are building a \textit{skeleton_key_token} to control the generation of an RSA key pair, the public key exponent can be one of the following values: 3, 65537 (( 2^{16} + 1 )), or 0 to indicate that a full random exponent should be generated. The exponent field can be a null-length field if the exponent value is 0.</td>
</tr>
<tr>
<td>018 + nnn + eee</td>
<td>ppp</td>
<td>Prime number, ( p ).</td>
</tr>
<tr>
<td>018 + nnn + eee + ppp</td>
<td>qqq</td>
<td>Prime number, ( q ).</td>
</tr>
<tr>
<td>018 + nnn + eee + ppp + qqq</td>
<td>rrr</td>
<td>( d_p = d \mod(p-1) ).</td>
</tr>
<tr>
<td>018 + nnn + eee + ppp + qqq + rrr</td>
<td>sss</td>
<td>( d_q = d \mod(q-1) ).</td>
</tr>
<tr>
<td>018 + nnn + eee + ppp + qqq + rrr + sss</td>
<td>uuu</td>
<td>( U = q^{-1} \mod(p) ).</td>
</tr>
</tbody>
</table>

**Key value structure** (RSA private, RSA private variable, or RSA public)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Modulus length in bits. This is required. When building a skeleton token, the modulus length in bits must be greater than or equal to 512 bits.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Modulus field length in bytes, ( XXX ). This value can be zero if you are using the key token as a skeleton in the PKA Key Generate verb. This value must not exceed 256 when the \textit{RSA-PUBL} keyword is used and must not exceed 128 when the \textit{RSA-PRIV} keyword is used. This verb can build a key token for a public RSA key with a 2048-bit modulus length or it can build a key token for a 1024-bit modulus length private key.</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Public exponent field length in bytes, ( YYY ). This value must not exceed 256 when the \textit{RSA-PUBL} keyword is used and must not exceed 128 when the \textit{RSA-PRIV} keyword is used. This value can be zero if you are using the key token as a skeleton token in the PKA Key Generate verb. In this case, a random exponent is generated. To obtain a fixed, predetermined public key exponent, you can supply this field and the public exponent as input to the PKA Key Generate verb.</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>Private exponent field length in bytes, ( ZZZ ). This field can be zero, indicating that private key information is not provided. This value must not exceed 128 bytes. This value can be zero if you are using the key token as a skeleton token in the PKA Key Generate verb.</td>
</tr>
</tbody>
</table>
### PKA Key Token Build (CSNDPKB)

Table 121. PKA Key Token Build - Key value structure elements (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>008</td>
<td>XXX</td>
<td>Modulus, ( n ). This is an integer such that ( 1 &lt; n &lt; 2^{2048} ). The ( n ) is the product of ( p ) and ( q ) for primes ( p ) and ( q ).</td>
</tr>
<tr>
<td>008 + XXX</td>
<td>YYY</td>
<td>RSA public exponent, ( e ). This is an integer such that ( 1 &lt; e &lt; n ). ( e ) must be odd. When you are building a skeleton_key_token to control the generation of an RSA key pair, the public key exponent can be one of the following values: 3, 65537 ( (2^{16} + 1) ), or 0 to indicate that a full random exponent should be generated. The exponent field can be a null-length field if the exponent value is 0.</td>
</tr>
<tr>
<td>008 + XXX + YYY</td>
<td>ZZZ</td>
<td>RSA secret exponent ( d ). This is an integer such that ( 1 &lt; d &lt; n ). The value of ( d ) is ( e^{-1} \mod(p-1)(q-1) ). You need not specify this value if you specify RSA-PUBL in the rule_array parameter.</td>
</tr>
</tbody>
</table>

Note:
1. All length fields are in binary.
2. All binary fields (exponent, lengths, modulus, and so on) are stored with the high-order byte field first. This integer number is right-aligned within the key structure element field.
3. You must supply all values in the structure to create a token containing an RSA or ECC private key for input to the PKA Key Import verb.

**private_key_name_length**
- Direction: Input
- Type: Integer
- The length can be 0 or 64.

**private_key_name**
- Direction: Input
- Type: EBCDIC character
- This field contains the name of a private key. The name must conform to CCA key label syntax rules. That is, allowed characters are alphanumeric, national (@, #, $) or period (.). The first character must be alphabetic or national. The name is folded to uppercase and converted to ASCII characters. ASCII is the permanent form of the name because the name should be independent of the platform. The name is then cryptographically coupled with clear private key data before encryption of the private key. Because of this coupling, the name can never change after the key token is imported. The parameter is valid only with key type RSA-CRT.

**user_definable_associated_data_length**
- Direction: Input
- Type: Integer
- Length in bytes of the user_definable_associated_data parameter. This parameter is valid only for a key type of ECC-PAIR, and must be set to 0 for all other key types. The maximum value is 100.

**user_definable_associated_data**
- Direction: Input
- Type: String
- The user_definable_associated_data parameter identifies a string variable containing the associated data that will be placed following the IBM associated...
PKA Key Token Build (CSNDPKB)

data in the token. The associated data is data whose integrity, but not whose confidentiality, is protected by a key wrap mechanism. The user_definable_associated_data can be used to bind usage control information.

This parameter is valid only for a key type of ECC-PAIR.

reserved_2_length
  Direction: Input
  Type: Integer
  Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_2
  Direction: Input
  Type: String
  The reserved_2 parameter identifies a string that is reserved. The verb ignores it.

reserved_3_length
  Direction: Input
  Type: Integer
  Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_3
  Direction: Input
  Type: String
  The reserved_3 parameter identifies a string that is reserved. The verb ignores it.

reserved_4_length
  Direction: Input
  Type: Integer
  Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_4
  Direction: Input
  Type: String
  The reserved_4 parameter identifies a string that is reserved. The verb ignores it.

reserved_5_length
  Direction: Input
  Type: Integer
  Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_5
  Direction: Input
  Type: String
  The reserved_5 parameter identifies a string that is reserved. The verb ignores it.

key_token_length
**PKA Key Token Build (CSNDPKB)**

**Direction: Input/Output**
Type: Integer

Length of the returned key token. The verb checks the field to ensure that it is at least equal to the size of the token to return. On return from this verb, this field is updated with the exact length of the key_token created. On input, a size of 3500 bytes is sufficient to contain the largest key_token created.

**key_token**
**Direction: Output**
Type: String

The returned key token containing an unenciphered private or public key. The private key is in an external form that can be exchanged with different CCA PKA systems. You can use the public key token directly in appropriate CCA signature verification or key management services.

**Restrictions**
The restrictions for CSNDPKB.

None

**Required commands**
The required commands for CSNDPKB.

None

**Usage notes**
The usage notes for CSNDPKB.

None

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNDPKBJ.

See ["Building Java applications to use with the CCA JNI"](chapter12.html) on page 26.

The parameters for CSNDPKBJ are shown here.

**Format**
```java
public native void CSNDPKBJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_values_structure_length,
    byte[] key_values_structure,
    hikmNativeInteger key_name_length,
    byte[] key_name,
    hikmNativeInteger user_definable_associated_data_length,
    byte[] user_definable_associated_data,
    hikmNativeInteger reserved_2_length,
    byte[] reserved_2,
    hikmNativeInteger reserved_3_length,
    byte[] reserved_3,
    hikmNativeInteger reserved_4_length,
);```
PKA Key Token Build (CSNDPKB)

byte[] reserved_4,
hikmNativeInteger reserved_5_length,
byte[] reserved_5,
hikmNativeInteger token_length,
byte[] token);}
The PKA Key Token Change verb changes PKA key tokens (RSA or ECC) or trusted block key tokens, from encipherment under old ASYM-MK or APKA-MK, to encipherment under the current ASYM-MK or APKA-MK master key.

### IMPORTANT

Two problems have been discovered with the CCA microcode related to the reenciphering of master keys. Although similar, the two problems are slightly different and exist in different levels of the microcode. These problems could lead to a loss of operational private keys after a master key change. Symmetric keys are not affected. Although it is expected few customers will be impacted this document describes the problems and how to recover.


The PKA Key Token Change (CSNDKTC) verb has been changed to not permit the use of the RTNMK keyword for processor firmware levels that have this problem.

**For RSA key tokens -** Key tokens must be private internal PKA key tokens in order to be changed by this verb. PKA private keys encrypted under the Key Management Master Key (KMMK) cannot be reenciphered using this service unless the KMMK has the same value as the Signature Master Key (SMK).

**For trusted block key tokens -** Trusted block key tokens must be internal.

**For ECC key tokens -** Key tokens must be private internal ECC key tokens encrypted under the APKA-MK.

### Format

The format of CSNDKTC.

```c
CSNDKTC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier_length,
    key_identifier)
```

### Parameters

The parameters for CSNDKTC.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

- **Direction:** Input
- **Type:** Integer
  
  A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1 or 2.

**rule_array**

- **Direction:** Input
- **Type:** String
PKA Key Token Change (CSNDKTC)

The process rule for the verb. The keyword must be in eight bytes of contiguous storage, left-aligned, and padded on the right with blanks. The \textit{rule_array} keywords are described in Table 122.

Table 122. Keywords for PKA Key Token Change control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Token type} (One, optional)</td>
<td></td>
</tr>
<tr>
<td>ECC</td>
<td>Specifies that the key being changed is an ECC key. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>RSA</td>
<td>Specifies that the key being changed is an RSA key or a trusted block. This is the default. This keyword was introduced with CCA 4.1.0.</td>
</tr>
<tr>
<td>\textit{Reencipherment method} (One, required)</td>
<td></td>
</tr>
<tr>
<td>RTCMK</td>
<td>If the \textit{key_identifier} is an RSA key token, the verb will change an RSA private key from encipherment with the old ASYM-MK to encipherment with the current ASYM-MK. If the \textit{key_identifier} is a trusted block token, the verb will change the trusted block's embedded MAC key from encipherment with the old ASYM-MK to encipherment with the current ASYM-MK. If the \textit{key_identifier} is an ECC key token, the verb will change an ECC private key from encipherment with the old APKA-MK to encipherment with the current APKA-MK.</td>
</tr>
<tr>
<td>RTN MK</td>
<td>Re-enciphers a private (internal) RSA or ECC key to the new master key.</td>
</tr>
</tbody>
</table>

A key enciphered under the new master key is not usable. It is expected that the user will use this keyword (RTN MK) to take a preparatory step in re-enciphering an external key store that they manage themselves to a new master-key, before the set operation has occurred. Note also that the new master-key register must be full; it must have had the last key part loaded and therefore not be empty or partially full (partially full means that one or more key parts have been loaded but not the last key part).

The 'SET' operation makes the new master-key operational, moving it to the current master-key register, and the current master-key is displaced into the old master-key register. When this happens, all the keys that were re-enciphered to the new master-key are now usable, because the new master-key is not 'new' any more, it is 'current'.

Because the RTN MK keyword is added primarily for support of externally managed key storage (see "Key Storage on z/OS (RTN MK-focused)" on page 349), it is not valid to pass a \textit{key_identifier} when the RTN MK keyword is used. Only a full internal key token (encrypted under the current master-key) can be passed for re-encipherment with the RTN MK keyword. When a key LABEL is passed along with the RTN MK keyword, the error return code 8 with reason code 63 will be returned.

For more information, see "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 348.

\texttt{key_identifier_length}

\textbf{Direction: Input}
\textbf{Type: Integer}

The length of the \textit{key_identifier} parameter. The maximum size is 3500 bytes.

\texttt{key_identifier}

\textbf{Direction: Input/Output}
\textbf{Type: String}

Contains an internal key token of an internal RSA or ECC key, or trusted block key. If the key token is an RSA key token, the private key within the token is securely reenciphered under the current ASYM-MK. If the key token is a trusted block key token, the MAC key within the token is securely...
re-enciphered under the current ASYM-MK. If the key token is an ECC key
token, the private key within the token is securely re-enciphered under the
current APKA-MK.

Restrictions
The restrictions for CSNDKTC.

None

Required commands
The required commands for CSNDKTC.

This verb requires the PKA Key Token Change RTCMK command (offset X'0102')
to be enabled in the active role.

Usage notes
The usage notes for CSNDKTC.

None

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDKTCJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDKTCJ are shown here.

Format

```java
public native void CSNDKTCJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger key_label_length,
    byte[] key_label);
```
PKA Key Translate (CSNDPKT)

The PKA Key Translate verb translates PKA key tokens from encipherment under the old Asymmetric-Keys Master Key to encipherment under the current Asymmetric-Keys Master Key.

This verb changes only Private Internal PKA Key Tokens.

The source CCA RSA key token must be wrapped with a transport key encrypting key (KEK). The XLate bit must also be turned on in the key usage byte of the source token. The source token is unwrapped using the specified source transport KEK. The target key token will be wrapped with the specified target transport KEK. Existing information in the target token is overwritten.

There are restrictions on which type key can be used for the source and target transport key tokens. These restrictions are enforced by access control points.

There are restrictions on which rule can be used. These restrictions are enforced by access control points.

Format

The format of CSNDPKT.

```
CSNDPKT(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  rule_array_count,
  rule_array,
  source_key_identifier_length,
  source_key_identifier,
  source_transport_key_identifier_length,
  source_transport_key_identifier,
  target_transport_key_identifier_length,
  target_transport_key_identifier,
  target_key_token_length,
  target_key_token)
```

Parameters

The parameters for CSNDPKT.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1.

`rule_array`

Direction: Input
Type: String
PKA Key Translate (CSNDPKT)

The process rule for the verb. The keyword must be in eight bytes of contiguous storage, left-aligned, and padded on the right with blanks. The rule_array keywords are described in Table 123.

Table 123. Keywords for PKA Key Translate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smartcard Format (One, required)</td>
<td>This keyword indicates translating the key into the smart card Visa proprietary format.</td>
</tr>
<tr>
<td>SCVISA</td>
<td>This keyword indicates translating the key into the smart card Modulus-Exponent format.</td>
</tr>
<tr>
<td>SCCOMME</td>
<td>This keyword indicates translating the key into the smart card Chinese Remainder Theorem format.</td>
</tr>
</tbody>
</table>

source_key_identifier_length

  Direction: Input  
  Type: Integer

  The length of the source_key_identifier parameter. The maximum size is 3500 bytes.

source_key_identifier

  Direction: Input  
  Type: String

  This field contains either a key label identifying an RSA private key, or an external public-private key token. The private key must be wrapped with a key encrypting key.

source_transport_key_identifier_length

  Direction: Input  
  Type: Integer

  Length in bytes of the source_transport_key_identifier parameter. This value must be 64.

source_transport_key_identifier

  Direction: Input/Output  
  Type: String

  This field contains an internal token or label of a DES key-encrypting key. This key is used to unwrap the input RSA key token specified with parameter source_key_identifier. See “Usage notes” on page 503 for details on the type of transport key that can be used.

target_transport_key_identifier_length

  Direction: Input  
  Type: Integer

  Length in bytes of the target_transport_key_identifier parameter. This value must be 64.

target_transport_key_identifier

  Direction: Input/Output  
  Type: String

  This field contains an internal token or label of a DES key-encrypting key. This key is used to wrap the output RSA key returned with the target_key_token parameter. See “Usage notes” on page 503 for details on the type of transport key that can be used.
PKA Key Translate (CSNDPKT)

**target_key_token_length**
- Direction: Input
- Type: Integer

Length in bytes of the target_key_token parameter. On output, the value in this variable is updated to contain the actual length of the target_key_token produced by the verb. The maximum length is 3500 bytes.

**target_key_token**
- Direction: Output
- Type: String

This field contains the RSA key in the smartcard format specified in the rule_array parameter, and is protected by the key-encrypting key specified in the target_transport_key parameter. This is not a CCA token, and cannot be stored in the key storage.

**Restrictions**
The restrictions for CSNDPKT.

CCA RSA Modulus-Exponent tokens will not be translated to the SCCOMCRT format. CCA RSA Chinese Remainder Theorem tokens will not be translated to the SCCOMME format. SCVISA supports only Modulus-Exponent (ME) keys.

**Required commands**
The required commands for CSNDPKT.

This verb requires the following commands to be enabled in the active role based on the keyword:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCVISA</td>
<td>X'0318'</td>
<td>PKA Key Translate - from CCA RSA to SC Visa Format</td>
</tr>
<tr>
<td>SCCOMME</td>
<td>X'0319'</td>
<td>PKA Key Translate - from CCA RSA to SC ME Format</td>
</tr>
<tr>
<td>SCCOMCRT</td>
<td>X'031A'</td>
<td>PKA Key Translate - from CCA RSA to SC CRT Format</td>
</tr>
</tbody>
</table>

These commands must also be enabled to allow the key type combinations shown in this table:

<table>
<thead>
<tr>
<th>Source transport key type</th>
<th>Target transport key type</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPORTER</td>
<td>EXPORTER</td>
<td>X'031B'</td>
<td>PKA Key Translate - from source EXP KEK to target EXP KEK</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>EXPORTER</td>
<td>X'031C'</td>
<td>PKA Key Translate - from source IMP KEK to target EXP KEK</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>IMPORTER</td>
<td>X'031D'</td>
<td>PKA Key Translate - from source IMP KEK to target IMP KEK</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>IMPORTER</td>
<td>N/A</td>
<td>This key type combination is not allowed.</td>
</tr>
</tbody>
</table>
Usage notes

The usage notes for CSNDPKT.

None.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNDPKTJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNDPKTJ are shown here.

Format

```java
public native void CSNDPKTJ (
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger source_key_identifier_length,
    byte[] source_key_identifier,
    hikmNativeInteger source_transport_key_identifier_length,
    byte[] source_transport_key_identifier,
    hikmNativeInteger target_transport_key_identifier_length,
    byte[] target_transport_key_identifier,
    hikmNativeInteger target_key_token_length,
    byte[] target_key_token);
```
PKA Public Key Extract (CSNDPKX)

Use the PKA Public Key Extract verb to extract a PKA public key token from a supplied PKA internal or external private key token.

This verb performs no cryptographic verification of the PKA private token. You can verify the private token by using it in a verb such as Digital Signature Generate.

Format

The format of CSNDPKX.

```
CSNDPKX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    source_key_identifier_length,
    source_key_identifier,
    target_public_key_token_length,
    target_public_key_token)
```

Parameters

The parameters for CSNDPKX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

`rule_array_count`

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0.

`rule_array`

Direction: Input

Type: String

This parameter is ignored.

`source_key_identifier_length`

Direction: Input

Type: Integer

The length of the `source_key_identifier` parameter. The maximum size is 3500 bytes. When the `source_key_identifier` parameter is a key label, this field specifies the length of the label.

`source_key_identifier`

Direction: Input/Output

Type: String

The internal or external token of a PKA private key or the label of a PKA private key. This can be the input or output from the PKA Key Import or PKA Key Generate verbs. This verb supports:
PKA Public Key Extract (CSNDPKX)

- RSA private key token formats supported on the CEX2C or CEX*C. If the source_key_identifier specifies a label for a private key that has been retained within a CEX2C, this verb extracts only the public key section of the token.
- ECC private key token formats supported starting with CEX3C.

**target_public_key_token_length**
Direction: Input/Output
Type: Integer
The length of the target_public_key_token parameter. The maximum size is 2500 bytes. On output, this field will be updated with the actual byte length of the target_public_key_token.

**target_public_key_token**
Direction: Output
Type: String
This field contains the token of the extracted PKA public key.

**Restrictions**
The restrictions for CSNDPKX.

None

**Required commands**
The required commands for CSNDPKX.

None

**Usage notes**
The usage notes for CSNDPKX.

This verb extracts the public key from the internal or external form of a private key. However, it does not check the cryptographic validity of the private token.

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNDPKXJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDPKXJ are shown here.

**Format**
```java
public native void CSNDPKXJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger source_key_identifier_length,
    byte[] source_key_identifier,
    hikmNativeInteger target_key_token_length,
    byte[] target_key_token);
```
Remote Key Export (CSNDRKX)

This verb is used as a method of secured transport of DES keys using asymmetric techniques from a security module (for example, the CEX3C) to a remote device such as an Automated Teller Machine (ATM).

The DES keys to be transported are either key encrypting keys that are generated within the coprocessor or, alternately, operational keys or replacement KEKs enciphered under a KEK currently installed in a remote device.

Generating and exporting DES keys

This verb uses a trusted block to generate or export DES keys.

To create a trusted block, see "Trusted Block Create (CSNDTBC)" on page 517. Remote Key Export accepts as input parameters a trusted block, a public-key certificate and certificate parameters, a transport key, a rule ID to identify the appropriate rule section to be used within a trusted block, an importer key, a source key, optional extra data that can be used as part of the OAEP key-wrapping process, and key-check parameters used to calculate an optional key-check value.

This verb validates all input parameters for generate and export operations. After the verb performs the input parameter validation, the remaining steps depend on whether the generate option or the export option is specified in the selected rule of the trusted block.

This is a high-level description of the remaining processing steps for generate and export.

Processing for generate operation: The verb performs these steps for the generate operation:

1. Generates a random value for the generated key, K. The generated key length specified by the selected rule determines the key length.
2. XORs the output key variant with the randomly generated key K from the previous step, if the selected rule contains a common export key parameters subsection and the output key variant length is greater than zero. Adjusts the result to have valid DES key parity.
3. Continues with Final processing steps common to generate and export operations.

Processing steps for export operation: The verb performs these steps for the export operation:

1. If the selected rule contains a transport key rule reference subsection, verifies that the rule ID in the transport key rule reference subsection matches the rule ID in the token identified by the transport_key_identifier parameter, provided that the token is an RKX key-token. For more information on RKX key tokens, see "External RKX DES key tokens" on page 613.
2. Verifies that the length of the transport key variant in the transport key variant subsection of the selected rule is greater than or equal to the length of the key identified by the transport_key_identifier parameter.
3. Verifies that the key token identified by the importer_key_identifier parameter is of key type IMPORTER, if the source_key_identifier parameter identifies an external CCA DES key-token.
4. Recovers the clear value of the source key, K, identified by the source_key_identifier parameter.
Remote Key Export (CSNDRKX)

5. Verifies that the length of key K is between the export key minimum length and export key maximum length specified in the common export key parameters subsection of the selected rule.

6. XORs the output key variant with the randomly generated key K from the previous step, if the selected rule contains a common export key parameters subsection and the output key variant length is greater than zero. Adjusts the result to have valid DES key parity.

7. Uses the public key in the trusted block to verify the digital signature embedded in the certificate variable if the certificate_length variable is greater than zero. Any necessary certificate objects are located with information from the certificate_parms variable. Returns an error if the signature verification fails.

8. XORs the transport key variant with the clear value of the transport key (recovered in the previous step) if the selected rule contains a transport key variant subsection and the output key variant length is greater than zero. Adjusts the result to have valid DES key parity.

9. Continues with Final processing steps common to generate and export operations:

Final processing steps common to generate and export operations:

1. Based on the symmetric encrypted output key format flag of the selected rule, returns the encrypted result in the token identified by the sym_encrypted_key_identifier parameter.
   - Of Processing for generate operation Step 2 on page 506 or Step 6 into an RKX key-token, if the flag indicates to return an RKX key-token.
   - Using the resulting key from Processing for generate operation Step 6 into a CCA DES key-token and returns it in the token identified by the sym_encrypted_key_identifier parameter, if the flag indicates to return a CCA DES key-token.

2. Encrypts the key result from Processing for generate operation Step 2 on page 506 or Step 6 with the format specified, if the asymmetric encrypted output key format flag of the selected rule indicates to output an asymmetric encrypted key.

3. Returns the computed key-check value as determined by the key-check algorithm identifier if the key-check algorithm identifier in the specified rule indicates to compute a key-check value. The value is returned in the key_check_value variable.
Remote Key Export (CSNDRKX)

Format

The format of CSNDRKX.

```c
CSNDRKX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    trusted_block_identifier_length,
    trusted_block_identifier,
    certificate_length,
    certificate,
    certificate_parms_length,
    certificate_parms,
    transport_key_identifier_length,
    transport_key_identifier,
    rule_id_length,
    rule_id,
    importer_key_identifier_length,
    importer_key_identifier,
    source_key_identifier_length,
    source_key_identifier,
    asym_encrypted_key_length,
    asym_encrypted_key,
    sym_encrypted_key_identifier_length,
    sym_encrypted_key_identifier,
    extra_data_length,
    extra_data,
    key_check_parameters_length,
    key_check_parameters,
    key_check_value_length,
    key_check_value
)
```

Parameters

The parameters for CSNDRKX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see “Parameters common to all verbs” on page 20.

**rule_array_count**

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 0.

**rule_array**

Direction: Input

Type: String

This parameter is ignored.

**trusted_block_identifier_length**

Direction: Input

Type: Integer

A pointer to an integer variable containing the number of bytes of data in the `trusted_block_identifier` variable. The maximum length is 3500 bytes.
trusted_block_identifier
  
  Direction: Input
  Type: String

  A pointer to a string variable containing a trusted block key-token of an internal trusted block, or the key label of a trusted block key-token record of an internal trusted block. It is used to validate the public-key certificate and to define the rules for key generation and key export.

certificate_length
  
  Direction: Input
  Type: Integer

  A pointer to an integer variable containing the number of bytes of data in the certificate variable. The maximum length is 5000 bytes.

  It is an error if the certificate_length variable is 0 and the trusted block’s asymmetric encrypted output key format in the rule section selected by the rule_id variable indicates PKCS-1.2 output format or RSA-OAEP output format.

  If the certificate_length variable is 0 or the trusted block’s asymmetric encrypted output key format in the rule section selected by the rule_id variable indicates no asymmetric key output, the certificate is ignored.

certificate
  
  Direction: Input
  Type: String

  A pointer to a string variable containing a public-key certificate. The certificate must contain the public-key modulus and exponent in binary form, as well as a digital certificate. The certificate must verify using the root public key that is in the trusted block pointed to by the trusted_block_identifier parameter.

  Note: After the hash is computed over the certificate data specified by offsets 28 and 32, the hash is BER encoded by pre-pending these bytes:

  X'30213009 06052B0E 03021A05 000 414'

  See “PKCS #1 formats” on page 720.

certificate_parms_length
  
  Direction: Input
  Type: Integer

  A pointer to an integer variable containing the number of bytes of data in the certificate_parms variable. The length must be 36 bytes if the certificate_length variable is 0, else the length must be 0.

certificate_parms
  
  Direction: Input
  Type: String

  A pointer to a string variable containing a structure for identifying the location and length of values within the public-key certificate pointed to by the certificate parameter. If the value of the certificate_length variable is 0, then the information in this variable is ignored but the variable must be declared. The format of the certificate_parms variable is defined in Table 124 on page 510.
### Keywords for Remote Key Export certificate_parms parameter

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Offset of modulus</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Length of modulus</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Offset of public exponent</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>Length of public exponent</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Offset of digital signature</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>Length of digital signature</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>Identifier for hash algorithm. The following values are defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hash algorithm</td>
</tr>
<tr>
<td>X'01'</td>
<td>SHA-1</td>
<td></td>
</tr>
<tr>
<td>X'02'</td>
<td>MD5 (Currently not supported)</td>
<td></td>
</tr>
<tr>
<td>X'03'</td>
<td>RIPEMD-160 (Currently not supported)</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>Identifier for digital signature hash formatting method used. The following values are defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hash formatting method</td>
</tr>
<tr>
<td>X'01'</td>
<td>PKCS-1.0</td>
<td></td>
</tr>
<tr>
<td>X'02'</td>
<td>PKCS-1.1</td>
<td></td>
</tr>
<tr>
<td>X'03'</td>
<td>X9.31 (Currently not supported)</td>
<td></td>
</tr>
<tr>
<td>X'04'</td>
<td>ISO-9796 (Currently not supported)</td>
<td></td>
</tr>
<tr>
<td>X'05'</td>
<td>ZERO-PAD (Currently not supported)</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>Reserved, must be binary zeros</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>Offset of first byte of certificate data hashed to compute the digital signature</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>Length of certificate data hashed to compute the digital signature</td>
</tr>
</tbody>
</table>

**Note:** The modulus, exponent, and signature values can have bit lengths that are not multiples of 8; each of these values is right-aligned and padded on the left with binary zeroes to make it an even number of bytes in length.

**transport_key_identifier_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the transport_key_identifier variable. The length must be 0 or 64 bytes.

**transport_key_identifier**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing a KEK key-token, or a key label of a KEK key-token record. The KEK is either an internal CCA DES key-token (key type IMPORTER or EXPORTER), or an external version X'10' (RKX) DES key-token. It is used to encrypt a key exported by the verb.

When the symmetric encrypted output key format flag of the selected rule indicates return an RKX key-token, this parameter is ignored but must be declared.

If this parameter points to a CCA DES key-token:
Remote Key Export (CSNDRKX)

- The token must be of key type IMPORTER or EXPORTER.
- If the source_key_identifier parameter identifies an internal CCA DES key-token, the token must be of key type EXPORTER.

For more information on RXX key tokens, see “External RXX DES key tokens” on page 613.

rule_id_length
Direction: Input
Type: Integer
A pointer to an integer variable containing the number of bytes of data in the rule_id variable. The length must be eight bytes.

rule_id
Direction: Input
Type: String
A pointer to a string variable that identifies the rule in the trusted block to be used to control key generation or export. The trusted block can contain multiple rules, each of which is identified by a unique rule ID value.

importer_key_identifier_length
Direction: Input
Type: Integer
A pointer to an integer variable containing the number of bytes of data in the importer_key_identifier variable. The length must be 0 or 64 bytes.

importer_key_identifier
Direction: Input
Type: String
A pointer to a string variable containing an IMPORTER KEK key-token or a label of an IMPORTER KEK key-token record. This KEK is used to decipher the key pointed to by source_key_identifier parameter.

This variable is ignored if the verb is used to generate a new key, or the source_key_identifier variable contains either an RXX key token or an internal CCA DES key-token. For more information on RXX key tokens, see “External RXX DES key tokens” on page 613.

source_key_identifier_length
Direction: Input
Type: Integer
A pointer to an integer variable containing the number of bytes of data in the source_key_identifier variable. The length must be 0 or 64 bytes.

source_key_identifier
Direction: Input
Type: String
A pointer to a string variable containing a DES key-token or a label of a DES key-token record. The key token contains the key to be exported, and must meet one of these criteria:
- It is a single-length or double-length external CCA DES key-token.
- It is a single-length or double-length internal CCA DES key-token.
- It is a single-length, double-length, or triple-length RXX key-token.
Remote Key Export (CSNDRKX)

Note:
1. If the key token is a CCA DES key-token, its XPORT-OK control vector bit (bit 17) must be B'1', or else the export will not be allowed.
2. If a DES key-token has three 8-byte key parts, the parts are considered unique if any two of the three key parts differ.

**asym_encrypted_key_length**

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the `asym_encrypted_key` variable. On output, the variable is updated with the actual length of the `asym_encrypted_key` variable. The input length must be at least the length of the modulus in bytes of the public-key in the certificate variable.

**asym_encrypted_key**

Direction: Output
Type: String

A pointer to a string variable containing a generated or exported clear key returned by the verb. The clear key is encrypted by the public (asymmetric) key provided by the certificate variable.

**sym_encrypted_key_identifier_length**

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the number of bytes of data in the `sym_encrypted_key_identifier` variable. On output, the variable is updated with the actual length of the `sym_encrypted_key_identifier` variable. The input length must be a minimum of 64 bytes.

**sym_encrypted_key_identifier**

Direction: Output
Type: String

A pointer to a string variable. On input, the `sym_encrypted_key_identifier` variable must contain either a key label of a CCA DES key-token record or an RKX key-token record, or be filled with binary zeros.

On output, the verb produces a CCA DES key-token or an RKX key-token, depending on the value of the symmetric encrypted output key format value of the rule section within the `trusted_block_identifier` variable. The key token produced contains either a generated or exported key encrypted using the key-encrypting key provided by the `transport_key_identifier` variable.

* If the output is an external CCA DES key-token:
  1. If a common export key parameters subsection (X'0003') is present in the selected rule, the control vector (CV) is copied from the subsection into the output CCA DES key-token. Otherwise, the CV is copied from source key-token.
  2. If a transport key variant subsection (X'0001') is present in the selected rule, the key is multiply enciphered under the transport key XORed with the transport key variant from the subsection. Otherwise, the key is multiply enciphered under the transport key XORed with binary zero.
  3. XORs the CV in the token with the encrypted result from the previous step.
  4. Stores the previous result in the token and updates the TVV.
Remote Key Export (CSNDRKX)

- If the output is an (external) RKX key-token:
  1. Encrypts the key using a variant of the trusted block MAC key.
  2. Builds the token with the encrypted key and the rule_id variable.
  3. Calculates the MAC of the token contents and stores the result in the token.

If the sym_encrypted_key_identifier variable is a key label on input, on output the key token produced by the verb is stored in DES key-storage and the variable remains the same. Otherwise, on output the variable is updated with the key token produced by the verb, provided the field is of sufficient length.

extra_data_length
  Direction: Input
    Type: Integer
    A pointer to an integer variable containing the number of bytes of data in the extra_data variable. The length must be less than or equal to the byte length of the certificate public key modulus minus the generated/exported key length minus 42 (X'2A'), which is the OAEP overhead. For example, if the public key in the certificate has a modulus length of 1024 bits (128 bytes), and the exported key is single length, then the extra data length must be less than or equal to 128 minus 8 minus 42, which equals 78.

extra_data
  Direction: Input
    Type: String
    A pointer to a string variable containing extra data to be used as part of the OAEP key-wrapping process. The extra_data variable is used when the output format for the RSA-encrypted key that is returned in the asym_encrypted_key variable is RSA-OAEP; otherwise, it is ignored.

  Note: The RSA-OAEP format is specified as part of the rule in the trusted block.

key_check_parameters_length
  Direction: Input
    Type: Integer
    A pointer to an integer variable containing the number of bytes of data in the key_check_parameters variable. The length must be 0.

key_check_parameters
  Direction: Input
    Type: String
    Reserved for future use.

key_check_value_length
  Direction: Input/Output
    Type: Integer
    A pointer to a string variable containing the number of bytes of data in the key_check_value variable. On output, and if the field is of sufficient length, the variable is updated with the actual length of the key_check_value variable.

key_check_value
Remote Key Export (CSNDRKX)

Direction: Output
Type: String

A pointer to a string variable containing the result of the key-check algorithm chosen in the rule section of the selected trusted block. See “Encrypt zeros DES-key verification algorithm” on page 700 and “Modification Detection Code calculation” on page 700. When the selected key-check algorithm is to encrypt an 8-byte block of binary zeros with the key, and the generated or exported key is:

- Single length
  1. A value of 0, 1, or 2 is considered insufficient space to hold the output encrypted result, and the verb returns an error.
  2. A value of 3 returns the leftmost three bytes of the encrypted result if the key_check_value_length variable is 3 or greater. Otherwise, an error is returned.
  3. A value of 4 - 8 returns the leftmost four bytes of the encrypted result if the key_check_value_length variable is 4 or greater. Otherwise, an error is returned.

- Double length or triple length
  The verb returns the entire 8-byte result of the encryption in the key_check_value variable if the key_check_value_length variable is 8 or more. Otherwise, an error is returned.

When the selected key-check algorithm is to compute the MDC-2 hash of the key, and the generated or exported key is single length, the 8-byte key is made into a double-length key by replicating the key halves. This is because the MDC-2 calculation method does no padding, and requires that the data be a minimum of 16 bytes and a multiple of eight bytes. If the generated or exported key is double length or triple length, the key is processed as is. The verb returns the 16-byte hash result of the key in the key_check_value variable if the key_check_value_length variable is large enough, else an error is returned.

Restrictions

The restrictions for CSNDRKX.
1. AES keys are not supported by this verb.
2. Keys with a modulus length greater than 2048 bits are not supported in releases before Release 3.30.
3. The maximum public exponent is 17 bits for any key that has a modulus greater than 2048 bits.

Required commands

The required commands for CSNDRKX.

This verb requires the Remote Key Export - Generate or export a key for use by a non-CCA node command (offset X'0312') to be enabled in the active role.

The verb also requires the Key Generate - SINGLE-R command (offset X'00DB') to be enabled to replicate a single-length source key (either from a CCA DES key-token or an RXX key-token). If authorized, key replication occurs if all of the following are true:
1. The key token returned using the sym_encrypted_key_identifier parameter is a CCA DES key-token, as defined in the rule section identified by the rule_id parameter.
2. The rule section identified by the `rule_id` parameter has a common export key parameters subsection defined, and the control vector in the subsection is 16 bytes in length with key-form bits of B’010’ for the left half and B’001’ for the right half.

3. The token identified by the `source_key_identifier` parameter is single length, and is either a CCA DES key-token or an RKX key-token.

To enable the use of key-encrypting-keys with the NOCV option for export, this verb requires the NOCV KEK usage for export-related functions command (offset X’0300’) to be enabled in the active role.

To enable the use of key-encrypting-keys with the NOCV option for import, this verb requires the NOCV KEK usage for import-related functions command (offset X’030A’) to be enabled in the active role.

Note: A role with X’00DB’ enabled can also use the Key Generate verb with the SINGLE-R key-length keyword.

Usage notes
The usage notes for CSNDRKX.

None

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDRKXJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDRKXJ are shown here.

Format
```java
public native void CSNDRKXJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger trusted_block_length,
    byte[] trusted_block_identifier,
    hikmNativeInteger certificate_length,
    byte[] certificate,
    hikmNativeInteger certificate_parms_length,
    byte[] certificate_parms,
    hikmNativeInteger transport_key_identifier_length,
    byte[] transport_key_identifier,
    hikmNativeInteger rule_id_length,
    byte[] rule_id,
    hikmNativeInteger export_key_kek_length,
    byte[] export_key_kek_identifier,
    hikmNativeInteger export_key_length,
    byte[] export_key_identifier,
    hikmNativeInteger asym_encrypted_key_length,
    byte[] asym_encrypted_key,
    hikmNativeInteger sym_encrypted_key_length,
    byte[] sym_encrypted_key,
    hikmNativeInteger extra_data_length,
    byte[] extra_data,
    hikmNativeInteger key_check_parameters_length,
)
```
Remote Key Export (CSNDRKX)

byte[] key_check_parameters,
hikmNativeInteger key_check_length,
byte[] key_check_value);
Trusted Block Create (CSNDTBC)

The verb creates an external trusted block under dual control. A trusted block is an extension of CCA PKA key tokens using new section identifiers.

Trusted blocks are an integral part of a remote key-loading process. They contain various items, some of which are optional, and some of which can be present in different forms. Tokens are composed of concatenated sections. For a detailed description of a trusted block, including its format and field values, see “Trusted blocks” on page 647.

Creating an external trusted block: Create an active external trusted block in two steps:
1. Create an inactive external trusted block using the INACTIVE rule_array keyword. This step requires the Trusted Block Create - Create a Trusted Key Block in Inactive form command (offset X'030F') to be enabled in the active role.
2. Complete the creation process by activating (promoting) an inactive external trusted block using the ACTIVE rule_array keyword. This step requires the Trusted Block Create - Activate an Inactive Trusted Key Block command (offset X'0310') to be enabled in the active role. Changing an external trusted block from inactive to active effectively approves the trusted block for further use.

Note: Authorize each command in a different role to enforce a dual-control policy.

Creating an inactive external trusted block: To create an inactive external trusted block, use a rule_array_count of 1 and a rule_array keyword of INACTIVE. Identify the input trusted block using the input_block_identifier parameter, and set the input_block_identifier_length variable to the length of the key label or the key token of the input block. The input block can be any one of these forms:
- An uninitialized trusted block. The trusted block is complete except that it does not have MAC protection.
- An inactive trusted block. The trusted block is external, and it is in inactive form. MAC protection is present due to recycling of an existing inactive trusted block.
- An active trusted block. The trusted block is internal or external, and it is in active form. MAC protection is present due to recycling of an existing active trusted block.

Note: The MAC key is replaced with a new MAC key, and any RKX key-token created with the input trusted block cannot be used with the output trusted block.

This verb randomly generates a confounder and triple-length MAC key, and uses a variant of the MAC key to calculate an ISO 16609 CBC mode TDES MAC of the trusted block contents. To protect the MAC key, the verb encrypts the confounder and MAC key using a variant of an IMP-PKA key. The calculated MAC and the encrypted confounder and MAC key are embedded in the output trusted block. Use the transport_key_identifier parameter to identify the key token that contains the IMP-PKA key.
**Trusted Block Create (CSNDTBC)**

On input, set the `trusted_block_identifier_length` variable to the length of the key label or at least the size of the output trusted block. The output trusted block is returned in the key-token identified by the `trusted_block_identifier` parameter, and the verb updates the `trusted_block_identifier_length` variable to the size of the key token if a key label is not specified.

*Creating an active external trusted block:* To create an active external trusted block, use a `rule_array_count` of 1 and a `rule_array` keyword of `ACTIVE`. Identify the input trusted block using the `input_block_identifier` parameter, and set the `input_block_identifier_length` variable to the length of the key label or the key token of the input block. The input block must be an inactive external trusted block that was created using the `INACTIVE` `rule_array` keyword.

Use the `transport_key_identifier` parameter to identify the key token that contains the IMP-PKA key.

On input, set the `trusted_block_identifier_length` variable to the length of the key label or at least the size of the output trusted block. The verb returns an error if the input trusted block is not valid. Otherwise, it changes the flag in the trusted block information section from the inactive state to the active state, recalculates the MAC, and embeds the updated MAC value in the output trusted block.

The output trusted block is returned in the key-token identified by the `trusted_block_identifier` parameter, and the verb updates the `trusted_block_identifier_length` variable to the size of the key token if a key label is not specified.

**Format**

The format of CSNDRKCX.

```c
CSNDTBC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    input_block_identifier_length,
    input_block_identifier,
    transport_key_identifier,
    trusted_block_identifier_length,
    trusted_block_identifier)
```

**Parameters**

The parameters for CSNDRKCX.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

*rule_array_count*

    Direction: Input
    Type: Integer

    A pointer to an integer variable containing the number of elements in the `rule_array` variable. This value must be 1.

*rule_array*
**Trusted Block Create (CSNDTBC)**

**Direction:** Input  
**Type:** Array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are described in Table 125.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong> (One required)</td>
<td></td>
</tr>
<tr>
<td>INACTIVE</td>
<td>Create an external trusted block, based on the <em>input_block_identifier</em> variable, and set the active flag to B'0'. This makes the trusted block unusable in any other CCA services.</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>Create an external trusted block, based on the token identified by the <em>input_block_identifier</em> parameter, and change the active flag from B'0' to B'1'. This makes the trusted block usable in other CCA services.</td>
</tr>
</tbody>
</table>

**input_block_identifier_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes in the *input_block_identifier* variable. The maximum length is 3500 bytes.

**input_block_identifier**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing a trusted block key-token or the key label of a trusted block key-token that has been built according to the format specified in “Trusted blocks” on page 647. The trusted block key-token will be updated by the verb and returned in the *trusted_block_identifier* variable.

When the operation is INACTIVE, the trusted block can have MAC protection (for example, due to recycling of an existing trusted block), but typically it does not.

**transport_key_identifier**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing an operational CCA DES key-token or the key label of an operational CCA DES key-token record. The key token must be of type IMP-PKA.

An IMP-PKA key type is an IMPORTER key-encrypting key with only its IMPORT key-usage bit (bit 21) on; its other key-usage bits (IMEX, OPIM, IMIM, and XLATE) must be off.

**Note:** An IMP-PKA control vector can be built using “Control Vector Generate (CSNBCVG)” on page 134 with a key type of IMPORTER and a *rule_array* keyword of IMPORT.

**trusted_block_identifier_length**

**Direction:** Input/Output  
**Type:** Integer
**Trusted Block Create (CSNDTBC)**

A pointer to an integer variable containing the number of bytes of data in the `trusted_block_identifier` variable. The maximum length is 3500 bytes. The output trusted block token can be up to seven bytes longer than the input trusted block token due to padding.

`trusted_block_identifier`

Direction: Output  
Type: String

A pointer to a string variable containing a trusted block token or a label of a trusted block token returned by the verb.

**Restrictions**

The restrictions for CSNDRKX.
1. AES keys are not supported by this verb.
2. Keys with a modulus length greater than 2048 bits are not supported in releases before Release 3.30.

**Required commands**

The required commands for CSNDRKX.

The verb requires the following commands to be enabled in the active role based on the keyword specified for the operation rule:

<table>
<thead>
<tr>
<th><code>rule_array</code></th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>INACTIVE</td>
<td>X'030F'</td>
<td>Trusted Block Create - Create a Trusted Key Block in Inactive form</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>X'0310'</td>
<td>Trusted Block Create - Activate an Inactive Trusted Key Block</td>
</tr>
</tbody>
</table>

**Usage notes**

The usage notes for CSNDRKX.

None

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNDTBCJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNDTBCJ are shown here.

**Format**

```java
public native void CSNDTBCJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger input_block_length,
    byte[] input_block_identifier,
```
Trusted Block Create (CSNDTBC)

byte[] transport_key_identifier,
hikmNativeInteger trusted_blokc_length,
byte[] trusted_blokc_identifier);
Trusted Block Create (CSNDTBC)
Chapter 13. TR-31 symmetric key management verbs

This topic describes the verbs that support TR-31 symmetric key management.

- “Key Export to TR31 (CSNBT31X)” on page 524
- “TR31 Key Import (CSNBT31I)” on page 553
- “TR31 Key Token Parse (CSNBT31P)” on page 578
- “TR31 Optional Data Build (CSNBT31O)” on page 583
- “TR31 Optional Data Read (CSNBT31R)” on page 587
Key Export to TR31 (CSNBT31X)

Use the Key Export to TR31 verb to convert a proprietary CCA external or internal symmetric key-token and its attributes into a non-proprietary key block that is formatted under the rules of TR-31.

After being exported into a TR-31 key block, the key and its attributes are ready to be interchanged with any outside third party who uses TR-31. The verb takes as input either an external or internal fixed-length DES key-token that contains a DES or Triple-DES (TDES) key, along with an internal DES EXPORTER or OKEYXLAT key-encrypting key used to wrap the external TR-31 key block.

The Key Export to TR31 verb is analogous to the Key Export verb, except that Key Export to TR31 accepts an external or internal fixed-length DES key-token as input, instead of only an internal fixed-length DES key-token, and it translates the key to an external non-CCA format instead of an external fixed-length DES key-token. The purpose of both verbs is to export a DES key to another party.

An external-to-external translation would not normally be called an export or import operation. Instead, it would be called a key translation, and would be handled by a verb such as Key Translate2. For practical reasons, the export of an external CCA DES key-token to external TR-31 format is supported by the Key Export to TR31 verb, and the import of an external TR-31 key block to an external CCA DES key-token is supported by the TR31 Key Import verb.

Note that the Key Export to TR31 verb does not support the translation of an external key from encipherment under one key-encrypting key to encipherment under a different key-encrypting key. When converting an external DES key to an external TR-31 format, the key-encrypting key used to wrap the external source key must be the same as the one used to wrap the TR-31 key block. If a translation of an external DES key from encipherment under one key-encrypting to a different key-encrypting key is desired, use the Key Translate or Key Translate2 verbs.

Both CCA and TR-31 define key attributes that control key usage. In both cases, the usage information is securely bound to the key so that the attributes cannot be changed in unintended or uncontrolled ways. CCA maintains its DES key attributes in a control vector (CV), while a TR-31 key block uses fields: key usage, algorithm, mode of use, and exportability.

Each attribute in a CCA control vector falls under one of these categories:
1. There is a one-to-one correspondence between the CV attribute and the TR-31 attribute. For these attributes, conversion is straightforward.
2. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, but the attribute can be automatically translated when performing this export operation.
3. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, in which case a rule-array keyword is defined to specify which attribute is used in the TR-31 key block.
4. Category (1), (2), or (3) applies, but there are some attributes that are lost completely on translation (for example, key-generating bits in key-encrypting keys).
5. None of the above categories applies, because the key type, its attributes, or both simply cannot be reasonably translated into a TR-31 key block.
The control vector is always checked for compatibility with the TR-31 attributes. It is an error if the specified TR-31 attributes are in any way incompatible with the control vector of the input key. In addition, access control points are defined that can be used to restrict the permitted attribute conversions.

The TR-31 key block has a header that can contain optional blocks. Optional blocks become securely bound to the key by virtue of the MAC on the TR-31 key block. The opt_blocks parameter is provided to allow a complete and properly formatted optional block structure to be included as part of the TR-31 key block that is returned by the verb. The TR31 Optional Data Build (CSNBT31O) verb can be used to construct an optional block structure, one optional block at a time.

An optional block has a 2-byte ASCII block ID value that determines the use of the block. The use of a particular optional block is either defined by TR-31, or it has a proprietary use. An optional block that has a block ID with a numeric value is a proprietary block. IBM has its own proprietary optional block to contain a CCA control vector. See “TR-31 optional block data” on page 646 for a description of the IBM-defined data.

To include a copy of the control vector from the DES source key in an optional block of the TR-31 key block, specify the ATTR-CV or INCL-CV control vector transport control keyword in the rule array. If either optional keyword is specified, the verb copies the single-length or double-length control vector field from the source key into the optional data field of the TR-31 header. The TR31 Key Import verb can later extract this data and use it as the control vector for the CCA key that it creates when importing the TR-31 key block. This method provides a way to use TR-31 for transport of CCA keys and to make the CCA key have identical control vectors on the sending and receiving nodes.

The ATTR-CV and INCL-CV keywords both cause the control vector to be included in a TR-31 optional block, but each has a different purpose:

ATTR-CV
Causes a copy of the control vector to be included, but both the TR-31 usage and mode of use fields in the non-optional part of the TR-31 key block header are set to IBM proprietary values. These values, described in “TR-31 optional block data” on page 646, indicate that the usage and mode information are specified in the control vector of the optional block and not in the TR-31 header. The restrictions imposed by the setting of the relevant access control points are bypassed, and any CCA key can be exported as long as the export control fields in the control vector allow it.

INCL-CV
Causes a copy of the control vector to be included as additional detail. The resulting attributes set in the non-optional part of the TR-31 key block header are identical to not using either keyword, except that the value for the number of optional blocks is increased by one. The export operation is still subject to the restrictions imposed by the settings of the relevant access control points.
Key Export to TR31 (CSNBT31X)

Format

The format of CSNBT31X.

```c
CSNBT31X(
    return_code,  
    reason_code,  
    exit_data_length,  
    exit_data,  
    rule_array_count,  
    rule_array,  
    key_version_number,  
    key_field_length,  
    source_key_identifier_length,  
    source_key_identifier,  
    unwrap_kek_identifier_length,  
    unwrap_kek_identifier,  
    wrap_kek_identifier_length,  
    wrap_kek_identifier,  
    opt_blocks_length,  
    opt_blocks,  
    tr31_key_block_length,  
    tr31_key_block
)
```

Parameters

The parameters for CSNBT31X.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. The value must be 2, 3, 4 or 5.

**rule_array**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The `rule_array` keywords for this verb are shown below:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key block protection method (one required). Specifies which version of the TR-31 key block to use for exporting the CCA DES key. The version defines the method by which the key block is cryptographically protected and the content and layout of the block.</td>
<td></td>
</tr>
<tr>
<td>VARXOR-A</td>
<td>Specifies to use the Key Variant Binding Method 2005 Edition. Corresponds to TR-31 Key Block Version ID of “A” (X’41).</td>
</tr>
<tr>
<td>VARDRV-B</td>
<td>Specifies to use the Key Derivation Binding Method 2010 Edition. Corresponds to TR-31 Key Block Version ID of “B” (X’42).</td>
</tr>
<tr>
<td>VARXOR-C</td>
<td>Specifies to use the Key Variant Binding Method 2010 Edition. Corresponds to TR-31 Key Block Version ID of “C” (X’43).</td>
</tr>
</tbody>
</table>
### Control Vector Transport Control

If no keyword from this group is provided, or keyword **INCL-CV** is specified, the control vector in the CCA key token identified by the source_key_identifier parameter is verified to agree with the TR-31 key usage and mode of key use keywords specified from the groups below.

#### INCL-CV

Specifies to copy the control vector from the CCA key-token into an optional proprietary block that is included in the TR-31 key block header. See Table 179 on page 647. The TR-31 key usage and mode of use fields indicate the key attributes. Those attributes, as derived from the keywords specified, must be compatible with the ones in the included CV. In addition, the export of the key must meet the translation and ACP authorizations indicated in the export translation table for the specified keywords. A key usage keyword and a mode of use keyword are required when this keyword is specified.

#### ATTR-CV

Same as keyword **INCL-CV**, except that the key usage field of the TR-31 key block (byte number 5 - 6) is set to the proprietary value ‘10’ (X’3130’), and the mode of use field (byte number 8) is set to the proprietary value ‘1’ (X’31’). These proprietary values indicate that the key usage and mode of use attributes are specified by the CV in the optional block. For this option, only the general ACPs related to export are checked, not the ones relating to specific CCA to TR-31 translations. No key usage or mode of use keywords are allowed when this keyword is specified.

### TR-31 Key Usage Value for Output Key (One Required)

Not valid if **ATTR-CV** keyword is specified. Only those TR-31 usages shown are supported.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>TR-31 key usage</th>
<th>CCA key types</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDK</td>
<td>&quot;B0&quot; KEYGENKY</td>
<td></td>
<td>Specifies to export to a TR-31 base derivation key (BDK). You must select one mode of use keyword from Table 126 on page 531 with this usage keyword. The table shows all of the supported translations for key usage keyword BDK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>CVK</td>
<td>&quot;C0&quot; MAC or DATA</td>
<td></td>
<td>Specifies to export to a TR-31 CVK card verification key. You must select one mode of use keyword from Table 127 on page 533 with this usage keyword. The table shows all of the supported translations for key usage keyword CVK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>ENC</td>
<td>&quot;D0&quot; ENCRYPT, DECIPHER, CIPHER, or DATA</td>
<td></td>
<td>Specifies to export to a TR-31 data encryption key. You must select one mode of use keyword from Table 128 on page 534 with this usage keyword. The table shows all of the supported translations for key usage keyword ENC. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>KEK</td>
<td>&quot;K0&quot; EXPORTER or OKEYXLAT</td>
<td></td>
<td>Specifies to export to a TR-31 key-encryption or wrapping key. You must select one mode of use keyword from Table 129 on page 535 with this usage keyword. The table shows all of the supported translations for key usage keyword KEK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
</tbody>
</table>
### Key Export to TR31 (CSNBT31X)

<table>
<thead>
<tr>
<th>Usage Keyword</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KEK-WRAP</strong></td>
<td>&quot;K1&quot;</td>
<td>IMPORTER or IKEXLAT specifies to export to a TR-31 key block protection key. You must select one mode of use keyword from Table 129 on page 535 with this usage keyword. The table shows all of the supported translations for key usage keyword <strong>KEK-WRAP</strong>. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td><strong>ISOMAC0</strong></td>
<td>&quot;M0&quot;</td>
<td>MAC, MACVER, DATA, DATAM, or DATAMV specifies to export to a TR-31 ISO 16609 MAC algorithm 1 (using TDEA) key. You must select one mode of use keyword from Table 130 on page 536 with this usage keyword. The table shows all of the supported translations for key usage keyword <strong>ISOMAC0</strong>. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td><strong>ISOMAC1</strong></td>
<td>&quot;M1&quot;</td>
<td>MAC, MACVER, DATA, DATAM, or DATAMV specifies to export to a TR-31 ISO 9797-1 MAC algorithm 1 key. You must select one mode of use keyword from Table 130 on page 536 with this usage keyword. The table shows all of the supported translations for key usage keyword <strong>ISOMAC1</strong>. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td><strong>ISOMAC3</strong></td>
<td>&quot;M3&quot;</td>
<td>MAC, MACVER, DATA, DATAM, or DATAMV specifies to export to a TR-31 ISO 9797-1 MAC algorithm 3 key. You must select one mode of use keyword from Table 130 on page 536 with this usage keyword. The table shows all of the supported translations for key usage keyword <strong>ISOMAC3</strong>. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td><strong>PINENC</strong></td>
<td>&quot;P0&quot;</td>
<td>OPINENC or IPINENC specifies to export to a TR-31 PIN encryption key. You must select one mode of use keyword from Table 132 on page 540 with this usage keyword. The table shows all of the supported translations for key usage keyword <strong>PINENC</strong>. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td><strong>PINVO</strong></td>
<td>&quot;V0&quot;</td>
<td>PINGEN or PINVER specifies to export to a TR-31 PIN verification key or other algorithm. You must select one mode of use keyword from Table 132 on page 540 with this usage keyword. The table shows all of the supported translations for key usage keyword <strong>PINVO</strong>. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
</tbody>
</table>
### Key Export to TR31 (CSNBT31X)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PINV3624</td>
<td>&quot;V1&quot;</td>
<td>PINGEN or PINVER Specifies to export to a TR-31 PIN verification, IBM 3624 key. You must select one mode of use keyword from Table 132 on page 540 with this usage keyword. The table shows all of the supported translations for key usage keyword PINV3624. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>VISAPVV</td>
<td>&quot;V2&quot;</td>
<td>PINGEN or PINVER Specifies to export to a TR-31 PIN verification, VISA PVV key. You must select one mode of use keyword from Table 132 on page 540 with this usage keyword. The table shows all of the supported translations for key usage keyword VISAPVV. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>EMVACMK</td>
<td>&quot;E0&quot;</td>
<td>DKGENDKY Specifies to export to a TR-31 EMV/chip issuer master key: application cryptograms key. You must select one mode of use keyword from Table 133 on page 543 with this usage keyword. The table shows all of the supported translations for key usage keyword EMVACMK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>EMVSCMK</td>
<td>&quot;E1&quot;</td>
<td>DKGENDKY Specifies to export to a TR-31 EMV/chip issuer master key: secure messaging for confidentiality key. You must select one mode of use keyword from Table 133 on page 543 with this usage keyword. The table shows all of the supported translations for key usage keyword EMVSCMK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>EMVSIMK</td>
<td>&quot;E2&quot;</td>
<td>DKGENDKY Specifies to export to a TR-31 EMV/chip issuer master key: secure messaging for integrity key. You must select one mode of use keyword from Table 133 on page 543 with this usage keyword. The table shows all of the supported translations for key usage keyword EMVSIMK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>EMVDAMK</td>
<td>&quot;E3&quot;</td>
<td>DATA, MAC, CIPHER, or ENCIPHER Specifies to export to a TR-31 EMV/chip issuer master key: data authentication code key. You must select one mode of use keyword from Table 133 on page 543 with this usage keyword. The table shows all of the supported translations for key usage keyword EMVDAMK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
</tbody>
</table>
**Key Export to TR31 (CSNBT31X)**

<table>
<thead>
<tr>
<th>Key Usage</th>
<th>Mode</th>
<th>Usage Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMVDNMK</td>
<td>&quot;E4&quot;</td>
<td>DKYGENKY</td>
<td>Specifies to export to a TR-31 EMV/chip issuer master key: dynamic numbers key. You must select one mode of use keyword from Table 133 on page 543 with this usage keyword. The table shows all of the supported translations for key usage keyword EMVDNMK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
<tr>
<td>EMVCPMK</td>
<td>&quot;E5&quot;</td>
<td>DKYGENKY</td>
<td>Specifies to export to a TR-31 EMV/chip issuer master key: card personalization key. You must select one mode of use keyword from Table 133 on page 543 with this usage keyword. The table shows all of the supported translations for key usage keyword EMVCPMK. It also shows the access control commands that must be enabled in the active role in order to use the combination of inputs shown.</td>
</tr>
</tbody>
</table>

**TR-31 mode of key use** (one required). Not valid if ATTR-CV keyword is specified. Only those TR-31 modes shown are supported.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>TR-31 mode</th>
<th>Usage keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCDEC</td>
<td>&quot;B&quot;</td>
<td>ENC, KEK, KEK-WRAP</td>
<td>Specifies both encrypt and decrypt, wrap and unwrap.</td>
</tr>
<tr>
<td>DEC-ONLY</td>
<td>&quot;D&quot;</td>
<td>ENC, KEK, KEK-WRAP, PINENC</td>
<td>Specifies to decrypt and unwrap only.</td>
</tr>
<tr>
<td>ENC-ONLY</td>
<td>&quot;E&quot;</td>
<td>ENC, PINENC</td>
<td>Specifies to encrypt and wrap only.</td>
</tr>
<tr>
<td>GENVER</td>
<td>&quot;C&quot;</td>
<td>CVK, ISOMAC0, ISOMAC1, ISOMAC3, PINVO, PINV3624, VISAPVV</td>
<td>Specifies to both generate and verify.</td>
</tr>
<tr>
<td>GEN-ONLY</td>
<td>&quot;G&quot;</td>
<td>CVK, ISOMAC0, ISOMAC1, ISOMAC3, PINVO, PINV3624, VISAPVV</td>
<td>Specifies to generate only.</td>
</tr>
<tr>
<td>VER-ONLY</td>
<td>&quot;V&quot;</td>
<td>CVK, ISOMAC0, ISOMAC1, ISOMAC3, PINVO, PINV3624, VISAPVV</td>
<td>Specifies to verify only.</td>
</tr>
<tr>
<td>DERIVE</td>
<td>&quot;X&quot;</td>
<td>BDK, EMVACMK, EMVSCMK, EMVSIMK, EMVDAMK, EMVDNMK, EMVCPMK</td>
<td>Specifies that key is used to derive other keys.</td>
</tr>
<tr>
<td>ANY</td>
<td>&quot;N&quot;</td>
<td>BDK, PINVO, PINV3624, VISAPVV, EMVACMK, EMVSCMK, EMVSIMK, EMVDAMK, EMVDNMK, EMVCPMK</td>
<td>Specifies no special restrictions (other than restrictions implied by the key usage).</td>
</tr>
</tbody>
</table>
Key Export to TR31 (CSNBT31X)

**TR-31 exportability** (one, optional). Use to set exportability field in TR-31 key block. Defines whether the key may be transferred outside the cryptographic domain in which the key is found.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>TR-31 byte</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP-ANY</td>
<td>“E”</td>
<td>Specifies that the key in the TR-31 key block is exportable under a key-encrypting key in a form that meets the requirements of X9.24 Parts 1 or 2. This is the default. <strong>Note:</strong> A TR-31 key block with a key block version ID of “B” or “C” and an exportability field value of “E” cannot be wrapped by a key-encrypting key that is wrapped in ECB mode (legacy wrap mode). This limitation is because ECB mode does not comply with ANSI X9.24 Part 1.</td>
</tr>
<tr>
<td>EXP-TRST</td>
<td>“S”</td>
<td>Specifies that the key in the TR-31 key block is sensitive, exportable under a key-encrypting key in a form not necessarily meeting the requirements of X9.24 parts 1 or 2.</td>
</tr>
<tr>
<td>EXP-NONE</td>
<td>“N”</td>
<td>Specifies that the key in the TR-31 key block is non-exportable.</td>
</tr>
</tbody>
</table>

**Note:**
1. These keys are the base keys from which derived unique key per transaction (DUKPT) initial keys are derived for individual devices such as PIN pads.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value: "B0" BDK base derivation key.
3. KEYGENKY keys are double length only.

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDK (&quot;B0&quot;)</td>
<td>VARXOR-A</td>
<td>KEYGENKY, double length, UKPT (CV bit 18 = B'1)</td>
<td>ANY (&quot;N&quot;)</td>
<td>X’0180’</td>
<td>TR31 Export - Permit KEYGENKY:UKPT to B0</td>
</tr>
<tr>
<td>BDK (&quot;B0&quot;)</td>
<td>VARDRV-B, VARXOR-C</td>
<td>KEYGENKY, double length, UKPT (CV bit 18 = B'1)</td>
<td>DERIVE (&quot;X&quot;)</td>
<td>X’0180’</td>
<td>TR31 Export - Permit KEYGENKY:UKPT to B0</td>
</tr>
</tbody>
</table>

**Security considerations:**
1. There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "C0" key, if one or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "C0" key cannot be imported to the lower-security CCA DATA key, it can be imported only to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.
2. Since the translation from TR-31 usage "C0" is controlled by rule array keywords when using the CSNBT31I1 verb, it is possible to convert an exported CCA CVVKEY-A or CVVKEY-B key into an AMEX-CSC key or the other way around. This conversion can be restricted by not enabling offsets X’015A’ (TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A) and X’015B’ (TR31 Import - Permit C0 to MAC/MACVER:AMEXCS) at the same time. However, if both CVVKEY-x and AMEX-CSC translation types are required, then offsets X’015A’ and X’015B’ must be enabled. In this case, control is up to the development, deployment, and execution of the applications themselves.
Key Export to TR31 (CSNBUS31X)

Note:
1. Card verification keys are used for computing or verifying (against supplied value) a card verification code with the CVV, CVC, CVC2, and CVV2 algorithms. In CCA, these keys correspond to keys used with two algorithms:
   - Visa CVV and MasterCard CVC codes are generated and verified using the CVV Generate and CVV Verify verbs. These verbs require a key type of DATA or MAC/MACVER with a subtype extension (CV bits 0 - 3) of ANY-MAC, single-length CVVKEY-A and single-length CVVKEY-B, a double-length CVVKEY-A (see CVV Key Combine verb). The MAC generate and the MAC verify (CV bits 20 - 21) key usage values must be set appropriately.
   - American Express CSC codes are generated and verified using the Transaction Validation verb. This verb requires a key type of MAC or MACVER with a subtype extension of ANY-MAC or AMEX-CSC.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "C0"   CVK card verification key.
3. CCA and TR-31 represent CVV keys differently. These differences make representations between CCA and TR-31 incompatible. CCA represents the key-A and key-B keys as two 8-byte (single length) keys, while TR-31 represents these keys as one 16-byte (double length) key. Current Visa standards now require one 16-byte key. The CVV Generate and CVV Verify verbs have support added to accept one 16-byte CVV key, using left and right key parts as key-A and key-B. See "CVV Key Combine (CSNBCKC)" on page 421. This new verb provides a way to combine two single-length MAC-capable keys into one double-length CVV key.
4. Import and export of 8-byte CVVKEY-A and CVVKEY-B MAC/MACVER keys is allowed only using the IBM proprietary TR-31 usage and mode values ("10" and "1", respectively) to indicate encapsulation of the IBM control vector in an optional block, since the 8-byte CVVKEY-A is meaningless and useless as a TR-31 "C0" usage key of any mode.
### Key Export to TR31 (CSNBT31X)

Table 127. Export translation table for a TR-31 CVK card verification key (CVK)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVK (&quot;C0&quot;)</td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>MAC, single or double length, AMEX-CSC (CV bits 0 - 3 = B'0100')</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X'0181'</td>
<td>TR31 Export - Permit MAC/MACVER:AMEX-CSC to C0:G/C/V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC, double length, CVVKEY-A (CV bits 0 - 3 = B'0010')</td>
<td>MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X'0182'</td>
<td>TR31 Export - Permit MAC/MACVER:CVVKEYA to C0:G/C/V</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC, double length, ANY-MAC (CV bits 0 - 3 = B'0000')</td>
<td>MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X'0183'</td>
<td>TR31 Export - Permit MAC/MACVER:ANY-MAC to C0:G/C/V</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA, double length</td>
<td>MAC generate on, MAC verify off (CV bits 20 - 21 = B'10')</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X'0184'</td>
<td>TR31 Export - Permit DATA to C0:G/C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC generate on, MAC verify on (CV bits 20 - 21 = B'11')</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Security consideration:** There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31.
Key Export to TR31 (CSNBT31X)

"D0" key, if one or both applicable Encipher or Decipher control vector bits are on. However, a TR-31 "D0" key cannot be imported to the lower-security CCA DATA key, it can be imported only to a CCA key type of ENCIPHER, DECIPHER, or CIPHER. This restriction eliminates the ability to export a CCA DATA key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to MAC generate and MAC verify.

Note:
1. Data encryption keys are used for the encryption and decryption of data.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "D0" Data encryption

Table 128. Export translation table for a TR-31 data encryption key (ENC)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENC (&quot;D0&quot;)</td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>ENCIPHER, single or double length</td>
<td>ENC-ONLY (&quot;E&quot;)</td>
<td>X'0185'</td>
<td>TR31 Export - Permit ENCIPHER/DECIPHER/CIPHER to D0:E/D/B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DECIPHER, single or double length</td>
<td>DEC-ONLY (&quot;D&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CIPHER, single or double length</td>
<td>ENCDEC (&quot;B&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA, single or double length, Encipher on, Decipher on (CV bits 18 - 19 = B'11')</td>
<td>ENCDEC (&quot;B&quot;)</td>
<td>X'0186'</td>
<td>TR31 Export - Permit DATA to D0:B</td>
</tr>
</tbody>
</table>

Security consideration: The CCA OKEYXLAT, EXPORTER, IKEYXLAT, or IMPORTER KEK translation to a TR-31 "K0" key with mode "B" (both wrap and unwrap) is not allowed for security reasons. Even with access-control point control, this capability would give an immediate path to turn a CCA EXPORTER key into a CCA IMPORTER key, and the other way around.

Note:
1. Key encryption or wrapping keys are used only to encrypt or decrypt other keys, or as a key used to derive keys that are used for that purpose.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "K0" Key encryption or wrapping
   "K1" TR-31 key block protection key
3. CCA mode support is the same for version IDs "B" and "C", because the distinction between TR-31 "K0" and "K1" does not exist in CCA keys. CCA does not distinguish between targeted protocols, and so there is no good way to represent the difference. Also note that most wrapping mechanisms now involve derivation or key variation steps.
### Security consideration:

There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "M0", "M1", or "M3" key, if one or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "M0", "M1", or "M3" key cannot be imported to the lower-security CCA DATA key, it **can be imported only** to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.

### Note:

1. MAC keys are used to compute or verify a code for message authentication.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:

   **"M0"**
   
   ISO 16609 MAC algorithm 1, TDEA
   
   The ISO 16609 MAC algorithm 1 is based on ISO 9797. It is identical to "M1", except that it does not support 8-byte DES keys.

   **"M1"**
   
   ISO 9797 MAC algorithm 1
   
   The ISO 9797 MAC algorithm 1 is identical to "M0", except that it also supports 8-byte DES keys.

   **"M3"**
   
   ISO 9797 MAC algorithm 3
   
   The X9.19 style of Triple-DES MAC.

3. A CCA control vector has no bits defined to limit key usage by algorithm, such as CBC MAC (TR-31 usage "M0" and "M1") or X9.19 (TR-31 usage "M3"). When importing a TR-31 key block, the resulting CCA key token deviates from the restrictions of usages "M0", "M1", and "M3". Importing a TR-31 key block which allows MAC generation ("G" or "C") results in a control vector with the ANY-MAC attribute rather than for the restricted algorithm that is set in the TR-31 key block. The ANY-MAC attribute provides the same restrictions as what CCA currently uses for generating and verifying MACs.
### Table 130. Export translation table for a TR-31 ISO MAC algorithm key (ISOMACn)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
</table>
| **ISOMAC0**
|               | VARXOR-A, VARDRV-B, VARXOR-C       | MAC, double length, MAC generate on (CV bit 20 = B’1’) | GEN-ONLY
|                 |                                     | DATA, double length, MAC generate on (CV bit 20 = B’1’) |          | X’018B’ | TR31 Export - Permit MAC/DATA/DATAM to M0:G/C |
|                 |                                     | MAC, double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B’11’) | VER-ONLY | X’018C’ | TR31 Export - Permit MACVER/DATAMV to M0:V  |
| **ISOMAC1**
|               | VARXOR-A, VARDRV-B, VARXOR-C       | MAC, single or double length, MAC generate on (CV bit 20 = B’1’) | GEN-ONLY
|                 |                                     | DATA, single or double length, MAC generate on (CV bit 20 = B’1’) |          | X’018D’ | TR31 Export - Permit MAC/DATA/DATAM to M1:G/C |
|                 |                                     | MAC, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B’11’) | VER-ONLY | X’018E’ | TR31 Export - Permit MACVER/DATAMV to M1:V   |
### Key Export to TR31 (CSNB31X)

Table 130. Export translation table for a TR-31 ISO MAC algorithm key (ISOMACn) (continued)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISOMAC3 (&quot;M3&quot;)</td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>MAC, single or double length, MAC generate on (CV bit 20 = B'1')</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X'018F</td>
<td>TR31 Export - Permit MAC/DATA/DATAM to M3:G/C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA, single or double length, MAC generate on (CV bit 20 = B'1')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'1')</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATAM, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'1')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA, single or double length, MAC generate on, MAC verify on (CV bits 20 - 21 = B'1')</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MACVER, single or double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td>X'0190'</td>
<td>TR31 Export - Permit MACVER/DATAMV to M3:V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATAMV, single or double length, MAC generate off, MAC verify on (CV bits 20 - 21 = B'01')</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Security notes:**

1. It is highly recommended that the **INCL-CV** keyword be used when exporting PINGEN, PINVER, IPINENC, or OPINENC keys. Using this keyword ensures that importing the TR-31 key block back into CCA will have the desired attributes.

2. TR-31 key blocks that are protected under legacy version ID "A" (keyword VARXOR-A, using the Key Variant Binding Method 2005 Edition) use the same mode of use "N" (keyword ANY) for PINGEN and PINVER keys. For version ID "A" keys only, for a given PIN key usage, enabling both the PINGEN and PINVER access-control points at the same time while enabling offset X'01B0' (for mode "N") is NOT recommended. In other words, for a particular PIN verification usage, you should not simultaneously enable the four commands shown below for that usage:

Table 131. Commands

<table>
<thead>
<tr>
<th>Key type, mode, or version</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;V0&quot;</td>
<td></td>
<td>TR31 Export - Permit PINVER:NO-SPEC to V0</td>
</tr>
</tbody>
</table>

"V0" For usage "V0", a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**
### Key Export to TR31 (CSNBT31X)

Table 131. Commands (continued)

<table>
<thead>
<tr>
<th>Key type, mode, or version</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key type PINGEN</td>
<td>X'0194'</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC to V0</td>
</tr>
<tr>
<td>Mode ANY</td>
<td>X'01B0'</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td>Version VARXOR-A</td>
<td>X'014D'</td>
<td>TR31 Export - Permit Version A TR-31 Key Blocks</td>
</tr>
</tbody>
</table>

"V1" - For usage "V1", a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**

| Key type PINVER            | X'0195' | TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/IBM-PINO to V1 |
| Key type PINGEN            | X'0196' | TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/IBM-PINO to V1 |
| Mode ANY                   | X'01B0' | TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N |
| Version VARXOR-A           | X'014D' | TR31 Export - Permit Version A TR-31 Key Blocks |

"V2" - For usage "V2", a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. **Avoid simultaneously enabling these four commands.**

| Key type PINVER            | X'0197' | TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2 |
| Key type PINGEN            | X'0198' | TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2 |
| Mode ANY                   | X'01B0' | TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N |
| Version VARXOR-A           | X'014D' | vTR31 Export - Permit Version A TR-31 Key Blocks |

Failure to comply with this recommendation allows changing PINVER keys into PINGEN and the other way around.

Note:
Key Export to TR31 (CSNBT31X)

1. PIN encryption keys are used to protect PIN blocks. PIN verification keys are used to generate or verify a PIN using a particular PIN-calculation method for that key type.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "P0" PIN encryption
   "V0" PIN verification, KPV, other algorithm
       Usage "V0" is intended to be a PIN-calculation method "other" than those methods defined for "V1" or "V2". Because CCA does not have a PIN-calculation method of "other" defined, it maps usage "V0" to the subtype extension of NO-SPEC (CV bits 0 - 3 = B'0000'). Be aware that NO-SPEC allows any method, including "V1" and "V2", and that this mapping is suboptimal.
   "V1" PIN verification, IBM 3624
   "V2" PIN verification, Visa PVV
3. Mode must be one of the following values:
   "E" Encrypt/unwrap only
       This mode restricts PIN encryption keys to encrypting a PIN block. May be used to create or reencipher an encrypted PIN block (for key-to-key translation).
   "D" Decrypt/unwrap only
       This mode restricts PIN encryption keys to decrypting a PIN block. Generally used in a PIN translation to decrypt the incoming PIN block.
   "N" No special restrictions (other than restrictions implied by the key usage)
       This mode is used by several vendors for a PIN generate or PIN verification key when the key block version ID is "A".
   "G" Generate only
       This mode is used for a PINGEN key that may not perform a PIN verification. This mode is the only mode available when the control vector in the CCA key-token (applicable when INCL-CV keyword is not provided) does NOT have the EPINVER control vector bit on.
   "V" Verify only
       This mode is used for PIN verification only. This mode is the only mode available when the control vector in the CCA key-token (applicable when INCL-CV is not provided) ONLY has the EPINVER control vector usage bit on (CV bits 18 - 22 = B'00001').
   "C" Both generate and verify (combined)
       This mode is the only output mode available for TR-31 when any of the CCA key-token PIN generating bits are on in the control vector (CPINGENA, EPINGENA, EPINGEN, or CPINGENA) in addition to the EPINVER bit.
### Table 132. Export translation table for a TR-31 PIN encryption or PIN verification key (PINENC, PINVO, PINV3624, VISAPVV)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>PINENC (&quot;P0&quot;)</td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>OPINENC, double length</td>
<td>ENC-ONLY (&quot;E&quot;)</td>
<td>X’0191’</td>
<td>TR31 Export - Permit OPINENC to P0:E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPINENC, double length</td>
<td>DEC-ONLY (&quot;D&quot;)</td>
<td>X’0192’</td>
<td>TR31 Export - Permit IPINENC to P0:D</td>
</tr>
<tr>
<td>PINVO (&quot;V0&quot;)</td>
<td>VARXOR-A</td>
<td>PINVER, double length, NO-SPEC (CV bits 0 - 4 = B’0000’)</td>
<td>ANY (&quot;N&quot;) (requires both commands)</td>
<td>X’0193’</td>
<td>TR31 Export - Permit PINVER:NO-SPEC to V0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X’01B0’</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td></td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>PINVER, double length, NO-SPEC (CV bits 0 - 4 = B’0000’), CPINGEN off, EPINGENA off, EPINGEN off, CPINGENA off (CV bits 18 - 21 = B’0000’)</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td>X’0193’</td>
<td>TR31 Export - Permit PINVER:NO-SPEC to V0</td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>PINGEN, double length, NO-SPEC (CV bits 0 - 4 = B’0000’)</td>
<td>ANY (&quot;N&quot;) (requires both commands)</td>
<td>X’0194’</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC to V0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X’01B0’</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td></td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>PINGEN, double length, NO-SPEC (CV bits 0 - 4 = B’0000’), EPINVER off (CV bit 22 = B’0’)</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X’0194’</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC to V0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PINGEN, double length, NO-SPEC (CV bits 0 - 4 = B’0000’), EPINVER on (CV bit 22 = B’1’)</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 132. Export translation table for a TR-31 PIN encryption or PIN verification key (PINENC, PINVO, PINV3624, VISAPVV) (continued)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>PINV3624 (&quot;V1&quot;)</td>
<td>VARXOR-A</td>
<td>PINVER, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B’0000’ or B’0001’)</td>
<td>ANY (&quot;N&quot;) (requires both commands)</td>
<td>X'0195'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/IBM-PINO to V1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X'01B0'</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td></td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>PINVER, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B’0000’ or B’0001’), CPINGEN off, EPINGEN off, EPINGEN off, CPINGEN off (CV bits 18 - 21 = B’0000’)</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td>X'0195'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/IBM-PINO to V1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X'01B0'</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>PINGEN, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B’0000’ or B’0001’)</td>
<td>ANY (&quot;N&quot;) (requires both commands)</td>
<td>X'0196'</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/IBM-PINO to V1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X'01B0'</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td></td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>PINGEN, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B’0000’ or B’0001’), EPINVER off (CV bit 22 = B’0’)</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X'0196'</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC/IBM-PIN/IBM-PINO to V1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PINGEN, double length, NO-SPEC or IBM-PIN/IBM-PINO (CV bits 0 - 4 = B’0000’ or B’0001’), EPINVER on (CV bit 22 = B’1’)</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key usage keyword</td>
<td>Key block protection method keyword</td>
<td>CCA key type and required control vector attributes</td>
<td>Mode of use keyword</td>
<td>Offset</td>
<td>Command</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------</td>
<td>-----------------------------------------------</td>
<td>--------------------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>VISAPVV (&quot;V2&quot;)</td>
<td>VARXOR-A</td>
<td>PINVER, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010')</td>
<td>ANY (&quot;N&quot;) (requires both commands)</td>
<td>X'0197'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X'01B0'</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>PINVER, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010'), CPINGEN off, EPINGENA off, EPINGEN off, CPINGENA off (CV bits 18 - 21 = B'0000')</td>
<td>VER-ONLY (&quot;V&quot;)</td>
<td>X'0197'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>PINGEN, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010')</td>
<td>ANY (&quot;N&quot;) (requires both commands)</td>
<td>X'0198'</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X'01B0'</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A, VARDRV-B, VARXOR-C</td>
<td>PINGEN, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010'), EPINVER off (CV bit 22 = B'0')</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td>X'0198'</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PINGEN, double length, NO-SPEC or VISA-PVV (CV bits 0 - 4 = B'0000' or B'0010'), EPINVER on (CV bit 22 = B'1')</td>
<td>GENVER (&quot;C&quot;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

1. EMV/chip issuer master-key keys are used by the chip cards to perform cryptographic operations or, in some cases, to derive keys used to perform operations. In CCA, these keys are (a) diversified key-generating keys (key type DKYGENKY), allowing derivation of operational keys, or (b) operational keys. Note that in this context, "master key" has a different meaning than for CCA. These master keys, also called KMCs, are described by EMV as DES master keys for personalization session keys. They are used to derive the corresponding chip card master keys, and not typically used directly for cryptographic operations other than key derivation. In CCA, these keys are usually key generating keys with derivation level DKYL1 (CV bits 12 - 14 = B'0001'), used to derive other key generating keys (the chip card master keys). For some cases, or for older EMV key derivation methods, the issuer master keys could be level DKYL0 (CV bits 12 - 14 = B'0000').

2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:

- "E0" Application cryptograms
- "E1" Secure messaging for confidentiality
- "E2" Secure messaging for integrity
- "E3" Data authentication code
- "E4" Dynamic numbers
"E5"  Card personalization
3. EMV support in CCA is different than TR-31 support, and CCA key types do not match TR-31 types.
4. DKYGENKY keys are double length only.

Table 133. Export translation table for a TR-31 EMV/chip issuer master-key key (DKYGENKY, DATA)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMVACMK ('E0')</td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMC (CV bits 19 - 22 = B'0010')</td>
<td>ANY ('N')</td>
<td>X'0199' TR31 Export - Permit DKYGENKY:DKYL0 +DMAC to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMC (CV bits 19 - 22 = B'0011')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMV (CV bits 19 - 22 = B'0011')</td>
<td>ANY ('N')</td>
<td>X'019A' TR31 Export - Permit DKYGENKY:DKYL0 +DMV to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMV (CV bits 19 - 22 = B'0011')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMC (CV bits 19 - 22 = B'0010')</td>
<td>ANY ('N')</td>
<td>X'019B' TR31 Export - Permit DKYGENKY:DKYL1 +DMAC to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMC (CV bits 19 - 22 = B'0011')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMV (CV bits 19 - 22 = B'0011')</td>
<td>ANY ('N')</td>
<td>X'019C' TR31 Export - Permit DKYGENKY:DKYL1 +DMV to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMV (CV bits 19 - 22 = B'0011')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DALL (CV bits 19 - 22 = B'0111')</td>
<td>ANY ('N')</td>
<td>X'019D' TR31 Export - Permit DKYGENKY:DKYL1 +DALL to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DALL (CV bits 19 - 22 = B'0111')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')</td>
<td>ANY ('N')</td>
<td>X'019F' TR31 Export - Permit DKYGENKY:DKYL0 +DDATA to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMPIN (CV bits 19 - 22 = B'1001')</td>
<td>ANY ('N')</td>
<td>X'01A0' TR31 Export - Permit DKYGENKY:DKYL0 +DMPIN to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMPIN (CV bits 19 - 22 = B'1001')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'0111')</td>
<td>ANY ('N')</td>
<td>X'01A1' TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'0111')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DDATA (CV bits 19 - 22 = B'0001')</td>
<td>ANY ('N')</td>
<td>X'01A2' TR31 Export - Permit DKYGENKY:DKYL1 +DDATA to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DDATA (CV bits 19 - 22 = B'0001')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMPIN (CV bits 19 - 22 = B'1001')</td>
<td>ANY ('N')</td>
<td>X'01A3' TR31 Export - Permit DKYGENKY:DKYL1 +DMPIN to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMPIN (CV bits 19 - 22 = B'1001')</td>
<td>DERIVE ('X')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DALL (CV bits 19 - 22 = B'0111')</td>
<td>ANY ('N')</td>
<td>X'01A4' TR31 Export - Permit DKYGENKY:DKYL1 +DALL to E1</td>
<td></td>
</tr>
</tbody>
</table>
### Key Export to TR31 (CSNBT31X)

Table 133. Export translation table for a TR-31 EMV/chip issuer master-key key (DKYGENKY, DATA) (continued)

<table>
<thead>
<tr>
<th>Key usage keyword</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EMVSIMK</strong> (&quot;E2&quot;)</td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')</td>
<td>ANY (&quot;N&quot;)</td>
<td>X'01A5'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DMAC to E2</td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'1'111)</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMAC (CV bits 19 - 22 = B'0010')</td>
<td>ANY (&quot;N&quot;)</td>
<td>X'01A6'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E2</td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL1 (CV bits 12 - 14 = B'001'), DMAC (CV bits 19 - 22 = B'1'111)</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DATA, double length</td>
<td>ANY (&quot;N&quot;)</td>
<td>X'01A7'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1 +DMAC to E2</td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>MAC (not MACVER), double length</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>MAC (not MACVER), double length</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>CIPHER, double length</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>ENCIPHER, double length</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EMVDAMK</strong> (&quot;E3&quot;)</td>
<td>VARXOR-A</td>
<td>DATA, double length</td>
<td>ANY (&quot;N&quot;)</td>
<td>X'01A9'</td>
<td>TR31 Export - Permit DATA/MAC/CIPHER/ENCIPHER to E3</td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>MAC (not MACVER), double length</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>MAC (not MACVER), double length</td>
<td>GEN-ONLY (&quot;G&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>CIPHER, double length</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>ENCIPHER, double length</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EMVDNMK</strong> (&quot;E4&quot;)</td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')</td>
<td>ANY (&quot;N&quot;)</td>
<td>X'01AA'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DDATA to E4</td>
</tr>
<tr>
<td></td>
<td>VARDRV-B, VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')</td>
<td>DERIVE (&quot;X&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'1'111)</td>
<td>ANY (&quot;N&quot;)</td>
<td>X'01AB'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E4</td>
</tr>
</tbody>
</table>
Key Export to TR31 (CSNBT31X)

Table 133. Export translation table for a TR-31 EMV/chip issuer master-key key (DKYGENKY, DATA) (continued)

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword</th>
<th>CCA key type and required control vector attributes</th>
<th>Mode of use keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMVCPMK ('E5')</td>
<td>VARXOR-A</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DEXP (CV bits 19 - 22 = B'0101')</td>
<td>ANY ('N')</td>
<td>X'01AC'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DEXP to E5</td>
</tr>
<tr>
<td></td>
<td>VARDRV-B</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DEXP (CV bits 19 - 22 = B'0101')</td>
<td>DERIVE ('X')</td>
<td>X'01AD'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DEXP to E5</td>
</tr>
<tr>
<td></td>
<td>VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')</td>
<td>ANY ('N')</td>
<td>X'01AE'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DMAC to E5</td>
</tr>
<tr>
<td></td>
<td>VARDRV-B</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')</td>
<td>DERIVE ('X')</td>
<td>X'01AF'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DDATA to E5</td>
</tr>
<tr>
<td></td>
<td>VARXOR-C</td>
<td>DKYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DALL (CV bits 19 - 22 = B'1111')</td>
<td>ANY ('N')</td>
<td>X'01AF'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0 +DALL to E5</td>
</tr>
</tbody>
</table>

**key_version_number**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing two numeric ASCII bytes that are copied into the key version number field of the output TR-31 key block. Use a value of "00" (X'3030') if no key version number is needed.

This value is ignored if the key identified by the source_key_identifier parameter contains a partial key, that is, the KEY-PART bit (CV bit 44) is on in the control vector. When a partial key is passed, the verb sets the key version number field in the TR-31 key block to "c0" (X'6330'). According to TR-31, this value indicates that the TR-31 key block contains a component of a key (key part).

**key_field_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the length of the key field that is encrypted in the TR-31 block. The length must be a multiple of the DES cipher block size, which is eight. It must also be greater than or equal to the length of the cleartext key passed using the source_key_identifier parameter plus the length of the key length field (two bytes) that precedes this key in the TR-31 block. For example, if the source key is a double-length TDES key (its length is 16 bytes), then the key field length must be greater than or equal to (16 + 2) bytes, and must also be a multiple of 8. This means that the minimum key_field_length in this case would be 24.

TR-31 allows a variable number of padding bytes to follow the cleartext key, and the application designer can choose to pad with more than the minimum number of bytes needed to form a block that is a multiple of 8. This padding is generally done to hide the length of the cleartext key from those who cannot decipher that key. Most often, all keys (single, double, or triple length) are padded to the same length so that it is not possible to determine which length is carried in the TR-31 block by examining the encrypted block.
Key Export to TR31 (CSNBT31X)

**Note:** This parameter is not expected to allow for ASCII encoding of the encrypted data stored in the key field according to the TR-31 specification. For example, when a value of 24 is passed here, following the minimum example above, the length of the final ASCII-encoded encrypted data in the key field in the output TR-31 key block is 48 bytes.

**source_key_identifier_length**
- **Direction:** Input
- **Type:** Integer
- A pointer to an integer variable containing the length in bytes of the `source_key_identifier` variable. The value must be 64.

**source_key_identifier**
- **Direction:** Input
- **Type:** String
- A pointer to a string variable containing either the key label for the source key, or the key token containing the source key. The source key is the key that is to be exported. The key must be a CCA fixed-length DES internal or external key-token. If the source key is an external token, an identifier for the KEK that wraps the source key must be identified by the `unwrap_kek_identifier` parameter. TR-31 currently supports only DES and TDES keys. AES is not supported.

**unwrap_kek_identifier_length**
- **Direction:** Input
- **Type:** Integer
- A pointer to an integer variable containing the length in bytes of the `unwrap_kek_identifier` variable. The value must be greater than or equal to 0. A null key-token can have a length of 1. Set this value to 64 for a key label or a KEK.

**unwrap_kek_identifier**
- **Direction:** Input
- **Type:** String
- A pointer to a string variable containing either the key label for the source key KEK, or the key token containing the source key KEK when the source key is an external CCA key token, and a NULL key token otherwise. The source key KEK can also be the wrapping key for the key that is to be exported if the `wrap_kek_identifier` is not specified. The source key KEK must be a CCA internal DES KEK token of type EXPORTER or OKEYXLAT.

**Note:** ECB-mode wrapped DES keys (CCA legacy wrap mode) cannot be used to wrap or unwrap TR-31 "B" or "C" key blocks that have or will have "E" exportability, because ECB-mode does not comply with ANSI X9.24 Part 1.

**wrap_kek_identifier_length**
- **Direction:** Input
- **Type:** Integer
- A pointer to an integer variable containing the length in bytes of the `wrap_kek_identifier` variable. Set this value to 64.

**wrap_kek_identifier**
- **Direction:** Input
- **Type:** String
Key Export to TR31 (CSNBT31X)

A pointer to a string variable containing an operational fixed-length DES key-token with a key type of EXPORTER or OKEYXLAT to use for wrapping the output TR-31 key block, a null key token, or a key label of such a key in DES key-storage. If the identified key token is not null, the key token must have the same key that is in the key-token identified by the unwrap_kek_identifier parameter. If it is null, then the key identified by the unwrap_kek_identifier parameter is also used for wrapping the output TR-31 key block.

Note: ECB-mode wrapped DES keys (CCA legacy wrap mode) cannot be used to wrap or unwrap TR-31 “B” or “C” key blocks that have or will have “E” exportability, because ECB-mode does not comply with ANSI X9.24 Part 1. This parameter exists to allow for KEK separation. It is possible that KEKs are restricted as to what they can wrap, such that a KEK for wrapping CCA external keys might not be usable for wrapping TR-31 external keys, or the other way around.

opt_blocks_length

Direction: Input
Type: Integer

A pointer to an integer variable that specifies the length in bytes of the opt_blocks variable. If no optional data is to be included in the TR-31 key block, set this value to zero.

opt_blocks

Direction: Input
Type: String

A pointer to a string variable containing optional blocks data that is to be included in the output TR-31 key block. The optional blocks data can be constructed using the TR31 Optional Data Build verb.

Note: The Padding Block, ID "PB" cannot be added by the user, and therefore is not accepted in the opt_blocks parameter. CCA adds a Padding Block of the appropriate size as needed when building the TR-31 key block in Key Export to TR31. The Padding Block for optional blocks serves no security purpose, unlike the padding in the encrypted key portion of the payload.

tr31_key_block_length

Direction: Input/Output
Type: Integer

A pointer to an integer variable containing the length in bytes of the tr31_key_block variable. On input, specify the size of the application program buffer available for the output key-token. On return from the verb, this variable is updated to contain the actual length of that returned token. TR-31 key blocks are variable in length.

tr31_key_block

Direction: Output
Type: String

A pointer to a string variable containing the output key block produced by the verb. The output key block contains the external form of the key created by the verb, wrapped according to the method specified.
Key Export to TR31 (CSNBT31X)

**Note:** The padding optional block in the output TR-31 key block can be present with zero data bytes. This situation can occur if the optional block portion of the header needs exactly four bytes of padding, the size of an optional block header without the data portion. The data portion is defined as optional by TR-31, which allows this.

**Restrictions**

The restrictions for CSNBT31X.

- The only proprietary values for the TR-31 header fields supported by this verb are those values defined and used by IBM CCA when carrying a control vector in an optional block in the header.
- AES is not currently supported for TR-31 key blocks.
- The export is prohibited if the CCA key does not have attributes XPORT-OK (CV bit 17 = B’1’) and T31XPTOK (CV bit 57 = B’1’).

**Required commands**

The required commands for CSNBT31X.

The Key Export to TR31 verb requires the following commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR-31 key block protection method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARXOR-A</td>
<td>X’014D’</td>
<td>TR31 Export - Permit Version A TR-31 Key Blocks</td>
</tr>
<tr>
<td>VARDRV-B</td>
<td>X’014E’</td>
<td>TR31 Export - Permit Version B TR-31 Key Blocks</td>
</tr>
<tr>
<td>VARXOR-C</td>
<td>X’014F’</td>
<td>TR31 Export - Permit Version C TR-31 Key Blocks</td>
</tr>
<tr>
<td>INCL-CV</td>
<td>X’0158’</td>
<td>TR31 Export - Permit Any CCA Key if INCL-CV Is Specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When providing the INCL-CV keyword:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If this command is enabled in the active role, the key-type specific commands are not checked.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If this command is not enabled in the active role, the key-type specific commands are required.</td>
</tr>
<tr>
<td>ATTR-CV</td>
<td>N/A</td>
<td>Note: No commands relating to specific CCA to TR-31 transitions are checked when this keyword is specified. Only the general access control commands related to export are checked.</td>
</tr>
</tbody>
</table>

Be aware of the interaction of access-control point X’0158’ (TR31 Export - Permit Any CCA Key if INCL-CV Is Specified) with the INCL-CV keyword. Without the INCL-CV keyword, most export translations are guarded by key-type-specific access control points, to guard the source CCA system against attacks involving reimport of the exported key under ambiguous circumstances. When the control vector is exported along with the key as an optional block securely bound to the encrypted key, the source system is somewhat protected because the key on import is allowed to have only the form of the included control vector. No expansion of capability is allowed. If access-control point X’0158’ is enabled in the active role, the key-type-specific access control points are not checked when INCL-CV is provided. If ACP X’0158’ is not enabled, the type-specific access control points are still required. The following figure illustrates this concept:
Be aware of access-control point X'01B0' (TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N) for export of PINGEN or PINVER keys to wrapping method "A", usage "V0", "V1", or "V2", and mode "N". TR-31 key blocks with legacy key usage "A" (key block protected using the Key Variant Binding Method 2005 Edition) use the same mode "N" for PINGEN as well as PINVER keys. For usage "A" keys only, enabling a PINGEN and PINVER access-control point while enabling offset X'01B0' (for keyword ANY, mode "N") is NOT recommended. Failure to comply with this recommendation allows changing PINVER keys into PINGEN and the other way around.

In addition to the above commands, the verb requires these additional commands to be enabled in the active role depending on the TR-31 key usage rule-array keyword provided and additional information as shown in the table referenced in the rightmost column:

<table>
<thead>
<tr>
<th>TR-31 key usage keyword</th>
<th>Offset</th>
<th>Command</th>
<th>Specific key type and control vector attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B0': TR-31 BDK base derivation keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BDK</td>
<td>X'0180'</td>
<td>TR31 Export - Permit KEYGENKY:UKPT to B0</td>
<td>See Table 126 on page 531</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'C0': TR-31 CVK card verification keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVK</td>
<td>X'0181'</td>
<td>TR31 Export - Permit MAC/MACVER:AMEX-CSC to C0:G/C/V</td>
<td>See Table 127 on page 533</td>
</tr>
<tr>
<td></td>
<td>X'0182'</td>
<td>TR31 Export - Permit MAC/MACVER:CVV-KEYA to C0:G/C/V</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0183'</td>
<td>TR31 Export - Permit MAC/MACVER:ANY-MAC to C0:G/C/V</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0184'</td>
<td>TR31 Export - Permit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'D0': TR-31 data encryption keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENC</td>
<td>X'0185'</td>
<td>TR31 Export - Permit ENCIPHER/DECIPHER/CIPHER to D0:E/D/B</td>
<td>See Table 128 on page 534</td>
</tr>
<tr>
<td></td>
<td>X'0186'</td>
<td>TR31 Export - Permit DATA to D0:B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"E0", "E1", "E2", "E3", "E4", and "E5": TR-31 EMC/chip issuer master-key keys
### Key Export to TR31 (CSNBT31X)

<table>
<thead>
<tr>
<th>TR-31 key usage keyword</th>
<th>Offset</th>
<th>Command</th>
<th>Specific key type and control vector attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMVACMK</td>
<td>X'0199'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E0</td>
<td>See Table 129 on page 535</td>
</tr>
<tr>
<td></td>
<td>X'019A'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMV to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'019B'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'019C'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'019D'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMV to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'019E'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DALL to E0</td>
<td></td>
</tr>
<tr>
<td>EMVSCMK</td>
<td>X'019F'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E1</td>
<td>See Table 129 on page 535</td>
</tr>
<tr>
<td></td>
<td>X'01A0'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMPIN to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01A1'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01A2'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DDATA to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01A3'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMPIN to E1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01A4'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DALL to E1</td>
<td></td>
</tr>
<tr>
<td>EMVSIMK</td>
<td>X'01A5'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E2</td>
<td>See Table 129 on page 535</td>
</tr>
<tr>
<td></td>
<td>X'01A6'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01A7'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01A8'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DALL to E2</td>
<td></td>
</tr>
<tr>
<td>EMVDAMK</td>
<td>X'01A9'</td>
<td>TR31 Export - Permit DATA/MAC/CIPHER/ENCIPHER to E3</td>
<td>See Table 129 on page 535</td>
</tr>
<tr>
<td>EMVDNMK</td>
<td>X'01AA'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E4</td>
<td>See Table 129 on page 535</td>
</tr>
<tr>
<td></td>
<td>X'01AB'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E4</td>
<td></td>
</tr>
<tr>
<td>EMVCPMK</td>
<td>X'01AC'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DEXP to E5</td>
<td>See Table 129 on page 535</td>
</tr>
<tr>
<td></td>
<td>X'01AD'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01AE'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'01AF'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E5</td>
<td></td>
</tr>
</tbody>
</table>

"K0" and "K1": TR-31 key encryption or wrapping, or key block protection keys
# Key Export to TR31 (CSNBT31X)

<table>
<thead>
<tr>
<th>TR-31 key usage keyword</th>
<th>Offset</th>
<th>Command</th>
<th>Specific key type and control vector attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEK</td>
<td>X’0187’</td>
<td>TR31 Export - Permit EXPORTER/OKEYXLAT to K0:E</td>
<td>See Table 130 on page 536</td>
</tr>
<tr>
<td></td>
<td>X’0188’</td>
<td>TR31 Export - Permit IMPORTER/IKEYXLAT to K0:D</td>
<td></td>
</tr>
<tr>
<td>KEK-WRAP</td>
<td>X’0189’</td>
<td>TR31 Export - Permit EXPORTER/OKEYXLAT to K1:E</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’018A’</td>
<td>TR31 Export - Permit IMPORTER/IKEYXLAT to K1:D</td>
<td></td>
</tr>
<tr>
<td>&quot;M0&quot;, &quot;M1&quot;, and &quot;M3&quot;: TR-31 ISO MAC algorithm keys</td>
<td></td>
<td>See Table 130 on page 536</td>
<td></td>
</tr>
<tr>
<td>ISOMAC0</td>
<td>X’018B’</td>
<td>TR31 Export - Permit MAC/DATA/DATAM to M0:G/C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’018C’</td>
<td>TR31 Export - Permit MACVER/DATAMV to M0:V</td>
<td></td>
</tr>
<tr>
<td>ISOMAC1</td>
<td>X’018D’</td>
<td>TR31 Export - Permit MAC/DATA/DATAM to M1:G/C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’018E’</td>
<td>TR31 Export - Permit MACVER/DATAMV to M1:V</td>
<td></td>
</tr>
<tr>
<td>ISOMAC3</td>
<td>X’018F’</td>
<td>TR31 Export - Permit MAC/DATA/DATAM to M3:G/C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’0190’</td>
<td>TR31 Export - Permit MACVER/DATAMV to M3:V</td>
<td></td>
</tr>
<tr>
<td>&quot;P0&quot;, &quot;V0&quot;, &quot;V1&quot;: TR-31 PIN encryption or PIN verification keys</td>
<td></td>
<td>See Table 133 on page 543</td>
<td></td>
</tr>
<tr>
<td>PINENC</td>
<td>X’0191’</td>
<td>TR31 Export - Permit OPINENC to P0:E</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’0192’</td>
<td>TR31 Export - Permit IPINENC to P0:D</td>
<td></td>
</tr>
<tr>
<td>PINVO</td>
<td>X’0193’</td>
<td>TR31 Export - Permit PINVER:NO-SPEC to V0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’0194’</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC to V0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’01B0’</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
<td></td>
</tr>
<tr>
<td>PINV3624</td>
<td>X’0195’</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/IBMPIN/IBM-PINO to V1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’0196’</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC/IBMPIN/IBM-PINO to V1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’01B0’</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
<td></td>
</tr>
<tr>
<td>VISAPVV</td>
<td>X’0197’</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’0198’</td>
<td>TR31 Export - Permit PINGEN:NO-SPEC/VISA-PVV to V2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X’01B0’</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
<td></td>
</tr>
</tbody>
</table>

## JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBT31XJ.

See "Building Java applications to use with the CCA JNI" on page 26.
Key Export to TR31 (CSNB31X)

The parameters for CSNB31XJ are shown here.

**Format**

```java
public native void CSNB31XJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] key_version_number,
    hikmNativeInteger key_field_length,
    hikmNativeInteger source_key_identifier_length,
    byte[] source_key_identifier,
    hikmNativeInteger unwrap_kek_identifier_length,
    byte[] unwrap_kek_identifier,
    hikmNativeInteger wrap_kek_identifier_length,
    byte[] wrap_kek_identifier,
    hikmNativeInteger opt_blocks_length,
    byte[] opt_blocks,
    hikmNativeInteger tr31_key_block_length,
    byte[] tr31_key_block);
```
Use the TR31 Key Import verb to convert a non-proprietary external key-block that is formatted under the rules of TR-31 into a proprietary CCA external or internal fixed-length DES key-token with its attributes in a control vector.

After being imported into a CCA key-token, the key and its attributes are ready to be used in a CCA system. The verb takes as input an external TR-31 key block and the internal DES IMPORTER or IKEYXLAT key-encrypting key of the key that was used to wrap the TR-31 key block.

The TR31 Key Import verb is analogous to the existing Key Import verb, except that TR31 Key Import accepts an external non-CCA DES key-token instead of an external CCA fixed-length DES key-token, and it translates the key to either an external or internal fixed-length DES key-token instead of only an internal fixed-length DES key-token. An import by TR31 Key Import to an external key-token requires a suitable internal fixed-length DES key-encrypting key. The purpose of both verbs is to import a DES key from another party.

An external-to-external translation would not normally be called an export or import operation. Instead, it would be called a key translation and would be handled by a verb such as Key Translate or Key Translate2. For practical reasons, the export of an external CCA DES key-token to an external TR-31 format is supported by the Key Export to TR31 verb, and the import of an external TR-31 key block to an external CCA DES key-token format is supported by the TR31 Key Import verb.

Note that the TR31 Key Import verb does not support the translation of an external key from encipherment under one key-encrypting key to encipherment under a different key-encrypting key. When converting an external TR-31 key block to an external fixed-length DES key-token, the key-encrypting key used to wrap the external TR-31 key block must be the same as the one used to wrap the external fixed-length DES key-token. Use the Key Translate or Key Translate2 verbs for switching external key wrapping keys: the normal function of those verbs.

Both CCA and TR-31 define key attributes that control key usage. In both cases, the usage information is securely bound to the key so that the attributes cannot be changed in unintended or uncontrolled ways. CCA maintains its DES key attributes in a control vector (CV), while a TR-31 key block uses fields: key usage, algorithm, mode of use, and exportability.

Each attribute in a CCA control vector falls under one of these categories:

1. There is a one-to-one correspondence between the CV attribute and the TR-31 attribute. For these attributes, conversion is straightforward.
2. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, but the attribute can be automatically translated when performing this export operation.
3. There is not a one-to-one correspondence between the CV attribute and the TR-31 attribute, in which case a rule-array keyword has been defined to specify which attribute is to be used in the TR-31 key block.
4. Category (1), (2), or (3) applies, but there are some attributes that are lost completely on translation (for example, key-generating bits in key-encrypting keys).
5. None of the above categories applies, because the key type, its attributes, or both simply cannot be reasonably translated into a TR-31 key block.
TR31 Key Import (CSNBT31I)

The control vector is always checked for compatibility with the TR-31 attributes. It is an error if the specified control vector attributes are in any way incompatible with the attributes of the input key. In addition, access control points are defined that can be used to restrict the permitted attribute conversions.

The import operation produces the CCA external or internal fixed-length DES key-token as its output. It does not return any field values or optional block data from the TR-31 key block header. To obtain the header field values, use the TR31 Key Token Parse verb. To obtain optional block data from the header, use the TR31 Optional Data Read verb.

An optional control vector transport control rule-array keyword can be passed to the Key Export to TR31 verb. Such a keyword specifies that the verb is to copy the control vector from the CCA DES key into the TR-31 key block. A copy of the control vector is passed in an IBM-proprietary optional block. See “TR-31 optional block data” on page 646.

If the TR-31 key block contains an IBM-proprietary block, the TR31 Key Import verb verifies that the control vector is compatible with the attributes in the TR-31 key block. If any incompatibility is found, the verb rejects the import. If the control vector is valid for the key, the verb uses it for the control vector of the CCA DES key-token. Note that the import operation is always subject to the restrictions imposed by the relevant access control points, even if a control vector is received.

A control vector, if present, can be in the TR-31 key block in one of two ways, depending on the control vector transport control keyword specified in the rule array of the Key Export to TR31 verb when the key was exported. One keyword option is ATTR-CV, and the other is INCL-CV:

ATTR-CV
Causes a copy of the control vector to be included in the TR-31 key block. The TR-31 key usage and mode of use fields are set to IBM proprietary. See “TR-31 optional block data” on page 646. These proprietary values indicate that the usage and mode information is contained in the included control vector. In this case, if the TR31 Key Import verb successfully verifies that the included control vector does not conflict with the rule-array keywords specified, it uses it as the control vector for the imported CCA DES key-token.

INCL-CV
Causes a copy of the control vector to be included in the TR-31 key block. The TR-31 key usage and mode of use fields contain attributes from the set defined in the TR-31 standard. In this case, the TR31 Key Import verb verifies that the usage and mode information in those fields are compatible with the included control vector. The verb also verifies that no rule array keywords conflict with the control vector.

Note that the included CV could have more capability from a CCA perspective than the TR-31 usage and mode fields indicate. This difference is not an error, because the key block binding methods give the importer assurance that the key block optional blocks are as secure as any other attribute.

Special notes
Additional information about CSNBT31I.
1. Several import situations might require keywords. Keywords are ignored for INCL-CV scenarios unless they directly conflict with the included CV. For
example, the verb returns an error if the control vector indicates that a
DKYGENKY key has a subtype of DKYL0 and the user specifies the DKYL1
keyword.

2. Be aware of the interaction of ACP X'0158' (TR31 Export - Permit Any CCA Key
if INCL-CV Is Specified) with the INCL-CV keyword for the Key Export to
TR31 verb. Without the INCL-CV keyword specified, most export translations
are guarded by key-type specific ACPs. These ACPs are used to guard the
source CCA system against attacks involving reimport of the exported key
under ambiguous circumstances. When the control vector is exported in an
optional block along with the key, it is securely bound to the encrypted key.
This somewhat protects the source system because the key on import is
allowed to have only the form of the included control vector. Expansion of
capability is blocked. If ACP X'0158' is not enabled in the active role, the
type-specific ACPs are still required. However, if ACP X'0158' is enabled, the
key-type specific ACPs are not checked when INCL-CV is specified.

3. Be aware of ACP X'017C' (TR31 Import - Permit V0/V1/V2:N to
PINGEN/PINVER) for import of PINGEN or PINVER keys to wrapping mode
A, usage V0, V1, V2, and mode N.

The extra translation-specific ACPs are intended to enable control of situations
where the CCA imported key type is ambiguous based on the TR-31 key attributes.
This ambiguity is never the case when INCL-CV has been specified with the Key
Export to TR31 verb, which ensures that the imported TR-31 key block has a valid
CV to precisely control the resultant CCA key. Therefore, there are no
translation-specific ACPs governing INCL-CV import translations.

Examples

Examples for CSNBT31I.

1. A full MAC key that is exported as TR-31 "C0" key block with an included
control vector will be re-imported as a full MAC key.

2. A DKYGENKY key with key usage DALL key exported as a TR-31 "E0", "E1",
or "E2" key block with an included control vector will be re-imported as a
DKYGENKY key with key usage DALL, even though the "E0", "E1", and "E2
types are more restricted.

Format

The format of CSNBT31I.

```c
CSNBT31I(  
  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  rule_array_count,  
  rule_array,  
  tr31_key_block_length,  
  tr31_key_block,  
  unwrap_kek_identifier_length,  
  unwrap_kek_identifier,  
  wrap_kek_identifier_length,  
  wrap_kek_identifier,  
  output_key_identifier_length,  
  output_key_identifier,  
  num_opt_blocks,  
  cv_source,  
  protection_method)
```
The parameter definitions for CSNBT31I.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 1, 2, 3, or 4.

**rule_array**

**Direction:** Input  
**Type:** String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token identifier</strong> (one required)</td>
<td></td>
</tr>
<tr>
<td>INTERNAL</td>
<td>Specifies to return the output key in an internal CCA key-token.</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>Specifies to return the output key in an external CCA key-token, wrapped by the transport key identified by the wrap_kek_identifier parameter.</td>
</tr>
</tbody>
</table>

**CCA output key usage subgroups** (One from one subgroup required based on TR-31 input key usage. Keywords for the subgroup are valid only for given TR-31 key usage.)

**Note:** None of the following keywords are allowed if the TR-31 key block provided as input has an optional block that contains a CCA control vector. See Table 179 on page 647. If the TR-31 key block header contains an optional block with a control vector in it, the control vector is used in place of keywords to produce the output CCA key-token. If the key usage and mode of use fields of the key block are not IBM-defined (see Table 179 on page 647), the control vector must not conflict with any TR-31 header fields.

**CV subtype extension for "C0" key usage** (one required). Only valid for TR-31 key block with key usage "C0" and no control vector in optional block.

**CVK-CVV**  
Convert a TR-31 card verification key (CVK) to a double-length CCA DES MAC key that has a subtype extension of CVVKEY-A. This restricts the key to generating or verifying a Visa CVV or MasterCard CVC.

**CVK-CSC**  
Convert a TR-31 CVK to a CCA DES MAC key that has a subtype extension of AMEX-CSC. This restricts the key to generating or verifying an American Express card security code, also known as a card identification number (CID).

**CV key type for "K0" key usage** (one required). Only valid for TR-31 key block with key usage "K0" and no control vector in optional block.

**EXPORTER**  
For TR-31 key usage "K0" and mode of use "E" or "B", convert a TR-31 key encryption or wrapping key to a CCA EXPORTER key.

**OKEYXLAT**  
For TR-31 key usage "K0" and mode of use "E" or "B", convert a TR-31 key encryption or wrapping key to a CCA OKEYXLAT key.

**IMPORTER**  
For TR-31 key usage "K0" and mode of use "D" or "B", convert a TR-31 key encryption or wrapping key to a CCA IMPORTER key.

**IKEYXLAT**  
For TR-31 key usage "K0" and mode of use "D" or "B", convert a TR-31 key encryption or wrapping key to a CCA IKEYXLAT key.
## TR31 Key Import (CSNBT31I)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CV key type for &quot;V0&quot;, &quot;V1&quot;, or &quot;V2&quot; key usage</strong> (one required). Only valid for TR-31 key block with key usage &quot;V0&quot;, &quot;V1&quot;, or &quot;V2&quot; and no control vector in optional block. When this keyword is specified, an optional CV key type modifier can be specified for key usage &quot;V0&quot; or &quot;V1&quot;.</td>
<td></td>
</tr>
<tr>
<td><strong>PINGEN</strong></td>
<td>Convert a TR-31 PIN verification key to a CCA PINGEN key.</td>
</tr>
<tr>
<td><strong>PINVER</strong></td>
<td>Convert a TR-31 PIN verification key to a CCA PINVER key.</td>
</tr>
<tr>
<td><strong>CV key usage for &quot;E0&quot; or &quot;E2&quot; key usage</strong> (one required) Only valid for TR-31 key block with key usage &quot;E0&quot; or &quot;E2&quot; and no control vector in optional block.</td>
<td></td>
</tr>
<tr>
<td><strong>DMAC</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: application cryptograms or secure messaging for integrity to CCA DKYGENKY with key usage DMAC.</td>
</tr>
<tr>
<td><strong>DMV</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: application cryptograms or secure messaging for integrity to CCA DKYGENKY with key usage DMV.</td>
</tr>
<tr>
<td><strong>CV key usage for &quot;E1&quot; key usage</strong> (one required) Only valid for TR-31 key block with key usage &quot;E1&quot; and no control vector in optional block.</td>
<td></td>
</tr>
<tr>
<td><strong>DMPIN</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: secure messaging for confidentiality to CCA DKYGENKY with key usage DMPIN.</td>
</tr>
<tr>
<td><strong>DDATA</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: secure messaging for confidentiality to CCA DKYGENKY with key usage DDATA.</td>
</tr>
<tr>
<td><strong>CV key usage for &quot;E5&quot; key usage</strong> (one required) Only valid for TR-31 key block with key usage &quot;E5&quot; and no control vector in optional block.</td>
<td></td>
</tr>
<tr>
<td><strong>DMAC</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: card personalization to CCA DKYGENKY with key usage DMAC.</td>
</tr>
<tr>
<td><strong>DMV</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: card personalization to CCA DKYGENKY with key usage DMV.</td>
</tr>
<tr>
<td><strong>DEXP</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: card personalization to CCA DKYGENKY with key usage DEXP.</td>
</tr>
<tr>
<td><strong>CV subtype for &quot;E0&quot;, &quot;E1&quot;, or &quot;E2&quot; key usage</strong> (one required). Only valid for TR-31 key block with key usage &quot;E0&quot;, &quot;E1&quot;, or &quot;E2&quot; and no control vector in optional block.</td>
<td></td>
</tr>
<tr>
<td><strong>DKYL0</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: application cryptograms, secure message for confidentiality, or secure message for integrity to CCA DKYGENKY with subtype DKYL0.</td>
</tr>
<tr>
<td><strong>DKYL1</strong></td>
<td>Convert TR-31 EMV/chip issuer master key: application cryptograms, secure message for confidentiality, or secure message for integrity to CCA DKYGENKY with subtype DKYL1.</td>
</tr>
<tr>
<td><strong>CV key type modifier for &quot;V0&quot; or &quot;V1&quot; key usage</strong> (one required). Only valid for TR-31 key block with key usage &quot;V0&quot; or &quot;V1&quot; and no control vector in optional block.</td>
<td></td>
</tr>
<tr>
<td><strong>NOOFFSET</strong></td>
<td>Convert a TR-31 PIN verification key to a CCA PINGEN or PINVER key with the key type modifier NOOFFSET, so that the key cannot participate in a PIN offset process or PVV process.</td>
</tr>
<tr>
<td><strong>Key-wrapping method</strong> (one, optional)</td>
<td></td>
</tr>
<tr>
<td><strong>USECONFIG</strong></td>
<td>Specifies to wrap the key using the configuration setting for the default wrapping method. This is the default. <strong>Note:</strong> Do not use this keyword if the default wrapping method is WRAP-ECB and a control vector is present in an optional block of the TR-31 key block with CV bit 56 = B'1' (ENH-ONLY). Use the WRAP-ENH keyword instead.</td>
</tr>
<tr>
<td><strong>WRAP-ECB</strong></td>
<td>Specifies to wrap the key using the legacy wrapping method. <strong>Note:</strong> Do not use this keyword if a control vector is present in an optional block of the TR-31 key block with CV bit 56 = B'1' (ENH-ONLY).</td>
</tr>
<tr>
<td><strong>WRAP-ENH</strong></td>
<td>Specifies to wrap the key using the enhanced wrapping method.</td>
</tr>
</tbody>
</table>

**Translation control** (optional). This keyword is valid only with key-wrapping method WRAP-ENH or with USECONFIG when the default wrapping method is WRAP-ENH. This option cannot be used on a key with a control vector valued to binary zeros.
**TR31 Key Import (CSNBT31I)**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENH-ONLY</td>
<td>Specifies to restrict the key from being wrapped with the legacy method once it has been wrapped with the enhanced method. Sets CV bit 56 = B'1' (ENH-ONLY).</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If a control vector is present in an optional block of the TR-31 key block with CV bit 56 = B'0', this keyword overrides that value in the CCA key-token. This keyword has no effect if the control vector in an optional block is all zeros.</td>
</tr>
</tbody>
</table>

Table 134 shows all valid translations for import of a TR-31 BDK base derivation key (usage "B0") to a CCA KEYGENKY key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are for translating derived unique key per transaction (DUKPT) base derivation keys.

Table 134. Import translation table for a TR-31 BDK base derivation key (usage "B0")

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;B0&quot;</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>N/A</td>
<td>KEYGENKY, double length, UKPT (CV bit 18 = B'1')</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>&quot;B0&quot;</td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td>N/A</td>
<td>KEYGENKY, double length, UKPT (CV bit 18 = B'1')</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Note:**
1. These keys are the base keys from which derived unique key per transaction (DUKPT) initial keys are derived for individual devices such as PIN pads.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value: "B0" BDK base derivation key.
3. There are no specific access-control commands for this translation because it is not ambiguous or in need of interpretation.

Table 135 shows all valid translations for import of a TR-31 CVK card verification key (usage "C0") to a CCA MAC or DATA key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are for computing or verifying (against supplied value) a card verification code with the CVV, CVC, CVC2, and CVV2 algorithm.

Table 135. Import translation table for a TR-31 CVK card verification key (usage "C0")

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
</table>

Table 135 shows all valid translations for import of a TR-31 CVK card verification key (usage "C0") to a CCA MAC or DATA key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are for computing or verifying (against supplied value) a card verification code with the CVV, CVC, CVC2, and CVV2 algorithm.

Table 135. Import translation table for a TR-31 CVK card verification key (usage "C0")

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
</table>
Table 135. Import translation table for a TR-31 CVK card verification key (usage "C0") (continued)

<table>
<thead>
<tr>
<th>&quot;C0&quot;</th>
<th>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</th>
<th>CVK-CSC</th>
<th>MAC, single or double length, AMEX-CSC (CV bits 0 - 3 = B’0100')</th>
<th>X'015B'</th>
<th>TR31 Import - Permit C0 to MAC/MACVER:AMEX-CSC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CVK-CVV</td>
<td>MAC, double length, CVVKEY-A (CV bits 0 - 3 = B’0010')</td>
<td>X'015A'</td>
<td>TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A</td>
</tr>
<tr>
<td>&quot;V&quot;</td>
<td></td>
<td>CVK-CSC</td>
<td>MACVER, single or double length, AMEX-CSC (CV bits 0 - 3 = B’0100')</td>
<td>X'015B'</td>
<td>TR31 Import - Permit C0 to MAC/MACVER:AMEX-CSC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVK-CVV</td>
<td>MACVER, double length, CVVKEY-A (CV bits 0 - 3 = B’0010')</td>
<td>X'015A'</td>
<td>TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A</td>
</tr>
</tbody>
</table>

Security considerations:
1. There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "C0" key, if one or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "C0" key cannot be imported to the lower-security CCA DATA key, it can be imported only to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.
2. The translation from TR-31 usage "C0" is controlled by rule array keywords when using the TR31_Import verb. This makes it possible to convert an exported CCA CVVKEY-A key into an AMEX-CSC key or the other way around. To prevent such a conversion, do not enable offsets X'015A' (TR31 Import - Permit C0 to MAC/MACVER:CVVKEY-A) and X'015B' (TR31 Import - Permit C0 to MAC/MACVER:AMEX-CSC) at the same time. However, if both CVVKEY-A and AMEX-CSC translation types are required, then offsets X'015A' and X'015B' must be enabled. In this case, control is up to the development, deployment, and execution of the applications themselves.

Note:
1. Card verification keys are used for computing or verifying (against supplied value) a card verification code with the CVV, CVC, CVC2, and CVV2 algorithms. In CCA, these keys correspond to keys used with two algorithms:
   - Visa CVV and MasterCard CVC codes are generated and verified using the CVV Generate and CVV Verify verbs. These verbs require a key type of DATA or MAC/MACVER with a subtype extension (CV bits 0 - 3) of ANY-MAC, single-length CVVKEY-A and single-length CVVKEY-B, and a double-length CVVKEY-A (see CVV Key Combine verb). The MAC generate and the MAC verify (CV bits 20 - 21) key usage values must be set appropriately.
   - American Express CSC codes are generated and verified using the Transaction Validation verb. This verb requires a key type of MAC or MACVER with a subtype extension of ANY-MAC or AMEX-CSC.
2. The translation from TR-31 usage "C0" to a CCA MAC/MACVER key with a subtype extension of ANY-MAC (CV bits 0 - 3 = B’0000') is not allowed.
3. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "C0" CVK card verification key
4. CCA does not have an equivalent to the TR-31 "generate only" mode of use, so a translation from TR-31 mode "G" will result in a CCA MAC key with both MAC generate and MAC verify attributes (CV bits 20 - 21 = B’11'). Note that any key that can perform a generate operation can readily verify a MAC as well.
5. The CCA representation and the TR-31 representation of CVV keys are incompatible. CCA represents the CVVKEY-A and CVVKEY-B keys as two 8-byte (single length) keys, while TR-31 represents these keys as one 16-byte key. The CVV Generate and CVV Verify verbs have support added to accept one 16-byte CVV key, using left and right key parts as A and B. Current Visa standards require this.
6. Import and export of 8-byte CVVKEY-A and CVVKEY-B MAC/MACVER keys is allowed only using the proprietary TR-31 usage+mode values ("10" and "1", respectively) to indicate encapsulation of the IBM control vector in an optional block, because the 8-byte CVVKEY-A is meaningless and useless as a TR-31 "C0" usage key of any mode.

Table 136 on page 560 shows all valid translations for import of a TR-31 data encryption key (usage "D0") to a CCA ENCIPHER, DECIPHER, CIPHER, or DATA key, along with any access control commands that must be enabled in.
the active role for that key type and control vector attributes. These keys are used for the encryption and/or decryption of data.

Table 136. Import translation table for a TR-31 data encryption key (usage "D0")

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;D0&quot;</td>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;E&quot;</td>
<td>N/A</td>
<td>ENCIPHER, single or double length</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td>N/A</td>
<td>DECIPHER, single or double length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot;</td>
<td>N/A</td>
<td>CIPHER, single or double length</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Security consideration: There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "D0" key, if one or both applicable Encipher or Decipher control vector bits are on. However, a TR-31 "D0" key cannot be imported to the lower-security CCA DATA key, it can be imported only to a CCA key type of ENCIPHER, DECIPHER, or CIPHER. This restriction eliminates the ability to export a CCA DATA key to a TR-31 key, and re-importing it back as a CCA DATA key with the capability to MAC generate and MAC verify.

Note:
1. Data encryption keys are used for the encryption and decryption of data.
2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   "D0" Data encryption
3. There are no specific access-control commands for this translation since it is not ambiguous or in need of interpretation.

Table 137 shows all valid translations for import of a TR-31 key encryption or wrapping, or key block protection key (usages "K0", "K1") to a CCA EXPORT, OKEYXLAT, IMPORTER, or IKEYXLAT key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are used only to encrypt or decrypt other keys, or as a key used to derive keys that are used for that purpose.

Table 137. Import translation table for a TR-31 key encryption or wrapping, or key block protection key (usages "K0", "K1")

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## TR31 Key Import (CSNBT31I)

### Table 137. Import translation table for a TR-31 key encryption or wrapping, or key block protection key (usage "K0", "K1") (continued)

<table>
<thead>
<tr>
<th>&quot;K0&quot;</th>
<th>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</th>
<th>&quot;E&quot;</th>
<th>TR31 Import - Permit K0:E to EXPORTER/OKEYXLAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKEYXLAT</td>
<td>OKEYXLAT, double length</td>
<td>X'015C'</td>
<td>EXPORTER, double length, EXPO on (CV bit 21 = B'1')</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>EXPORTER, double length, EXPO on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&quot;D&quot;</th>
<th>IKEYXLAT</th>
<th>IKEYXLAT, double length</th>
<th>TR31 Import - Permit K0:D to IMPORTER/IKEYXLAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORTER</td>
<td>IMPORTER, double length, IMPORT on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&quot;B&quot;</th>
<th>OKEYXLAT</th>
<th>OKEYXLAT, double length</th>
<th>TR31 Import - Permit K0:B to EXPORTER/OKEYXLAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPORTER</td>
<td>EXPORTER, double length, EXPO on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>IKEYXLAT, double length</td>
<td>X'015E'</td>
<td>IMPORTER, double length, IMPORT on (CV bit 21 = B'1')</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>IMPORTER, double length, IMPORT on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&quot;K1&quot;</th>
<th>&quot;B&quot; or &quot;C&quot;</th>
<th>&quot;E&quot;</th>
<th>TR31 Import - Permit K1:E to EXPORTER/OKEYXLAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKEYXLAT</td>
<td>OKEYXLAT, double length</td>
<td>X'0160'</td>
<td>EXPORTER, double length, EXPO on (CV bit 21 = B'1')</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>EXPORTER, double length, EXPO on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&quot;D&quot;</th>
<th>IKEYXLAT</th>
<th>IKEYXLAT, double length</th>
<th>TR31 Import - Permit K1:D to IMPORTER/IKEYXLAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORTER</td>
<td>IMPORTER, double length, IMPORT on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&quot;B&quot;</th>
<th>OKEYXLAT</th>
<th>OKEYXLAT, double length</th>
<th>TR31 Import - Permit K1:B to EXPORTER/OKEYXLAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPORTER</td>
<td>EXPORTER, double length, EXPO on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>IKEYXLAT, double length</td>
<td>X'0162'</td>
<td>IMPORTER, double length, IMPORT on (CV bit 21 = B'1')</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>IMPORTER, double length, IMPORT on (CV bit 21 = B'1')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Security considerations:

1. The CCA OKEYXLAT, EXPORTER, IKEYXLAT, or IMPORTER KEK translation to a TR-31 "K0" key with mode "B" (both wrap and unwrap) is not allowed for security reasons. Even with access-control point control, this capability would give an immediate path to turn a CCA EXPORTER key into a CCA IMPORTER, and the other way around.

2. When a TR-31 key block does not have an included control vector as an optional block, the default control vector is used to construct the output key-token. Default CCA EXPORTER or IMPORTER keys have CV bits 18 - 20 on, which are used for key generation.

### Note:

1. Key encryption or wrapping keys are used only to encrypt or decrypt other keys, or as a key used to derive keys that are used for that purpose.

2. This table defines the only supported translations for this TR-31 usage. Usage must be the following value:
   - "K0" Key encryption or wrapping
   - "K1" TR-31 key block protection key

3. Any attempt to import a TR-31 "K0" or "K1" key that has algorithm "D" (DEA) will result in an error because CCA does not support single-length KEKs.

4. CCA mode support is the same for version IDs "A", "B", and "C", because the distinction between TR-31 "K0" and "K1" does not exist in CCA keys. CCA does not distinguish between targeted protocols currently, and so there is no good way to represent the difference. Also note that most wrapping mechanisms now involve derivation or key variation steps.

---

Table 138 on page 562 shows all valid translations for import of a TR-31 ISO MAC algorithm key (usages "M0", "M1", "M3") to a CCA MAC, MACVER, DATA, DATAM, or DATAMV key, along with any access control commands.
that must be enabled in the active role for that key type and control vector attributes. These keys are use to compute or verify a code for message authentication.

Table 138. Import translation table for a TR-31 ISO MAC algorithm key (usages "M0", "M1", "M3")

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;M0&quot;</td>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td>N/A</td>
<td>MAC, double length, ANY-MAC (CV bits 0 - 3 = B'0000')</td>
<td>X'0164'</td>
<td>TR31 Import - Permit M0/M1/M3 to MAC/MACVER:ANY-MAC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;V&quot;</td>
<td></td>
<td>MACVER, double length, ANY-MAC (CV bits 0 - 3 = B'0000')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;M1&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td></td>
<td></td>
<td>MAC, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;V&quot;</td>
<td></td>
<td>MACVER, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;M3&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td></td>
<td></td>
<td>MAC, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;V&quot;</td>
<td></td>
<td>MACVER, single or double length, ANY-MAC (CV bits 0 - 3 = B'0000')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Security consideration: There is asymmetry in the translation from a CCA DATA key to a TR-31 key. The asymmetry results from CCA DATA keys having attributes of both data encryption keys and MAC keys, while TR-31 separates data encryption keys from MAC keys. A CCA DATA key can be exported to a TR-31 "M0", "M1", or "M3" key, if one or both applicable MAC generate and MAC verify control vector bits are on. However, a TR-31 "M0", "M1", or "M3" key cannot be imported to the lower-security CCA DATA key, it can be imported only to a CCA key type of MAC or MACVER. This restriction eliminates the ability to export a CCA MAC or MACVER key to a TR-31 key, and re-importing it back as a CCA DATA key with the capability to Encipher, Decipher, or both.

Note:
1. MAC keys are used to compute or verify a code for message authentication.
2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   "M0"  SO 16609 MAC algorithm 1, TDEA
   The ISO 16609 MAC algorithm 1 is based on ISO 9797. It is identical to "M1" except that it does not support 8-byte DES keys.
   "M1"  SO 9797 MAC algorithm 1
   The ISO 9797 MAC algorithm 1 is identical to "M0" except that it also supports 8-byte DES keys.
   "M3"  ISO 9797 MAC algorithm 3
   The X9.19 style of Triple-DES MAC.
3. A CCA control vector has no bits defined to limit key usage by algorithm, such as CBC MAC (TR-31 usage "M0" and "M1") or X9.19 (TR-31 usage "M3"). When importing a TR-31 key block, the resulting CCA key token deviates from the restrictions of usages "M0", "M1", and "M3". Importing a TR-31 key block which allows MAC generation ("G" or "C") results in a control vector with the ANY-MAC attribute rather than for the restricted algorithm that is set in the TR-31 key block. The ANY-MAC attribute provides the same restrictions as what CCA currently uses for generating and verifying MACs.
### Table 139. Import translation table for a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2")

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;P0&quot;</td>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;E&quot;</td>
<td>N/A</td>
<td>OPINENC, double length</td>
<td>X'0165'</td>
<td>TR31 Import - Permit Pt:E to OPINENC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;D&quot;</td>
<td></td>
<td></td>
<td></td>
<td>IPINENC, double length</td>
<td>X'0166'</td>
<td>TR31 Import - Permit Pt:D to IPINENC</td>
</tr>
</tbody>
</table>

Table 139 shows all valid translations for import of a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2") to a CCA OPINENC, IPINENC, PINGEN, or PINVER key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are used to protect PIN blocks and to generate or verify a PIN using a particular PIN-calculation method for that key type.
### Table 139. Import translation table for a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2") (continued)

<table>
<thead>
<tr>
<th>&quot;V0&quot;</th>
<th>&quot;A&quot;</th>
<th>&quot;N&quot; (requires both commands)</th>
<th>PINGEN</th>
<th>PINGEN, double length, NO-SPEC (CV bits 0 - 3 = B'0000')</th>
<th>NOOFFSET off (CV bit 37 = B'0')</th>
<th>X'0167' TR31 Import - Permit V0 to PINGEN: NO-SPEC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td>NOOFFSET off (CV bit 37 = B'0')</td>
<td>X'0167' TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot; (requires both commands)</td>
<td>PINGEN, NOOFFSET</td>
<td>NOOFFSET on (CV bit 37 = B'1')</td>
<td>X'0167' TR31 Import - Permit V0 to PINGEN: NO-SPEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td>NOOFFSET on (CV bit 37 = B'1')</td>
<td>X'0167' TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot; (requires both commands)</td>
<td>PINVER</td>
<td>NOOFFSET off (CV bit 37 = B'0')</td>
<td>X'0168' TR31 Import - Permit V0 to PINVER: NO-SPEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;V&quot;</td>
<td>NOOFFSET off (CV bit 37 = B'0')</td>
<td>X'0168' TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot; (requires both commands)</td>
<td>PINVER, NOOFFSET</td>
<td>NOOFFSET on (CV bit 37 = B'1')</td>
<td>X'0168' TR31 Import - Permit V0 to PINVER: NO-SPEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;V&quot;</td>
<td>NOOFFSET on (CV bit 37 = B'1')</td>
<td>X'0168' TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Value</td>
<td>PINGEN or PINVER</td>
<td>Description</td>
<td>Import Command</td>
<td>Import Action</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>----------------</td>
<td>-------------</td>
<td>---------------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>&quot;V1&quot;</td>
<td>&quot;A&quot;</td>
<td>PINGEN</td>
<td>PINGEN, double length, IBM PIN/IBM-PINO (CV bits 0 - 3 = B'0001')</td>
<td>X'0169'</td>
<td>TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBM-PINO</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td>NOOFFSET</td>
<td>NOOFFSET off (CV bit 37 = B'0')</td>
<td>X'0169'</td>
<td>TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot; (requires both commands)</td>
<td>PINGEN, NOOFFSET</td>
<td>NOOFFSET off (CV bit 37 = B'0')</td>
<td>X'0169'</td>
<td>TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBM-PINO</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td>NOOFFSET</td>
<td>NOOFFSET on (CV bit 37 = B'1')</td>
<td>X'0169'</td>
<td>TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot; (requires both commands)</td>
<td>PINVER</td>
<td>PINVER, double length, IBM PIN/IBM-PINO (CV bits 0 - 3 = B'0001')</td>
<td>X'016A'</td>
<td>TR31 Import - Permit V1 to PINVER:IBM-PIN/IBM-PINO</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;V&quot;</td>
<td>NOOFFSET</td>
<td>NOOFFSET off (CV bit 37 = B'0')</td>
<td>X'016A'</td>
<td>TR31 Import - Permit V1 to PINVER:IBM-PIN/IBM-PINOEC</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot; (requires both commands)</td>
<td>PINVER, NOOFFSET</td>
<td>NOOFFSET on (CV bit 37 = B'1')</td>
<td>X'016A'</td>
<td>TR31 Import - Permit V1 to PINVER:IBM-PIN/IBM-PINOC</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;V&quot;</td>
<td>NOOFFSET</td>
<td>NOOFFSET on (CV bit 37 = B'1')</td>
<td>X'016A'</td>
<td>TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
</tr>
</tbody>
</table>
**TR31 Key Import (CSNBT31I)**

Table 139. Import translation table for a TR-31 PIN encryption or PIN verification key (usages "P0", "V0", "V1", "V2") (continued)

<table>
<thead>
<tr>
<th>&quot;V2&quot;</th>
<th>&quot;A&quot;</th>
<th>&quot;N&quot; (requires both commands)</th>
<th>PINGEN</th>
<th>PINGEN, double length, VISA-PVV (CV bits 0 - 3 = B'0010')</th>
<th>X'016B' TR31 Import - Permit V2 to PINGEN:VISA-PVV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;G&quot; or &quot;C&quot;</td>
<td>PINGEN, double length, VISA-PVV (CV bits 0 - 3 = B'0010')</td>
<td>X'016B' TR31 Import - Permit V2 to PINGEN:VISA-PVV</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot; (requires both commands)</td>
<td>PINVER</td>
<td>PINVER, double length, VISA-PVV (CV bits 0 - 3 = B'0010')</td>
<td>X'016C' TR31 Import - Permit V2 to PINVER:VISA-PVV</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;, &quot;B&quot;, or &quot;C&quot;</td>
<td>&quot;V&quot;</td>
<td>PINVER</td>
<td>PINVER, double length, VISA-PVV (CV bits 0 - 3 = B'0010')</td>
<td>X'016C' TR31 Import - Permit V2 to PINVER:VISA-PVV</td>
<td></td>
</tr>
</tbody>
</table>

**Security note:** TR-31 key blocks that are protected under legacy version ID "A" (using the Key Variant Binding Method 2005 Edition) use the same mode of use "N" for PINGEN and PINVER keys. For version ID "A" keys only, for a given PIN key usage, enabling both the PINGEN and PINVER access-control points at the same time while enabling offset X'017C' (for mode "N", no special restrictions) is **NOT** recommended. In other words, for a particular PIN verification key usage, you should not simultaneously enable the four commands shown below for that usage:

**Table 140. Commands**

<table>
<thead>
<tr>
<th>Key type, mode, or version</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;V0&quot; For usage &quot;V0&quot;, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. <strong>Avoid simultaneously enabling these four commands.</strong></td>
<td>Key type PINGEN X'0167' TR31 Import - Permit V0 to PINGEN:NO-SPEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Key type PINVER X'0168' TR31 Import - Permit V0 to PINVER:NO-SPEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode &quot;N&quot; X'017C' TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version &quot;A&quot; X'0150' TR31 Import - Permit Version A TR-31 Key Blocks</td>
<td></td>
</tr>
<tr>
<td>&quot;V1&quot; - For usage &quot;V1&quot;, a user with the following four commands enabled in the active role can change a PINVER key into a PINGEN key and the other way around. <strong>Avoid simultaneously enabling these four commands.</strong></td>
<td>Key type PINGEN X'0169' TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBMPINO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Key type PINVER X'016A' TR31 Import - Permit V1 to PINVER:IBM-PIN/IBMPINO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode &quot;N&quot; X'017C' TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version &quot;A&quot; X'0150' TR31 Import - Permit Version A TR-31 Key Blocks</td>
<td></td>
</tr>
</tbody>
</table>
### Table 140. Commands (continued)

<table>
<thead>
<tr>
<th>Key type, mode, or version</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;V2&quot;</td>
<td></td>
<td>TR31 Import - Permit V2 to PINGEN:VISA-PVV</td>
</tr>
<tr>
<td>Key type PINGEN</td>
<td>X'016B'</td>
<td></td>
</tr>
<tr>
<td>Key type PINVER</td>
<td>X'016C'</td>
<td>TR31 Import - Permit V2 to PINVER:VISA-PVV</td>
</tr>
<tr>
<td>Mode &quot;N&quot;</td>
<td>X'017C'</td>
<td>TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
</tr>
<tr>
<td>Version &quot;A&quot;</td>
<td>X'0150'</td>
<td></td>
</tr>
</tbody>
</table>

Failure to comply with this recommendation allows changing PINVER keys into PINGEN and the other way around.

**Note:**

1. PIN encryption keys are used to protect PIN blocks. PIN verification keys are used to generate or verify a PIN using a particular PIN-calculation method for that key type.

2. This table defines the only supported translations for this TR-31 usage.
   - Usage must be one of the following values:
     - "P0" PIN encryption
     - "V0" PIN verification, KPV, other algorithm
       - Usage "V0" does not have its own PIN-calculation method defined. The mapping to NO-SPEC is sub-optimal. Exporting to "N" mode restricts keys from being imported with the IBM-PIN/IBM-PINO or VISA-PVV attribute, while CCA NO-SPEC allows any method.
     - "V1" PIN verification, IBM 3624
     - "V2" PIN verification, Visa PVV
       - The NOOFFSET keyword is not allowed for the Visa PVV algorithm because it does not support this attribute.

3. Mode must be one of the following values:
   - "E" Encrypt/wrap only
     - This mode restricts PIN encryption keys to encrypting a PIN block. May be used to create or reencipher an encrypted PIN block (for key-to-key translation).
   - "D" Decrypt/unwrap only
     - This mode restricts PIN encryption keys to decrypting a PIN block. Generally used in a PIN translation to decrypt the incoming PIN block.
   - "N" No special restrictions (other than restrictions implied by the key usage)
     - This mode is used by several vendors for a PIN generate or PIN verification key when the key block version ID is "A".
   - "G" Generate only
     - This mode is used for a PINGEN key that may not perform a PIN verification. The control vector will not have its EPINVER attribute on (CV bit 22 = B'0').
   - "V" Verify only
This mode is used for PIN verification only. If the TR-31 key block does not have a control vector included, the only usage bits set on in the control vector is the EPINVER bit (CV bits 18 - 22 = B'00001').

"C" Both generate and verify (combined)

This mode indicates that the control vector will have the default PINGEN bits on (CV bits 18 - 22 = B'11111').

4. Any attempt to import a TR-31 "P0" key that has mode "B" (both encrypt and decrypt) will result in an error because CCA does not support this combination of attributes.

5. If the TR-31 key block contains a control vector, and the control vector has NOOFFSET on, the NOOFFSET keyword is not necessary because the verb will automatically set NOOFFSET on in this case.

Table 141 shows all valid translations for import of a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5") to a CCA DKYGENKY, DATA, MAC, CIPHER, or ENCIPHER key, along with any access control commands that must be enabled in the active role for that key type and control vector attributes. These keys are used by the chip cards to perform cryptographic operations or, in some cases, to derive keys used to perform operations.

<table>
<thead>
<tr>
<th>Key usage</th>
<th>Key block protection method keyword (version ID)</th>
<th>Mode of use</th>
<th>Rule-array keywords</th>
<th>CCA key type and control vector attributes</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;E0&quot;</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>DKYL0, DMAC</td>
<td>DXYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')</td>
<td>X'016D'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>&quot;X&quot;</td>
<td>DXYGENKY, double length, DXYGENKY, double length, DKYL0 (CV bits 12 - 14 = B'000'), DMV (CV bits 19 - 22 = B'0011')</td>
<td>X'016E'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV</td>
<td></td>
</tr>
<tr>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
<td></td>
<td>X'016F'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC</td>
<td></td>
</tr>
</tbody>
</table>

Table 141. Import translation table for a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5")
### Table 141. Import translation table for a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5") (continued)

<table>
<thead>
<tr>
<th>Usage</th>
<th>Option 1</th>
<th>Option 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>DKYGENKY, double length, DKY0 (CV bits 12 - 14 = B'000'), DMPIN (CV bits 19 - 22 = B'1001')</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0171' TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>DKYGENKY, double length, DDATA (CV bits 19 - 22 = B'0001')</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0172' TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>DKYGENKY, double length, DKL1 (CV bits 12 - 14 = B'001'), DMPIN (CV bits 19 - 22 = B'1001')</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0173' TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>DKYGENKY, double length, DKL1 (CV bits 12 - 14 = B'001'), DDATA (CV bits 19 - 22 = B'0001')</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0174' TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>DKYGENKY, double length, DKL0 (CV bits 12 - 14 = B'000'), DMAC (CV bits 19 - 22 = B'0010')</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0175' TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>DKYGENKY, double length, DKL1 (CV bits 12 - 14 = B'001'), DMAC (CV bits 19 - 22 = B'0010')</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0176' TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
</tr>
<tr>
<td>E3</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>N/A ENCIPHER</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0177' TR31 Import - Permit E3 to ENCIPHER</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;G&quot;</td>
<td></td>
</tr>
<tr>
<td>E4</td>
<td>&quot;A&quot;</td>
<td>&quot;N&quot;</td>
<td>N/A DKYGENKY, double length, DKL0 (CV bits 12 - 14 = B'000'), DDATA (CV bits 19 - 22 = B'0001')</td>
</tr>
<tr>
<td></td>
<td>&quot;D&quot;</td>
<td></td>
<td>X'0178' TR31 Import - Permit E4 to DKYGENKY:DKYL0+DDATA</td>
</tr>
<tr>
<td></td>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
</tr>
</tbody>
</table>
### TR31 Key Import (CSNBT31I)

**Table 141. Import translation table for a TR-31 EMV/chip issuer master-key key (usages "E0", "E1", "E2", "E3", "E4", "E5") (continued)**

<table>
<thead>
<tr>
<th>Usage (E0, E1, E2, E3, E4, E5)</th>
<th>Key Type (G)</th>
<th>Key Length</th>
<th>Key Details</th>
<th>Key Import</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;E5&quot;</td>
<td>&quot;A&quot;</td>
<td>&quot;G&quot;</td>
<td>DKYL0, DMAC</td>
<td>X'0179'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DMAC</td>
</tr>
<tr>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;G&quot;</td>
<td>&quot;C&quot;</td>
<td>DKYL0, DDATA</td>
<td>X'017A'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA</td>
</tr>
<tr>
<td>&quot;B&quot; or &quot;C&quot;</td>
<td>&quot;X&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

1. EMV/chip issuer master-key keys are used by the chip cards to perform cryptographic operations or, in some cases, to derive keys used to perform operations. In CCA, these keys are (a) diversified key-generating keys (key type DKYGENKY), allowing derivation of operational keys, or (b) operational keys. Note that in this context, "master key" has a different meaning than for CCA. These master keys, also called KMCs, are described by EMV as DES master keys for personalization session keys. They are used to derive the corresponding chip card master keys, and not typically used directly for cryptographic operations other than key derivation. In CCA, these keys are usually key generating keys with derivation level DKYL1 (CV bits 12 - 14 = B'001'), used to derive other key generating keys (the chip card master keys). For some cases, or for older EMV key derivation methods, the issuer master keys could be level DKYL0 (CV bits 12 - 14 = B'000').

2. This table defines the only supported translations for this TR-31 usage. Usage must be one of the following values:
   - "E0" Application cryptograms
   - "E1" Secure messaging for confidentiality
   - "E2" Secure messaging for integrity
   - "E3" Data authentication code
   - "E4" Dynamic numbers
   - "E5" Card personalization

3. EMV support in CCA is different than TR-31 support, and CCA key types do not match TR-31 types.

4. DKYGENKY keys are double length only.

5. In CCA, a MAC key that can perform a MAC generate operation also can perform a MAC verify. For TR-31 mode "G" (generate only), the translation to a CCA key results in a key that can perform MAC generate and MAC verify.
TR31 Key Import (CSNBT31I)

**tr31_key_block_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the `tr31_key_block` variable. The length field in the TR-31 block is a 4-digit decimal number, so the maximum acceptable length is 9992 bytes. For more information, see “TR31 Key Token Parse (CSNBT31P)” on page 578.

**tr31_key_block**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing the TR-31 key block that is to be imported. The key block is protected with the key identified by the `unwrap_kek_identifier` parameter.

**unwrap_kek_identifier_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the `unwrap_kek_identifier` variable. Set this value to 64.

**unwrap_kek_identifier**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing the operational fixed-length DES key-token used to unwrap the key identified by the `tr31_key_block` parameter, or a key label of such a key in DES key-storage. The key must have a key type of IMPORTER or IKEYXLAT, and be authorized for import.

**Note:** DES keys wrapped in ECB mode (CCA legacy wrap mode) cannot be used to wrap or unwrap TR-31 "B" or "C" key blocks that have or will have "E" exportability, because ECB-mode does not comply with ANSI X9.24 Part 1.

**wrap_kek_identifier_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the length in bytes of the `wrap_kek_identifier` variable. The value must be greater than or equal to 0. A null key-token can have a length of 1. Set this value to 64 for a key label or a KEK.

**wrap_kek_identifier**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing the operational fixed-length DES key-token used to wrap the key identified by the `output_key_identifier` parameter, a null key-token, or a key label of such a key in DES key-storage. If the key token identified by the parameter is not null, it must have a key type of IMPORTER or IKEYXLAT, be authorized for import, and have the same clear key as the key identified by the `unwrap_kek_identifier` parameter. If the
parameter identifies a null key-token, then the `unwrap_kek_identifier` parameter is also used for wrapping the CCA output key token.

**output_key_identifier_length**

**Direction:** Input/Output  
**Type:** Integer  

A pointer to an integer specifying the length in bytes of the `output_key_identifier` variable. This is an input/output parameter.

**output_key_identifier**

**Direction:** Input/Output  
**Type:** String  

A pointer to a string variable containing the key token or the key label for the token that is to receive the imported key. The output key-token is a CCA internal or external key token containing the key received in the TR-31 token. If a key token is provided, it must be a null token (64 bytes of X'00'). If a key label is provided, the imported token is stored in the key storage file and identified by that label.

**num_opt_blocks**

**Direction:** Output  
**Type:** Integer  

A pointer to an integer variable where the verb stores the number of optional blocks that are present in the TR-31 key token.

**cv_source**

**Direction:** Output  
**Type:** Integer  

A pointer to an integer variable where the verb stores a value indicating how the control vector in the output key token was created. It can be one of the values in Table 142.

**Table 142. TR31 Key Import CV sources**

<table>
<thead>
<tr>
<th>CSNBT31I CV source</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No CV was present in an optional block, and the output CV was created by the verb based on input parameters and on the attributes in the TR-31 key block header.</td>
</tr>
<tr>
<td>1</td>
<td>A CV was obtained from an optional block in the TR-31 key block, and the key usage and mode of use were also specified in the TR-31 header. The verb verified compatibility of the header values with the CV and then used that CV in the output key token.</td>
</tr>
<tr>
<td>2</td>
<td>A CV was obtained from an optional block in the TR-31 key block, and the key usage and mode of use in the TR-31 header held the proprietary values indicating that key use and mode should be obtained from the included CV. The CV from the TR-31 token was used as the CV for the output key token.</td>
</tr>
</tbody>
</table>

Any values other than these three are reserved and are currently invalid.

**protection_method**

**Direction:** Output  
**Type:** Integer
A pointer to an integer variable where the verb stores a value indicating what method was used to protect the input TR-31 key block. The TR-31 standard allows two methods, and the application program might want to know which was used for security purposes. The variable can have one of the values in Table 143.

<table>
<thead>
<tr>
<th>CSNBT31I protection method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The TR-31 key block was protected using the variant method as identified by a Key Block Version ID value of &quot;A&quot; (X'41').</td>
</tr>
<tr>
<td>1</td>
<td>The TR-31 key block was protected using the derived key method as identified by a Key Block Version ID value of &quot;B&quot; (X'42').</td>
</tr>
<tr>
<td>2</td>
<td>The TR-31 key block was protected using the variant method as identified by a Key Block Version ID value of &quot;C&quot; (X'43'). Functionally this method is the same as &quot;A&quot;, but to maintain consistency a different value is returned here for &quot;C&quot;.</td>
</tr>
</tbody>
</table>

Any values other than these three are reserved and are currently invalid.

**Restrictions**

The restrictions for CSNBT31I.

None.

**Required commands**

The required commands for CSNBT31I.

The TR31 Key Import verb requires the following commands to be enabled in the active role:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAP-ENH or WRAP-ECB</td>
<td>X'0153'</td>
<td>TR31 Import - Permit Override of Default Wrapping Method</td>
</tr>
<tr>
<td>TR-31 key block version ID</td>
<td>Offset</td>
<td>Command</td>
</tr>
<tr>
<td>&quot;A&quot; (X'41')</td>
<td>X'0150'</td>
<td>TR31 Import - Permit Version A TR-31 Key Blocks</td>
</tr>
<tr>
<td>&quot;B&quot; (X'42')</td>
<td>X'0151'</td>
<td>TR31 Import - Permit Version B TR-31 Key Blocks</td>
</tr>
<tr>
<td>&quot;C&quot; (X'43')</td>
<td>X'0152'</td>
<td>TR31 Import - Permit Version C TR-31 Key Blocks</td>
</tr>
</tbody>
</table>

Be aware of access-control point X'017C' (TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER) for import of PINGEN or PINVER keys to wrapping method "A", usage "V0", "V1", or "V2", and mode "N". TR-31 key blocks with legacy key usage "A" (key block protected using the Key Variant Binding Method 2005 Edition) use the same mode "N" for PINGEN as well as PINVER keys. For usage "A" keys only, enabling a PINGEN and PINVER access-control point while
enabling offset X'017C' (for mode "N") is NOT recommended. Failure to comply with this recommendation allows changing PINVER keys into PINGEN, and the other way around.

In addition to the above commands, the verb requires these additional commands to be enabled in the active role depending on the rule-array keyword provided:

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
<th>Specific key usage, version ID, and mode values</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C0&quot;: TR-31 CVK card verification keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVK-CSC</td>
<td>X'015B'</td>
<td>TR31 Import - Permit C0 to MAC/MACVER:AMEXCSC</td>
<td>See Table 135 on page 558</td>
</tr>
<tr>
<td>CVK-CVV</td>
<td>X'015A'</td>
<td>TR31 Import - Permit C0 to MAC/MACVER:CVKEY-A</td>
<td></td>
</tr>
<tr>
<td>&quot;K0&quot; and &quot;K1&quot;: TR-31 key encryption or wrapping, or key block protection keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OKEYXLAT or EXPORTER</td>
<td>X'015C'</td>
<td>TR31 Import - Permit K0:E to EXPORTER/OKEYXLAT</td>
<td>See Table 137 on page 560</td>
</tr>
<tr>
<td></td>
<td>X'015E'</td>
<td>TR31 Import - Permit K0:B to EXPORTER/OKEYXLAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0160'</td>
<td>TR31 Import - Permit K1:E to EXPORTER/OKEYXLAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0162'</td>
<td>TR31 Import - Permit K1:B to EXPORTER/OKEYXLAT</td>
<td></td>
</tr>
<tr>
<td>IKEYXLAT or IMPORTER</td>
<td>X'015D'</td>
<td>TR31 Import - Permit K0:D to IMPORTER/IKEYXLAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'015F'</td>
<td>TR31 Import - Permit K0:B to IMPORTER/IKEYXLAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0161'</td>
<td>TR31 Import - Permit K1:D to IMPORTER/IKEYXLAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0163'</td>
<td>TR31 Import - Permit K1:B to IMPORTER/IKEYXLAT</td>
<td></td>
</tr>
<tr>
<td>&quot;M0&quot;, &quot;M1&quot;, and &quot;M3&quot;: TR-31 ISO MAC algorithm keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>X'0164'</td>
<td>TR31 Import - Permit M0/M1/M3 to MAC/MACVER:ANY-MAC</td>
<td>See Table 138 on page 562</td>
</tr>
<tr>
<td></td>
<td>X'018C'</td>
<td>TR31 Export - Permit MACVER/DATAMV to M0:V</td>
<td></td>
</tr>
</tbody>
</table>
# TR31 Key Import (CSNBT31I)

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
<th>Specific key usage, version ID, and mode values</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>X'0165'</td>
<td>TR31 Import - Permit P0:E to OPINENC</td>
<td>See Table 139 on page 563</td>
</tr>
<tr>
<td></td>
<td>X'0166'</td>
<td>TR31 Import - Permit P0:D to IPINENC</td>
<td></td>
</tr>
<tr>
<td>PINGEN</td>
<td>X'0167'</td>
<td>TR31 Import - Permit V0 to PINGEN:NO-SPEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0169'</td>
<td>TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBMPINO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'016B'</td>
<td>TR31 Import - Permit V2 to PINGEN:VISA-PVV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'017C'</td>
<td>TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
</tr>
<tr>
<td>PINVER</td>
<td>X'0168'</td>
<td>TR31 Import - Permit V0 to PINVER:NO-SPEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'016A'</td>
<td>TR31 Import - Permit V1 to PINVER:IBM-PIN/IBMPINO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'016C'</td>
<td>TR31 Import - Permit V2 to PINVER:VISA-PVV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'017C'</td>
<td>TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td></td>
</tr>
</tbody>
</table>

"E0", "E1", "E2", "E3", "E4", and "E5": TR-31 EMC/chip issuer master-key keys

<table>
<thead>
<tr>
<th>DKYL0</th>
<th>X'016D'</th>
<th>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC</th>
<th>See Table 141 on page 568</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X'016E'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0171'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0172'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0175'</td>
<td>TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0179'</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DALL to E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'017A'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'017B'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DEXP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DKYL1</th>
<th>X'016F'</th>
<th>TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC</th>
<th>See Table 141 on page 568</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X'0170'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0173'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0174'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'0176'</td>
<td>TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC</td>
<td></td>
</tr>
</tbody>
</table>
### Rule-array keyword Offset Command Specific key usage, version ID, and mode values

<table>
<thead>
<tr>
<th>Rule-array keyword</th>
<th>Offset</th>
<th>Command</th>
<th>See Table 141 on page 568</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMAC X'016D'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'016F'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0175'</td>
<td>TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0176'</td>
<td>TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0179'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMV X'016E'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0170'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMPIN X'0171'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0173'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDATA X'0172'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0174'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'017A'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEXP X'017B'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DEXP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N/A X'0177'</td>
<td>TR31 Import - Permit E3 to ENCIPHER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0178'</td>
<td>TR31 Import - Permit E4 to DKYGENKY:DKYL0+DDATA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNBT31IJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBT31IJ are shown here.

### Format

```java
public native void CSNBT31IJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger tr31_key_block_length,
    byte[] tr31_key_block,
    hikmNativeInteger unwrap_kek_identifier_length,
    byte[] unwrap_kek_identifier,
    hikmNativeInteger wrap_kek_identifier_length,
```
TR31 Key Import (CSNB31I)

byte[] wrapkek_identifier,
hkmNativeInteger output_key_identifier_length,
byte[] output_key_identifier,
hkmNativeInteger num_opt_blocks,
hkmNativeInteger cv_source,
hkmNativeInteger protection_method);
Use the TR31 Key Token Parse verb to disassemble the unencrypted header of an external TR-31 key block into separate pieces of information.

The part of the header that is optional, called optional blocks, is not disassembled. To obtain the contents of optional blocks, use the TR31 Optional Data Read verb. Neither verb performs any cryptographic services, and both disassemble a key block in application storage. The validity of the key block is verified as much as can be done without performing any cryptographic services.

The TR-31 header fields that are disassembled into separate pieces of information include a key block version ID, key block length, key usage, algorithm, mode of use, key version number, exportability, and number of optional blocks. Except for the two length values, which are returned as integers, the verb returns the field values as ASCII strings. This format is used in the TR-31 key block itself. For more information, see X9 TR-31 2010: Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms.

The following table summarizes the key blocks fields returned by this verb:

<table>
<thead>
<tr>
<th>TR-31 field name</th>
<th>Verb parameter</th>
<th>Field or buffer string length in bytes</th>
<th>Description of TR-31 field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key block version ID</td>
<td>key_block_version</td>
<td>1</td>
<td>Identifies the method by which the key block is cryptographically protected and the content layout of the block.</td>
</tr>
<tr>
<td>Key block length</td>
<td>key_block_length</td>
<td>4 (integer)</td>
<td>Entire key block length after encoding (header, encrypted confidential data, and MAC).</td>
</tr>
<tr>
<td>Key usage</td>
<td>key_usage</td>
<td>2</td>
<td>Provides information about the intended function of the protected key/sensitive data, such as data encryption, PIN encryption, or key wrapping. Numeric values are reserved for proprietary use (that is, not defined by TR-31).</td>
</tr>
<tr>
<td>Algorithm</td>
<td>algorithm</td>
<td>1</td>
<td>The approved symmetric algorithm for which the protected key may be used. Numeric values are reserved for proprietary use.</td>
</tr>
<tr>
<td>Mode of use</td>
<td>mode</td>
<td>1</td>
<td>Defines the operation for which the protected key can perform. Numeric values are reserved for proprietary use.</td>
</tr>
<tr>
<td>Key version number</td>
<td>key_version_number</td>
<td>2</td>
<td>Version number to optionally indicate that the contents of the key block is a component (key part), or to prevent re-injection of an old key. This field is a tool for enforcement of local key change rules.</td>
</tr>
<tr>
<td>Exportability</td>
<td>exportability</td>
<td>1</td>
<td>Defines whether the protected key may be exported.</td>
</tr>
<tr>
<td>Number of optional blocks</td>
<td>num_opt_blocks</td>
<td>4 (integer)</td>
<td>Defines the number of optional blocks included in the key block. If this value is greater than zero, use the TR31 Optional Data Read verb to obtain the contents of the optional blocks.</td>
</tr>
</tbody>
</table>

This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.
TR31 Key Token Parse (CSNBT31P)

Format
The format of CSNBT31P.

```
CSNBT31P(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  rule_array_count,
  rule_array,
  tr31_key_length,
  tr31_key,
  key_block_version,
  key_block_length,
  key_usage,
  algorithm,
  mode,
  key_version_number,
  exportability,
  num_opt_blocks)
```

Parameters
The parameters for CSNBT31P.

For the definitions of the `return_code`, `reason_code`, `exit_data_length`, and `exit_data` parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**
- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. The value must be 0.

**rule_array**
- **Direction:** Input
- **Type:** String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. No rule array keywords are currently defined for this verb.

**tr31_key_length**
- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the `tr31_key` variable. Specify a length that is greater than or equal to the size of the key block. The verb determines the actual length of the key by parsing its contents.

**tr31_key**
- **Direction:** Input
- **Type:** String

A pointer to a string variable containing the TR-31 key block to be disassembled.

**key_block_version**
TR31 Key Token Parse (CSNBT31P)

Direction: Output
Type: String

A pointer to a string variable. The verb copies the one byte found in the key block version ID field of the input key block to this variable.

Note that if the verb finds a proprietary key block version ID, the verb treats it as an invalid value, because the verb is not capable of disassembling a key block that has a proprietary ID. This variable is not updated if a processing error occurs.

key_block_length

Direction: Output
Type: Integer

A pointer to an integer variable. The verb parses the 2-byte numeric ASCII key block length field from the input key block, converts the string value into an integer, and returns the integer in this variable. This value must be less than or equal to the tr31_key_length input variable.

key_usage

Direction: Output
Type: String

A pointer to a string variable. The verb copies the two bytes found in the key usage field of the input key block to this variable.

algorithm

Direction: Output
Type: String

A pointer to a string variable. The verb copies the one byte found in the algorithm field of the input key block to this variable. The verb does not treat a proprietary algorithm value as an error.

mode

Direction: Output
Type: String

A pointer to a one-byte string variable containing the TR-31 mode of use for the key contained in the block. The value is obtained from the TR-31 header. The mode of use describes what operations the key can perform, within the limitations specified with the key usage value. For example, a key with usage for data encryption can have a mode to indicate that it can be used only for encryption, decryption, or both.

This pointer must be non-NULL and point to application storage with at least the size given by the byte count noted. The storage is updated with the noted value on a successful return from this verb, and unchanged otherwise.

key_version_number

Direction: Output
Type: String

A pointer to a 2-byte string variable obtained from the TR-31 header, which can be used for one of three purposes, or can be unused.

- If both bytes are X'30' ("0"), then key versioning is unused for this key. In this case, the second byte is not examined and can contain any value.
TR31 Key Token Parse (CSNB31P)

- If the first byte is X'63' ("c"), then the block contains a component of a key which must be combined with other components in order to form the complete key. TR-31 does not define the method through which the components are combined. TR-31 specifies that local rules are used for that purpose.
  
  In this case, the second byte is not examined and can contain any value.

- If the first byte is anything other than the two values above, then the 2-byte key version value is an identifier of the version of the key that is carried in the block. This key version value can be used by an application, for example, to ensure that an old version of a key is not reentered into the system.

This pointer must be non-NULL and point to application storage with at least the size given by the byte count noted. The storage is updated with the noted value on a successful return from this verb, and unchanged otherwise.

exportability

**Direction:** Output  
**Type:** String

A pointer to a one-byte string variable containing the key exportability value from the TR-31 header. This value indicates whether the key can be exported from this system, and if so specifies conditions under which export is permitted. The following three values are possible:

- If the value is X'4E' ("N"), then the key is not exportable.
- If the value is X'53' ("S"), then the key is exportable under any key-encrypting key.
- If the value is X'45' ("E"), then the key is exportable only under a trusted key-encrypting key. TR 31 defines such a trusted key as either one that is encrypted under the HSM master key or one that is itself contained in a TR-31 key block. CCA does not support KEKs that are wrapped in TR-31 key blocks.

This pointer must be non-NULL and point to application storage with at least the size given by the byte count noted. The storage is updated with the noted value on a successful return from this verb, and unchanged otherwise.

num_opt_block

**Direction:** Output  
**Type:** Integer

A pointer to an integer value containing the number of optional blocks that are part of the TR-31 key block. Information about each optional block can be obtained using the TR31 Optional Data Read verb. In this verb, use the number of optional blocks acquired with this verb to obtain a list of the IDs and lengths for each optional block. Then, use those lists to read the data from each desired block.

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNB31PJ.

See "Building Java applications to use with the CCA JNI" on page 26.

The parameters for CSNB31PJ are shown here.
TR31 Key Token Parse (CSNBT31P)

Format

```java
public native void CSNBT31PJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger tr31_key_length,
    byte[] tr31_key,
    byte[] key_block_version,
    hikmNativeInteger key_block_length,
    byte[] key_usage,
    byte[] algorithm,
    byte[] mode,
    byte[] key_version_number,
    byte[] exportability,
    hikmNativeInteger num_opt_blocks);
```
Use the TR31 Optional Data Build verb to build a properly formatted TR-31 optional block from the data provided.

The newly constructed optional block can optionally be appended to an existing structure of optional blocks, or it can be returned as a new optional blocks structure. After the last optional block has been constructed, the completed structure containing the optional blocks can be included in a TR-31 key block during an export operation by the Key Export to TR31 verb by using its opt_blocks parameter. For information about TR-31, including the format of a TR-31 optional block, see X9 TR-31 2010: Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms.

The TR-31 key block has an unencrypted header that can contain optional blocks. The header is securely bound to the key block using the integrated MAC. An optional block has a 2-byte ASCII block ID value that determines the use of the block. The ID of each block in an optional blocks structure must be unique.

The verb builds a structure of optional blocks by adding one optional block with each call. This process is repeated until the entire set of optional blocks has been added. For each call, provide the components for a single optional block. This includes the optional block ID, the optional block length in bytes, and the optional block data. In addition, provide an optional blocks buffer large enough to add the optional block being built.

There are two valid scenarios for the optional blocks buffer provided on input, as determined by the value of the opt_blocks_length variable:
1. The optional blocks buffer is empty. In this case, the newly constructed optional block is copied into the buffer.
2. The optional blocks buffer contains one or more existing optional blocks. In this case, the newly constructed optional block is appended to the existing optional blocks. No duplicate IDs are allowed.

Upon successful completion, the opt_blocks_length variable is updated to the length of the returned optional blocks structure.

This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

**Format**

The format of CSNBT31O.

```c
CSNBT310(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    opt_blocks_bfr_length,
    opt_blocks_length,
    opt_blocks,
    num_opt_blocks,
    opt_block_id,
    opt_block_data_length,
    opt_block_data)
```
Parameters

The parameters for CSNBT31O.

For the definitions of the return_code, reason_code, exit_data_length, and exit_data parameters, see "Parameters common to all verbs" on page 20.

rule_array_count
  Direction: Input
  Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 0, because no keywords are currently defined for this verb.

rule_array
  Direction: Input
  Type: String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. No rule_array keywords are currently defined for this verb.

opt_blocks_bfr_length
  Direction: Input
  Type: Integer

A pointer to an integer variable containing the length in bytes of the buffer allocated for the opt_blocks variable. Set this length to at least the size of any optional blocks structure in the buffer plus the optional block being added.

opt_blocks_length
  Direction: Input/Output
  Type: Integer

A pointer to an integer variable containing the length in bytes of the data in the opt_blocks variable. This length must be less than or equal to the value of the opt_blocks_bfr_length variable. On input, set this variable to the length of the optional blocks structure being updated. Set this value to zero if it is the first optional block in the structure. On successful completion, this variable is updated with the length of the updated variable.

opt_blocks
  Direction: Input/Output
  Type: String

A pointer to a string variable containing a buffer for the optional blocks structure that the verb updates. In the first call to the verb, the buffer will generally be empty. The verb appends one optional block to the buffer with each call.

num_opt_blocks
  Direction: Output
  Type: Integer

The num_opt_blocks parameter is a pointer to an integer variable containing the number of optional blocks contained in the opt_blocks variable that is returned by the verb.
TR31 Optional Data Build (CSNBT31O)

**opt_block_id**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing a 2-byte value that identifies the use of the optional block. Each ID must be unique, that is, no duplicates are allowed.

Note that a value of “PB” is not allowed. The Key Export to TR31 verb adds a padding block of the appropriate size as needed. Unlike the padding in the encryption key portion of the TR-31 key block, the padding block for optional blocks serves no security purpose.

**opt_block_data_length**

**Direction:** Input  
**Type:** Integer

A pointer to an integer variable containing the length in bytes of the data passed in the **opt_block_data** variable. Note that it is valid for this length to be zero, since an optional block can have an ID and a length, but no data.

**opt_block_data**

**Direction:** Input  
**Type:** String

A pointer to a string variable containing the data for the optional block that is to be constructed.

**Restrictions**

The restrictions for CSNBT31O.

An optional block with an ID of “PB” (padding block) cannot be added by the user. The Key Export to TR31 verb adds a padding block of the appropriate size as needed when building the TR-31 key block. Unlike the padding within the encrypted key portion of the key block, the padding block for optional blocks serves no security purpose.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBT31OJ.

See “Building Java applications to use with the CCA JNI” on page 26.

The parameters for CSNBT31OJ are shown here.

**Format**

```java
public native void CSNBT31OJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger opt_blocks_bfr_length,
    hikmNativeInteger opt_blocks_length,
    byte[] opt_blocks,
    hikmNativeInteger num_opt_blocks,
)
```
TR31 Optional Data Build (CSNBT31O)

byte[] opt_block_id,
hkmNativeInteger opt_block_data_length,
byte[] opt_block_data);
TR31 Optional Data Read (CSNBT31R)

Use the TR31 Optional Data Read verb to either obtain information about all of the optional blocks in the header of an external TR-31 key block, or obtain the length and data of the specified optional block.

To disassemble the part of the header that is not optional, use the TR31 Key Token Parse verb. Neither verb performs any cryptographic services, and both disassemble a key block in application storage. The validity of the key block is verified as much as can be done without performing any cryptographic services.

A TR-31 key block contains an unencrypted header that can include one or more optional blocks. All parts of the header are securely bound to the key block using the integrated MAC.

Optional blocks in a key block must each be identified by a unique 2-byte ID. The value of an ID must either be defined by TR-31 or be a numeric value, otherwise the key block is invalid. Numeric IDs are reserved for proprietary use. For more information, see X9 TR-31 2010: Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms.

In order to obtain the data of a particular optional block from the header of an external TR-31 key block, perform the following steps:

1. Use the tr31_key parameter to identify the TR-31 key block that this verb is to process.
2. Call the TR31 Key Token Parse verb to parse the TR-31 key block. See "TR31 Key Token Parse (CSNBT31P)" on page 578. Upon successful completion:
   • Set the value of the tr31_key_length variable to the value returned in the tr31_key_length variable.
   • Set the value of the num_opt_blocks variable to the value returned in the num_opt_blocks variable.
   • Allocate a string buffer in bytes for the opt_blocks_id and an integer buffer in bytes for the opt_blocks_length variables. These buffers must be at least two times the value of the num_opt_blocks variable.
3. Specify a rule-array keyword of INFO to obtain information about the optional blocks in the key block. The opt_block_id, opt_block_data_length, and opt_block_data parameters are ignored.
4. Call the TR31 Optional Data Read verb to read data from the TR-31 key block. Upon successful completion, the verb returns an array of optional block IDs in the opt_blocks_id variable, and an array of lengths for the optional block IDs in the opt_blocks_length variable. The IDs and lengths are returned in same order as the optional blocks appear in the header of the TR-31 key block.
5. Determine which ID of the unique IDs contained in the opt_blocks_id variable is to be obtained from the TR-31 key block. Set the opt_block_id variable to this 2-byte value. Set the value of the opt_block_data_length variable to the corresponding length from the opt_blocks_length variable.
   
   **Note:** The offset used to locate the ID in the opt_blocks_id variable has the same value as the offset for the corresponding length in the opt_blocks_length variable.
6. Allocate a buffer in bytes for the opt_block_data variable that is at least the value of the opt_block_data_length variable.
TR31 Optional Data Read (CSNBT31R)

7. Specify a rule-array keyword of **DATA** to obtain the length and data of the specified optional block. The *num_opt_blocks* and the *opt_blocks_id* parameters are ignored.

8. Call the TR31 Optional Data Read verb. Upon successful completion, the verb returns the data of the specified optional block in the *opt_block_data* variable. The verb updates the *opt_block_data_length* variable to the number of bytes returned in the *opt_block_data* variable.

This verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

**Format**

The format of CSNBT31R.

```
CSNBT31R(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    tr31_key_length,
    tr31_key,
    opt_block_id,
    num_opt_blocks,
    opt_block_ids,
    opt_block_lengths,
    opt_block_data_length,
    opt_block_data)
```

**Parameters**

The parameters for CSNBT31R.

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters common to all verbs" on page 20.

**rule_array_count**

- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 1.

**rule_array**

- **Direction:** Input
- **Type:** String array

A pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords for this verb are shown below:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong> (one required)</td>
<td></td>
</tr>
<tr>
<td>INFO</td>
<td>Return information about the optional blocks in the TR-31 key block.</td>
</tr>
<tr>
<td>DATA</td>
<td>Return the data contained in a specified optional block in the TR-31 key block.</td>
</tr>
</tbody>
</table>
TR31 Key Token Parse (CSNBT31R)

**tr31_key_length**
- **Direction:** Input
- **Type:** Integer

A pointer to an integer variable containing the number of bytes of data in the `tr31_key` variable. Specify a length that is greater than or equal to the size of the key block. The verb determines the actual length of the key by parsing its contents.

**tr31_key**
- **Direction:** Input
- **Type:** String

A pointer to a string variable containing the TR-31 key block to be disassembled.

**opt_block_id**
- **Direction:** Input
- **Type:** String

This parameter is used when operation keyword **DATA** is specified, otherwise it is ignored. For keyword **DATA**, this parameter is a pointer to a string variable that identifies the 2-byte ID of the optional block to obtain from the TR-31 key block.

**num_opt_blocks**
- **Direction:** Input
- **Type:** Integer

This parameter is used when operation keyword **INFO** is specified, otherwise it is ignored. For keyword **INFO**, this parameter is a pointer to an integer variable that specifies the number of 2-byte optional block IDs that are allocated for (1) the `opt_block_ids` variable and (2) the number of 2-byte integers that are allocated for the `opt_block_lengths` variable. This the value must specify the exact number of optional blocks that are in the header of the TR-31 key block. Use the TR31 Key Token Parse verb to determine the number of optional blocks IDs in a TR-31 key block before calling this verb.

**opt_block_ids**
- **Direction:** Output
- **Type:** String array

This parameter is used when operation keyword **INFO** is specified, otherwise it is ignored. For keyword **INFO**, this parameter is a pointer to a string array of 2-byte values that lists the identifiers of each optional block contained in the header of the TR-31 key block. Each ID must be unique, that is, no duplicates are allowed. The IDs, along with the associated lengths listed in the `opt_block_lengths` variable, are returned in the order that the optional blocks appear in the header of the TR-31 key block. The size of the variable must be at least two times the value of the `num_opts_blocks` variable.

**opt_block_lengths**
- **Direction:** Output
- **Type:** String array

This parameter is used when operation keyword **INFO** is specified, otherwise it is ignored. For keyword **INFO**, this parameter is a pointer to an integer array of 2-byte values that are 16-bit unsigned integers corresponding to the
**TR31 Optional Data Read (CSNBT31R)**

associated length of the optional block identified in the `opt_block_ids` variable. The lengths, along with the associated IDs listed in the `opt_block_ids` variable, are returned in the order that the optional blocks appear in the header of the TR-31 key block. The size of the variable must be at least two times the value of the `num_opt_blocks` variable.

**opt_block_data_length**

*Direction: Input/Output*  
*Type: Integer*

This parameter is used when operation keyword DATA is specified, otherwise it is ignored. For keyword DATA, this parameter is a pointer to an integer variable containing the length of the `opt_block_data` parameter. On input, this variable specifies the maximum permissible length of the result. On output, the verb updates the value to length of the returned optional block data.

**opt_block_data**

*Direction: Output*  
*Type: String*

This parameter is used when operation keyword DATA is specified, otherwise it is ignored. For keyword DATA, this parameter is a pointer to a string variable. If the TR-31 key block is found to be valid and the TR-31 key block contains an optional block specified by the `optional_block_ID` variable, the optional block is copied into this variable if it is large enough. The `opt_block_data_length` variable is updated with the length of the data returned in the variable.

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBT31RJ.

See [“Building Java applications to use with the CCA JNI” on page 26.](#)

The parameters for CSNBT31RJ are shown here.

**Format**

```java
public native void CSNBT31RJ(  
    hkmNativeInteger return_code,  
    hkmNativeInteger reason_code,  
    hkmNativeInteger exit_data_length,  
    byte[] exit_data,  
    hkmNativeInteger rule_array_count,  
    byte[] rule_array,  
    hkmNativeInteger tr31_key_length,  
    byte[] tr31_key,  
    byte[] opt_block_id,  
    hkmNativeInteger num_opt_blocks,  
    byte[] opt_blocks,  
    byte[] opt_block_ids,  
    byte[] opt_block_lengths,  
    hkmNativeInteger opt_block_data_length,  
    byte[] opt_block_data);
```
Part 3. Appendixes
Appendix A. Return codes and reason codes

This appendix describes the return codes and reason codes reported at the conclusion of verb processing.

Reason code numbers narrow down the meaning of a return code. All reason code numbers are unique and associated with a single return code. Generally, you can base your application program design on the return codes.

Each verb supplies a return code and a reason code in the variables identified by the `return_code` and `reason_code` parameters. See "Parameters common to all verbs" on page 20.

Return codes

A return code provides a general indication of the results of verb processing.

A return code can have the values shown in Table 144.

Table 144. Return code values

<table>
<thead>
<tr>
<th>Hex value</th>
<th>Decimal value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>This return code indicates a normal completion of verb processing. To provide additional information, there are also nonzero reason codes associated with this return code.</td>
</tr>
<tr>
<td>04</td>
<td>04</td>
<td>This return code is a warning indicating the verb completed processing; however, an unusual event occurred. The event is most likely related to a problem created by the user or is a normal occurrence based on the data supplied to the verb.</td>
</tr>
<tr>
<td>08</td>
<td>08</td>
<td>This return code indicates the verb prematurely stopped processing. Generally, the application programmer needs to investigate the significance of the associated reason code to determine the origin of the problem. In some cases, due to transient conditions, retrying the verb might produce different results.</td>
</tr>
<tr>
<td>0C</td>
<td>12</td>
<td>This return code indicates the verb prematurely stopped processing. Either a coprocessor is not available or a processing error occurred. The reason is most likely related to a problem in the set up of the hardware or in the configuration of the software.</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>This return code indicates the verb prematurely stopped processing. A processing error occurred. If these errors persist, a repair of the coprocessor hardware or a correction to the coprocessor software might be required.</td>
</tr>
</tbody>
</table>

Note: If an application receives a return code greater than 4, an error occurred. In the case of an error, assume any output variables other than the return code and reason code are not valid, unless otherwise indicated in the description of verb processing.

Reason codes

A reason code details the results of verb processing.
Every reason code is associated with a single return code. A nonzero reason code can be associated with a zero return code.

User Defined Extensions (UDX) return reason codes in the range of 20480 (X’5000’) - 24575 (X’5FFF’).

The remainder of this appendix lists the reason codes that accompany each of the return codes. The return codes are shown in decimal form and the reason codes are shown in decimal and in hexadecimal (hex) form.

**Reason codes that accompany return code 0**

Reason codes that accompany return code 0.

These codes are listed in [Table 145](#).

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000 (000)</td>
<td>The verb completed processing successfully.</td>
</tr>
<tr>
<td>0</td>
<td>002 (002)</td>
<td>One or more bytes of a key do not have odd parity.</td>
</tr>
<tr>
<td>0</td>
<td>008 (008)</td>
<td>No value is present to be processed.</td>
</tr>
<tr>
<td>0</td>
<td>151 (097)</td>
<td>The key token supplies the MAC length or MACLEN4 is the default for key tokens that contain MAC or MACVER keys.</td>
</tr>
<tr>
<td>0</td>
<td>701 (2BD)</td>
<td>A new master-key value has duplicate thirds.</td>
</tr>
<tr>
<td>0</td>
<td>702 (2BE)</td>
<td>A provided master-key part does not have odd parity. See &quot;Master Key Process (CSNBMKP)&quot; on page 119 about parity requirements for master key parts.</td>
</tr>
<tr>
<td>0</td>
<td>2013 (7DD)</td>
<td>The Pending Change Buffer (PCB) is empty. This return code and reason code pair applies only to IBM System z.</td>
</tr>
<tr>
<td>0</td>
<td>2146 (862)</td>
<td>A weaker key was used to wrap a stronger key and the Warn When Wrapping Weak Keys command (offset X’032C’) was enabled in the active role.</td>
</tr>
<tr>
<td>0</td>
<td>3010 (BC2)</td>
<td>This card is currently disabled. A card is placed in this state so that it can be moved from one piece of hardware to another, while keeping its secret keys and master keys intact. Normally, when a card has been moved a 'tamper’ event is recorded and all secrets are erased. A TKE workstation is typically required to put a card in this state and to remove it from this state after the card is installed on the new hardware. This return code and reason code pair applies only to IBM System z.</td>
</tr>
<tr>
<td>0</td>
<td>10001 (2711)</td>
<td>A key encrypted under the old master key was used.</td>
</tr>
<tr>
<td>0</td>
<td>10 002 (2712)</td>
<td>A fully qualified dataset name is longer than 64 bytes and the environment variable CSU.xxxLD is not defined (where xxx is either AES, DES, or PKA). The current directory has been abbreviated as a single dot (period).</td>
</tr>
<tr>
<td>0</td>
<td>10 003 (2713)</td>
<td>A fully qualified dataset name is longer than 64 bytes and the environment variable CSU.xxxLD is defined (where xxx is either AES, DES, or PKA). Only the dataset name is returned. Use the CSU.xxxLD environment variable to determine the fully qualified dataset name.</td>
</tr>
</tbody>
</table>
Reason codes that accompany return code 4

These codes are listed in Table 146.

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>001 (001)</td>
<td>The verification test failed.</td>
</tr>
<tr>
<td>4</td>
<td>013 (00D)</td>
<td>The key token has an initialization vector and the initialization_vector parameter value is nonzero. The verb uses the value in the key token.</td>
</tr>
<tr>
<td>4</td>
<td>016 (010)</td>
<td>The rule_array and the rule_array_count are too small to contain the complete result.</td>
</tr>
<tr>
<td>4</td>
<td>017 (011)</td>
<td>The requested ID is not present in any profile in the specified cryptographic hardware component.</td>
</tr>
<tr>
<td>4</td>
<td>019 (013)</td>
<td>The financial PIN in a PIN block is not verified.</td>
</tr>
<tr>
<td>4</td>
<td>158 (09E)</td>
<td>The verb did not process any key records.</td>
</tr>
<tr>
<td>4</td>
<td>166 (0A6)</td>
<td>The control-vector is not valid because of parity bits, anti-variant bits, inconsistent KEK bits or because bits 59 - 62 are not zero.</td>
</tr>
<tr>
<td>4</td>
<td>179 (0B3)</td>
<td>The control-vector keywords in the rule_array are ignored.</td>
</tr>
<tr>
<td>4</td>
<td>283 (11B)</td>
<td>The coprocessor battery is low.</td>
</tr>
<tr>
<td>4</td>
<td>287 (11F)</td>
<td>The PIN-block format is not consistent.</td>
</tr>
<tr>
<td>4</td>
<td>429 (1AD)</td>
<td>The digital signature is not verified. The verb completed its processing normally.</td>
</tr>
<tr>
<td>4</td>
<td>1024 (400)</td>
<td>Sufficient shares have been processed to create a new master key.</td>
</tr>
<tr>
<td>4</td>
<td>2039 (7F7)</td>
<td>At least one control vector bit cannot be parsed.</td>
</tr>
<tr>
<td>4</td>
<td>2042 (7FA)</td>
<td>The supplied passphrase is not valid.</td>
</tr>
<tr>
<td>4</td>
<td>2133 (855)</td>
<td>The verb_data value identifies one or more PIN decimalization tables to be deleted that are not stored on the coprocessor. All PIN tables that were requested to be deleted are removed.</td>
</tr>
</tbody>
</table>

Reason codes that accompany return code 8

The codes are listed in Table 147.

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>012 (00C)</td>
<td>The token-validation value in an external key token is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>022 (016)</td>
<td>The ID number in the request field is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>023 (017)</td>
<td>An access to the data area is outside the data-area boundary.</td>
</tr>
<tr>
<td>8</td>
<td>024 (018)</td>
<td>The master key verification pattern is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>025 (019)</td>
<td>The value that the text_length parameter specifies is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>026 (01A)</td>
<td>The value of the PIN is not valid.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>029 (01D)</td>
<td>The token-validation value in an internal key token is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>030 (01E)</td>
<td>No record with a matching key label is in key storage.</td>
</tr>
<tr>
<td>8</td>
<td>031 (01F)</td>
<td>The control vector does not specify a DATA key.</td>
</tr>
<tr>
<td>8</td>
<td>032 (020)</td>
<td>A key label format is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>033 (021)</td>
<td>A rule_array or other parameter specifies a keyword that is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>034 (022)</td>
<td>A rule_array keyword combination is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>035 (023)</td>
<td>A rule_array_count is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>036 (024)</td>
<td>The action command must be specified in the rule_array.</td>
</tr>
<tr>
<td>8</td>
<td>037 (025)</td>
<td>The object type must be specified in the rule_array.</td>
</tr>
<tr>
<td>8</td>
<td>039 (027)</td>
<td>A control vector violation occurred. Check all control vectors employed with the verb. For security reasons, no detail is provided.</td>
</tr>
<tr>
<td>8</td>
<td>040 (028)</td>
<td>The service code does not contain numerical character data.</td>
</tr>
<tr>
<td>8</td>
<td>041 (029)</td>
<td>The keyword supplied with the key_form parameter is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>042 (02A)</td>
<td>The expiration date is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>043 (02B)</td>
<td>The keyword supplied with the key_length or the key_token_length parameter is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>044 (02C)</td>
<td>A record with a matching key label already exists in key storage.</td>
</tr>
<tr>
<td>8</td>
<td>045 (02D)</td>
<td>The input character string cannot be found in the code table.</td>
</tr>
<tr>
<td>8</td>
<td>046 (02E)</td>
<td>The card-validation value (CVV) is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>047 (02F)</td>
<td>A source key token is unusable because it contains data that is not valid or is undefined.</td>
</tr>
<tr>
<td>8</td>
<td>048 (030)</td>
<td>One or more keys has a master key verification pattern that is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>049 (031)</td>
<td>A key-token-version-number found in a key token is not supported.</td>
</tr>
<tr>
<td>8</td>
<td>050 (032)</td>
<td>The key-serial-number specified in the rule_array is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>051 (033)</td>
<td>The value that the text_length parameter specifies is not a multiple of eight bytes.</td>
</tr>
<tr>
<td>8</td>
<td>054 (036)</td>
<td>The value that the pad_character parameter specifies is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>055 (037)</td>
<td>The initialization vector in the key token is enciphered.</td>
</tr>
<tr>
<td>8</td>
<td>056 (038)</td>
<td>The master key verification pattern in the OCV is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>058 (03A)</td>
<td>The parity of the operating key is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>059 (03B)</td>
<td>Control information (for example, the processing method or the pad character) in the key token conflicts with that in the rule_array.</td>
</tr>
<tr>
<td>8</td>
<td>060 (03C)</td>
<td>A cryptographic request with the FIRST or MIDDLE keywords and a text length less than eight bytes is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>061 (03D)</td>
<td>The keyword supplied with the key_type parameter is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>062 (03E)</td>
<td>The source key is not present.</td>
</tr>
<tr>
<td>8</td>
<td>063 (03F)</td>
<td>A key token has an invalid token header (for example, not an internal token).</td>
</tr>
<tr>
<td>8</td>
<td>064 (040)</td>
<td>The RSA key is not permitted to perform the requested operation. Likely cause is key distribution usage is not enabled for the key.</td>
</tr>
<tr>
<td>8</td>
<td>065 (041)</td>
<td>The key token failed consistency checking.</td>
</tr>
</tbody>
</table>
### Table 147. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 066 (042)</td>
<td></td>
<td>The recovered encryption block failed validation checking.</td>
</tr>
<tr>
<td>8 067 (043)</td>
<td></td>
<td>RSA encryption failed.</td>
</tr>
<tr>
<td>8 068 (044)</td>
<td></td>
<td>RSA decryption failed.</td>
</tr>
<tr>
<td>8 070 (046)</td>
<td></td>
<td>An invalid block identifier (identifier tag) was found. Either a block ID (identifier tag) that was proprietary was found, a reserved block ID was used, a duplicate block ID was found, or the specified optional block in the TR-31 key block could not be found.</td>
</tr>
<tr>
<td>8 072 (048)</td>
<td></td>
<td>The value that the size parameter specifies is not valid (too small, too large, negative, or zero).</td>
</tr>
<tr>
<td>8 081 (051)</td>
<td></td>
<td>The modulus length (key size) exceeds the allowable maximum.</td>
</tr>
<tr>
<td>8 085 (055)</td>
<td></td>
<td>The date or the time value is not valid.</td>
</tr>
<tr>
<td>8 090 (05A)</td>
<td></td>
<td>Access control checking failed. See the Required Commands descriptions for the failing verb.</td>
</tr>
<tr>
<td>8 091 (05B)</td>
<td></td>
<td>The time that was sent in your logon request was more than five minutes different from the clock in the secure module.</td>
</tr>
<tr>
<td>8 092 (05C)</td>
<td></td>
<td>The user profile is expired.</td>
</tr>
<tr>
<td>8 093 (05D)</td>
<td></td>
<td>The user profile has not yet reached its activation date.</td>
</tr>
<tr>
<td>8 094 (05E)</td>
<td></td>
<td>The authentication data (for example, passphrase) is expired.</td>
</tr>
<tr>
<td>8 095 (05F)</td>
<td></td>
<td>Access to the data is not authorized.</td>
</tr>
<tr>
<td>8 096 (060)</td>
<td></td>
<td>An error occurred reading or writing the secure clock.</td>
</tr>
<tr>
<td>8 100 (064)</td>
<td></td>
<td>The PIN length is not valid.</td>
</tr>
<tr>
<td>8 101 (065)</td>
<td></td>
<td>The PIN check length is not valid. It must be in the range from 4 to the PIN length inclusive.</td>
</tr>
<tr>
<td>8 102 (066)</td>
<td></td>
<td>The value of the decimalization table is not valid.</td>
</tr>
<tr>
<td>8 103 (067)</td>
<td></td>
<td>The value of the validation data is not valid.</td>
</tr>
<tr>
<td>8 104 (068)</td>
<td></td>
<td>The value of the customer-selected PIN is not valid or the PIN length does not match the value supplied with the PIN_length parameter or defined by the PIN-block format specified in the PIN profile.</td>
</tr>
<tr>
<td>8 105 (069)</td>
<td></td>
<td>The value of the transaction_security parameter is not valid.</td>
</tr>
<tr>
<td>8 106 (06A)</td>
<td></td>
<td>The PIN-block format keyword is not valid.</td>
</tr>
<tr>
<td>8 107 (06B)</td>
<td></td>
<td>The format control keyword is not valid.</td>
</tr>
<tr>
<td>8 108 (06C)</td>
<td></td>
<td>The value or the placement of the padding data is not valid.</td>
</tr>
<tr>
<td>8 109 (06D)</td>
<td></td>
<td>The extraction method keyword is not valid.</td>
</tr>
<tr>
<td>8 110 (06E)</td>
<td></td>
<td>The value of the PAN data is not numeric character data.</td>
</tr>
<tr>
<td>8 111 (06F)</td>
<td></td>
<td>The sequence number is not valid.</td>
</tr>
<tr>
<td>8 112 (070)</td>
<td></td>
<td>The PIN offset is not valid.</td>
</tr>
<tr>
<td>8 114 (072)</td>
<td></td>
<td>The PVV value is not valid.</td>
</tr>
<tr>
<td>8 116 (074)</td>
<td></td>
<td>The clear PIN value is not valid. For example, digits other than 0 - 9 were found.</td>
</tr>
<tr>
<td>8 120 (078)</td>
<td></td>
<td>An origin or destination identifier is not valid.</td>
</tr>
<tr>
<td>8 121 (079)</td>
<td></td>
<td>The value of the inbound_key or source_key parameter is not valid.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>122 (07A)</td>
<td>The value of the <code>inbound KEK_count</code> or <code>outbound_count</code> parameter is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>125 (07D)</td>
<td>A PKA92-encrypted key having the same Environment Identifier (EID) as the local node cannot be imported.</td>
</tr>
<tr>
<td>8</td>
<td>129 (081)</td>
<td>Required rule-array keyword not found.</td>
</tr>
<tr>
<td>8</td>
<td>153 (099)</td>
<td>The text length exceeds the system limits.</td>
</tr>
<tr>
<td>8</td>
<td>154 (09A)</td>
<td>The key token the <code>key_identifier</code> parameter specifies is not an internal key-token or a key label.</td>
</tr>
<tr>
<td>8</td>
<td>155 (09B)</td>
<td>The value that the <code>generated_key_identifier</code> parameter specifies is not valid or it is not consistent with the value that the <code>key_form</code> parameter specifies.</td>
</tr>
<tr>
<td>8</td>
<td>156 (09C)</td>
<td>A keyword is not valid with the specified parameters.</td>
</tr>
<tr>
<td>8</td>
<td>157 (09D)</td>
<td>The key-token type is not specified in the <code>rule_array</code>.</td>
</tr>
<tr>
<td>8</td>
<td>159 (09F)</td>
<td>The keyword supplied with the option parameter is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>160 (0A0)</td>
<td>The key type and the key length are not consistent.</td>
</tr>
<tr>
<td>8</td>
<td>161 (0A1)</td>
<td>The value that the <code>dataset_name_length</code> parameter specifies is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>162 (0A2)</td>
<td>The offset value is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>163 (0A3)</td>
<td>The value that the <code>dataset_name</code> parameter specifies is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>164 (0A4)</td>
<td>The starting address of the output area falls inside the input area.</td>
</tr>
<tr>
<td>8</td>
<td>165 (0A5)</td>
<td>The <code>carry_over_character_count</code> specified in the chaining vector is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>168 (0A8)</td>
<td>A hexadecimal MAC value contains characters that are not valid or the MAC, on a request or reply failed, because the user session key in the host and the adapter card do not match.</td>
</tr>
<tr>
<td>8</td>
<td>169 (0A9)</td>
<td>Specific to MDC Generate, indicates that the length of the text supplied is not correct, either not long enough for the algorithm parameters used or not the correct multiple (must be multiple of eight bytes).</td>
</tr>
<tr>
<td>8</td>
<td>170 (0AA)</td>
<td>Special authorization through the operating system is required to use this verb.</td>
</tr>
<tr>
<td>8</td>
<td>171 (0AB)</td>
<td>The <code>control_array_count</code> value is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>175 (0AF)</td>
<td>The key token cannot be parsed because no control vector is present.</td>
</tr>
<tr>
<td>8</td>
<td>180 (0B4)</td>
<td>A key token presented for parsing is null.</td>
</tr>
<tr>
<td>8</td>
<td>181 (0B5)</td>
<td>The key token is not valid. The first byte is not valid or an incorrect token type was presented.</td>
</tr>
<tr>
<td>8</td>
<td>183 (0B7)</td>
<td>The key type is not consistent with the key type of the control vector.</td>
</tr>
<tr>
<td>8</td>
<td>184 (0B8)</td>
<td>An input pointer is null.</td>
</tr>
<tr>
<td>8</td>
<td>185 (0B9)</td>
<td>A disk I/O error occurred: perhaps the file is in-use, does not exist, and so forth.</td>
</tr>
<tr>
<td>8</td>
<td>186 (0BA)</td>
<td>The key-type field in the control vector is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>187 (0BB)</td>
<td>The requested MAC length (MACLEN4, MACLEN6, MACLEN8) is not consistent with the control vector (key-A, key-B).</td>
</tr>
<tr>
<td>8</td>
<td>191 (0BF)</td>
<td>The requested MAC length (MACLEN6, MACLEN8) is not consistent with the control vector (MAC-LN-4).</td>
</tr>
<tr>
<td>8</td>
<td>192 (0C0)</td>
<td>A key-storage record contains a record validation value that is not valid.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8 204 (0CC)</td>
<td>A memory allocation failed. This can occur in the host and in the coprocessor. Try closing other host tasks. If the problem persists, contact the IBM support center.</td>
<td></td>
</tr>
<tr>
<td>8 205 (0CD)</td>
<td>The X9.23 ciphering method is not consistent with the use of the CONTINUE keyword.</td>
<td></td>
</tr>
<tr>
<td>8 323 (143)</td>
<td>The ciphering method the Decipher verb used does not match the ciphering method the Encipher verb used.</td>
<td></td>
</tr>
<tr>
<td>8 335 (14F)</td>
<td>Either the specified cryptographic hardware component or the environment cannot implement this function.</td>
<td></td>
</tr>
<tr>
<td>8 340 (154)</td>
<td>One of the input control vectors has odd parity.</td>
<td></td>
</tr>
<tr>
<td>8 343 (157)</td>
<td>Either the data block or the buffer for the block is too small or a variable has caused an attempt to create an internal data structure that is too large.</td>
<td></td>
</tr>
<tr>
<td>8 345 (159)</td>
<td>Insufficient storage space exists for the data in the data block buffer.</td>
<td></td>
</tr>
<tr>
<td>8 374 (176)</td>
<td>Less data was supplied than expected or less data exists than was requested.</td>
<td></td>
</tr>
<tr>
<td>8 377 (179)</td>
<td>A key-storage error occurred.</td>
<td></td>
</tr>
<tr>
<td>8 382 (17E)</td>
<td>A time-limit violation occurred.</td>
<td></td>
</tr>
<tr>
<td>8 385 (181)</td>
<td>The cryptographic hardware component reported that the data passed as part of a command is not valid for that command.</td>
<td></td>
</tr>
<tr>
<td>8 387 (183)</td>
<td>The cryptographic hardware component reported that the user ID or role ID is not valid.</td>
<td></td>
</tr>
<tr>
<td>8 393 (189)</td>
<td>The command was not processed because the profile cannot be used.</td>
<td></td>
</tr>
<tr>
<td>8 394 (18A)</td>
<td>The command was not processed because the expiration date was exceeded.</td>
<td></td>
</tr>
<tr>
<td>8 397 (18D)</td>
<td>The command was not processed because the active profile requires the user to be verified first.</td>
<td></td>
</tr>
<tr>
<td>8 398 (18E)</td>
<td>The command was not processed because the maximum PIN or password failure limit is exceeded.</td>
<td></td>
</tr>
<tr>
<td>8 407 (197)</td>
<td>There is a PIN-block consistency-check-error.</td>
<td></td>
</tr>
<tr>
<td>8 439 (1B7)</td>
<td>Key cannot be completed because all required key parts have not yet been accumulated, or key is already complete.</td>
<td></td>
</tr>
<tr>
<td>8 441 (1B9)</td>
<td>Key part cannot be added because key is complete. The key to be processed should be partial, but the key is not partial according to the control vector or other control bits of the key.</td>
<td></td>
</tr>
<tr>
<td>8 442 (1BA)</td>
<td>DES keys with replicated halves are not allowed.</td>
<td></td>
</tr>
<tr>
<td>8 605 (25D)</td>
<td>The number of output bytes is greater than the number that is permitted.</td>
<td></td>
</tr>
<tr>
<td>8 703 (2BF)</td>
<td>A new master-key value is one of the weak DES keys.</td>
<td></td>
</tr>
<tr>
<td>8 704 (2C0)</td>
<td>A new master key cannot have the same master key version number as the current master-key.</td>
<td></td>
</tr>
<tr>
<td>8 705 (2C1)</td>
<td>Both exporter keys specify the same key-encrypting key.</td>
<td></td>
</tr>
<tr>
<td>8 706 (2C2)</td>
<td>Pad count in deciphered data is not valid.</td>
<td></td>
</tr>
<tr>
<td>8 707 (2C3)</td>
<td>The master-key registers are not in the state required for the requested function.</td>
<td></td>
</tr>
<tr>
<td>8 714 (2CA)</td>
<td>A reserved parameter must be a null pointer or an expected value.</td>
<td></td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>715 (2CB)</td>
<td>A parameter that must have a value of zero is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>718 (2CE)</td>
<td>The hash value of the data block in the decrypted RSA-OAEP block does not match the hash of the decrypted data block.</td>
</tr>
<tr>
<td>8</td>
<td>719 (2CF)</td>
<td>The block format (BT) field in the decrypted RSA-OAEP block does not have the correct value.</td>
</tr>
<tr>
<td>8</td>
<td>720 (2D0)</td>
<td>The initial byte (I) in the decrypted RSA-OAEP block does not have a valid value.</td>
</tr>
<tr>
<td>8</td>
<td>721 (2D1)</td>
<td>The V field in the decrypted RSA-OAEP does not have the correct value.</td>
</tr>
<tr>
<td>8</td>
<td>752 (2F0)</td>
<td>The key-storage file path is not usable.</td>
</tr>
<tr>
<td>8</td>
<td>753 (2F1)</td>
<td>Opening the key-storage file failed.</td>
</tr>
<tr>
<td>8</td>
<td>754 (2F2)</td>
<td>An internal call to the key_test command failed.</td>
</tr>
<tr>
<td>8</td>
<td>756 (2F4)</td>
<td>Creation of the key-storage file failed.</td>
</tr>
<tr>
<td>8</td>
<td>760 (2F8)</td>
<td>An RSA-key modulus length in bits or in bytes is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>761 (2F9)</td>
<td>An RSA-key exponent length is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>762 (2FA)</td>
<td>A length in the key value structure is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>763 (2FB)</td>
<td>The section identification number within a key token is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>770 (302)</td>
<td>The PKA key token has a field that is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>771 (303)</td>
<td>The user is not logged on.</td>
</tr>
<tr>
<td>8</td>
<td>772 (304)</td>
<td>The requested role does not exist.</td>
</tr>
<tr>
<td>8</td>
<td>773 (305)</td>
<td>The requested profile does not exist.</td>
</tr>
<tr>
<td>8</td>
<td>774 (306)</td>
<td>The profile already exists.</td>
</tr>
<tr>
<td>8</td>
<td>775 (307)</td>
<td>The supplied data is not replaceable.</td>
</tr>
<tr>
<td>8</td>
<td>776 (308)</td>
<td>The requested ID is already logged on.</td>
</tr>
<tr>
<td>8</td>
<td>777 (309)</td>
<td>The authentication data is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>778 (30A)</td>
<td>The checksum for the role is in error.</td>
</tr>
<tr>
<td>8</td>
<td>779 (30B)</td>
<td>The checksum for the profile is in error.</td>
</tr>
<tr>
<td>8</td>
<td>780 (30C)</td>
<td>There is an error in the profile data.</td>
</tr>
<tr>
<td>8</td>
<td>781 (30D)</td>
<td>There is an error in the role data.</td>
</tr>
<tr>
<td>8</td>
<td>782 (30E)</td>
<td>The function-control-vector header is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>783 (30F)</td>
<td>The command is not permitted by the function-control-vector value.</td>
</tr>
<tr>
<td>8</td>
<td>784 (310)</td>
<td>The operation you requested cannot be performed because the user profile is in use.</td>
</tr>
<tr>
<td>8</td>
<td>785 (311)</td>
<td>The operation you requested cannot be performed because the role is in use.</td>
</tr>
<tr>
<td>8</td>
<td>1025 (401)</td>
<td>The registered public key or retained private key name already exists.</td>
</tr>
<tr>
<td>8</td>
<td>1026 (402)</td>
<td>The key name (registered public key or retained private key) does not exist.</td>
</tr>
<tr>
<td>8</td>
<td>1027 (403)</td>
<td>Environment identifier data is already set.</td>
</tr>
<tr>
<td>8</td>
<td>1028 (404)</td>
<td>Master key share data is already set.</td>
</tr>
<tr>
<td>8</td>
<td>1029 (405)</td>
<td>There is an error in the Environment Identifier (EID) data.</td>
</tr>
<tr>
<td>8</td>
<td>1030 (406)</td>
<td>There is an error in using the master key share data.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>1031 (407)</td>
<td>There is an error in using registered public key or retained private key data.</td>
</tr>
<tr>
<td>8</td>
<td>1032 (408)</td>
<td>There is an error in using registered public key hash data.</td>
</tr>
<tr>
<td>8</td>
<td>1033 (409)</td>
<td>The public key hash was not registered.</td>
</tr>
<tr>
<td>8</td>
<td>1034 (40A)</td>
<td>The public key was not registered.</td>
</tr>
<tr>
<td>8</td>
<td>1035 (40B)</td>
<td>The public key certificate signature was not verified.</td>
</tr>
<tr>
<td>8</td>
<td>1037 (40D)</td>
<td>There is a master key shares distribution error.</td>
</tr>
<tr>
<td>8</td>
<td>1038 (40E)</td>
<td>The public key hash is not marked for cloning.</td>
</tr>
<tr>
<td>8</td>
<td>1039 (40F)</td>
<td>The registered public key hash does not match the registered hash.</td>
</tr>
<tr>
<td>8</td>
<td>1040 (410)</td>
<td>The master key share enciphering key failed encipher.</td>
</tr>
<tr>
<td>8</td>
<td>1041 (411)</td>
<td>The master key share enciphering key failed decipher.</td>
</tr>
<tr>
<td>8</td>
<td>1042 (412)</td>
<td>The master key share digital signature generate failed.</td>
</tr>
<tr>
<td>8</td>
<td>1043 (413)</td>
<td>The master key share digital signature verify failed.</td>
</tr>
<tr>
<td>8</td>
<td>1044 (414)</td>
<td>There is an error in reading VPD data from the adapter.</td>
</tr>
<tr>
<td>8</td>
<td>1045 (415)</td>
<td>Encrypting the cloning information failed.</td>
</tr>
<tr>
<td>8</td>
<td>1046 (416)</td>
<td>Decrypting the cloning information failed.</td>
</tr>
<tr>
<td>8</td>
<td>1047 (417)</td>
<td>There is an error loading the new master key from the master key shares.</td>
</tr>
<tr>
<td>8</td>
<td>1048 (418)</td>
<td>The clone information has one or more sections that are not valid.</td>
</tr>
<tr>
<td>8</td>
<td>1049 (419)</td>
<td>The master key share index is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>1050 (41A)</td>
<td>The public-key encrypted-key is rejected because the Environment Identifier (EID) with the key is the same as the EID for this node.</td>
</tr>
<tr>
<td>8</td>
<td>1051 (41B)</td>
<td>The private key is rejected because the key is not flagged for use in master-key cloning.</td>
</tr>
<tr>
<td>8</td>
<td>1052 (41C)</td>
<td>The token identifier of the trusted block's header section is in the range X'20' - X'FF'. Check the token identifier of the trusted block.</td>
</tr>
<tr>
<td>8</td>
<td>1053 (41D)</td>
<td>The active flag in the trusted block's trusted block section X'14' is not disabled. Use the Trusted Block Create verb to create an inactive/external trusted block.</td>
</tr>
<tr>
<td>8</td>
<td>1054 (41E)</td>
<td>The token identifier of the trusted block's header section is not X'1E' (external). Use the Trusted Block Create verb to create an inactive/external trusted block.</td>
</tr>
<tr>
<td>8</td>
<td>1055 (41F)</td>
<td>The active flag of the trusted block's trusted block section X'14' is not enabled. Use the Trusted Block Create verb to create an active/external trusted block.</td>
</tr>
<tr>
<td>8</td>
<td>1056 (420)</td>
<td>The token identifier of the trusted block's header section is not X'1F' (internal). Use the PKA Key Import verb to import the trusted block.</td>
</tr>
<tr>
<td>8</td>
<td>1057 (421)</td>
<td>The trusted block rule section X'T2' rule ID does not match input parameter rule ID. Verify that the trusted block used has the rule section specified.</td>
</tr>
<tr>
<td>8</td>
<td>1058 (422)</td>
<td>The trusted block contains a value that is too small or too large.</td>
</tr>
<tr>
<td>8</td>
<td>1059 (423)</td>
<td>A trusted block parameter that must have a value of zero (or a grouping of bits set to zero) is invalid.</td>
</tr>
<tr>
<td>8</td>
<td>1060 (424)</td>
<td>The trusted block public key section failed consistency checking.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>1061 (425)</td>
<td>The trusted block contains extraneous sections or subsections (TLVs). Check the trusted block for undefined sections or subsections.</td>
</tr>
<tr>
<td>8</td>
<td>1062 (426)</td>
<td>The trusted block contains missing sections or subsections (TLVs). Check the trusted block for required sections and subsections applicable to the verb invoked.</td>
</tr>
<tr>
<td>8</td>
<td>1063 (427)</td>
<td>The trusted block contains duplicate sections or subsections (TLVs). Check the trusted block’s sections and subsections for duplicates. Multiple rule sections are allowed.</td>
</tr>
<tr>
<td>8</td>
<td>1064 (428)</td>
<td>The trusted block expiration date has expired (as compared to the IBM 4764 clock). Validate the expiration date in the trusted block’s trusted information section’s Activation and Expiration Date TLV object.</td>
</tr>
<tr>
<td>8</td>
<td>1065 (429)</td>
<td>The trusted block expiration date is at a date prior to the activation date. Validate the expiration date in the trusted block’s trusted information section’s Activation and Expiration Date TLV object.</td>
</tr>
<tr>
<td>8</td>
<td>1066 (42A)</td>
<td>The trusted block public key modulus length in bits is not consistent with the byte length. The bit length must be less than or equal to byte length * 8 and greater than (byte length - 1) * 8.</td>
</tr>
<tr>
<td>8</td>
<td>1067 (42B)</td>
<td>The trusted block public key modulus length in bits exceeds the maximum allowed bit length, as defined by the Function Control Vector.</td>
</tr>
<tr>
<td>8</td>
<td>1068 (42C)</td>
<td>One or more trusted block sections or TLV objects contained data that is invalid (an example would be invalid label data in label section X’13’).</td>
</tr>
<tr>
<td>8</td>
<td>1069 (42D)</td>
<td>Trusted block verification was attempted by a verb other than CSNDDSV, CSNDKTC, CSNDPKI, CSNDRKX, or CSNDTBC.</td>
</tr>
<tr>
<td>8</td>
<td>1070 (42E)</td>
<td>The trusted block rule ID contained within a rule section has invalid characters.</td>
</tr>
<tr>
<td>8</td>
<td>1071 (42F)</td>
<td>The source key’s length or CV does not match what is expected by the rule section in the trusted block that was selected by the rule ID input parameter.</td>
</tr>
<tr>
<td>8</td>
<td>1072 (430)</td>
<td>The activation data is not valid. Validate the activation data in the trusted block’s trusted information section’s Activation and Expiration Date TLV object.</td>
</tr>
<tr>
<td>8</td>
<td>1073 (431)</td>
<td>The source-key label does not match the template in the export key DES token parameters TLV object of the selected trusted block rule section.</td>
</tr>
<tr>
<td>8</td>
<td>1074 (432)</td>
<td>The control-vector value specified in the common export key parameters TLV object in the selected rule section of the trusted block contains a control vector that is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>1075 (433)</td>
<td>The source-key label template in the export key DES token parameters TLV object in the selected rule section of the trusted block contains a label template that is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>1077 (435)</td>
<td>Key wrapping option input error.</td>
</tr>
<tr>
<td>8</td>
<td>1078 (436)</td>
<td>Key wrapping Security Relevant Data Item (SRDI) error.</td>
</tr>
<tr>
<td>8</td>
<td>1100 (44C)</td>
<td>There is a general hardware device driver execution error.</td>
</tr>
<tr>
<td>8</td>
<td>1101 (44D)</td>
<td>There is a hardware device driver parameter that is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>1102 (44E)</td>
<td>There is a hardware device driver non-valid buffer length.</td>
</tr>
<tr>
<td>8</td>
<td>1103 (44F)</td>
<td>The hardware device driver has too many opens. The device cannot open now.</td>
</tr>
</tbody>
</table>
### Table 147. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1104 (450)</td>
<td>The hardware device driver is denied access.</td>
</tr>
<tr>
<td>8</td>
<td>1105 (451)</td>
<td>The hardware device driver device is busy and cannot perform the request now.</td>
</tr>
<tr>
<td>8</td>
<td>1106 (452)</td>
<td>The hardware device driver buffer is too small and the received data is truncated.</td>
</tr>
<tr>
<td>8</td>
<td>1107 (453)</td>
<td>The hardware device driver request is interrupted and the request is aborted.</td>
</tr>
<tr>
<td>8</td>
<td>1108 (454)</td>
<td>The hardware device driver detected a security tamper event.</td>
</tr>
<tr>
<td>8</td>
<td>1114 (45A)</td>
<td>The communications manager detected that the host-supplied buffer for the reply control block is too small.</td>
</tr>
<tr>
<td>8</td>
<td>1115 (45B)</td>
<td>The communications manager detected that the host-supplied buffer for the reply data block is too small.</td>
</tr>
<tr>
<td>8</td>
<td>2034 (7F2)</td>
<td>The environment variable that was used to set the default coprocessor is not valid, or does not exist for a coprocessor in the system.</td>
</tr>
<tr>
<td>8</td>
<td>2036 (7F4)</td>
<td>The contents of a chaining vector are not valid. Ensure the chaining vector was not modified by your application program.</td>
</tr>
<tr>
<td>8</td>
<td>2038 (7F6)</td>
<td>No RSA private key information is provided.</td>
</tr>
<tr>
<td>8</td>
<td>2041 (7F9)</td>
<td>A default card environment variable is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>2050 (802)</td>
<td>The current key serial number field in the PIN profile variable is not valid (not hexadecimal or too many one bits).</td>
</tr>
<tr>
<td>8</td>
<td>2051 (803)</td>
<td>There is a non-valid message length in the OAEP-decoded information.</td>
</tr>
<tr>
<td>8</td>
<td>2053 (805)</td>
<td>No message found in the OAEP-decoded data.</td>
</tr>
<tr>
<td>8</td>
<td>2054 (806)</td>
<td>There is a non-valid RSA Enciphered Key cryptogram: OAEP optional encoding parameters failed validation.</td>
</tr>
<tr>
<td>8</td>
<td>2055 (807)</td>
<td>Based on the hash method and size of the symmetric key specified, the RSA public key size is too small to format the symmetric key into a PKOAEP2 message.</td>
</tr>
<tr>
<td>8</td>
<td>2062 (80E)</td>
<td>The active role does not permit you to change the characteristic of a double-length key in the key_Part_Import parameter.</td>
</tr>
<tr>
<td>8</td>
<td>2065 (811)</td>
<td>The specified key token is not null.</td>
</tr>
<tr>
<td>8</td>
<td>2080 (820)</td>
<td>The group profile was not found.</td>
</tr>
<tr>
<td>8</td>
<td>2081 (821)</td>
<td>The group has duplicate elements.</td>
</tr>
<tr>
<td>8</td>
<td>2082 (822)</td>
<td>The group profile is not in the group.</td>
</tr>
<tr>
<td>8</td>
<td>2083 (823)</td>
<td>The group has the wrong user ID count.</td>
</tr>
<tr>
<td>8</td>
<td>2084 (824)</td>
<td>The group user ID failed.</td>
</tr>
<tr>
<td>8</td>
<td>2085 (825)</td>
<td>The profile is not in the specified group.</td>
</tr>
<tr>
<td>8</td>
<td>2086 (826)</td>
<td>The group role was not found.</td>
</tr>
<tr>
<td>8</td>
<td>2087 (827)</td>
<td>The group profile has not been activated.</td>
</tr>
<tr>
<td>8</td>
<td>2088 (828)</td>
<td>The expiration date of the group profile has been reached or exceeded.</td>
</tr>
<tr>
<td>8</td>
<td>2089 (829)</td>
<td>The verb contains multiple keywords or parameters that indicate the algorithm to be used, and at least one of these specifies a different algorithm from the others.</td>
</tr>
<tr>
<td>8</td>
<td>2090 (82A)</td>
<td>A required SRDI was not found.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>2091 (82B)</td>
<td>A required CA SRDI was not found.</td>
</tr>
<tr>
<td>8</td>
<td>2093 (82D)</td>
<td>Specific to IBM System z - an AES key is encrypted under a DES master key, which is not acceptable for the requested operation.</td>
</tr>
<tr>
<td>8</td>
<td>2095 (82F)</td>
<td>The key_form is incompatible with the key_type.</td>
</tr>
<tr>
<td>8</td>
<td>2097 (831)</td>
<td>The key_length is incompatible with the key_type.</td>
</tr>
<tr>
<td>8</td>
<td>2098 (832)</td>
<td>Either a key bit length that was not valid was found in an AES key token (length not 128, 192, or 256 bits) or a version X'01' DES token had a token-marks field that was not valid.</td>
</tr>
<tr>
<td>8</td>
<td>2099 (833)</td>
<td>Invalid encrypted key length in the AES token, when an encrypted key is present.</td>
</tr>
<tr>
<td>8</td>
<td>2106 (83A)</td>
<td>An input/output error occurred while accessing the logged on users table.</td>
</tr>
<tr>
<td>8</td>
<td>2110 (83E)</td>
<td>Invalid wrapping type.</td>
</tr>
<tr>
<td>8</td>
<td>2111 (83F)</td>
<td>Control vector enhanced bit (bit 56) conflicts with key wrapping keyword.</td>
</tr>
<tr>
<td>8</td>
<td>2113 (841)</td>
<td>A key token contains invalid payload.</td>
</tr>
<tr>
<td>8</td>
<td>2114 (842)</td>
<td>Clear-key bit length is out of range.</td>
</tr>
<tr>
<td>8</td>
<td>2115 (843)</td>
<td>Input key token cannot have a key present when importing the first key part; skeleton key token is required.</td>
</tr>
<tr>
<td>8</td>
<td>2118 (846)</td>
<td>One or more invalid values in the TR-31 key block header.</td>
</tr>
<tr>
<td>8</td>
<td>2119 (847)</td>
<td>The &quot;mode&quot; value in the TR-31 header is invalid or is not acceptable in the chosen operation.</td>
</tr>
<tr>
<td>8</td>
<td>2121 (849)</td>
<td>The &quot;algorithm&quot; value in the TR-31 header is invalid or is not acceptable in the chosen operation.</td>
</tr>
<tr>
<td>8</td>
<td>2122 (84A)</td>
<td>For import, the exportability byte in the TR-31 header contains a value that does not support import of the key into CCA. For export, the requested exportability does not match circumstances (for example, a 'B' Key Block Version ID key can be wrapped only by a KEK that is wrapped in CBC mode, the ECB mode KEK violates ANSI X9.24).</td>
</tr>
<tr>
<td>8</td>
<td>2123 (84B)</td>
<td>The length of the cleartext key in the TR-31 block is invalid (for example, the algorithm is 'D' for single-length DES, but the key length is not 64 bits).</td>
</tr>
<tr>
<td>8</td>
<td>2125 (84D)</td>
<td>The Key Block Version ID in the TR-31 header contains an invalid value.</td>
</tr>
<tr>
<td>8</td>
<td>2126 (84E)</td>
<td>The key-usage field in the TR-31 header contains a value that is not supported for import of the key into CCA.</td>
</tr>
<tr>
<td>8</td>
<td>2127 (84F)</td>
<td>The key-usage field in the TR-31 header contains a value that is not valid with the other parameters in the header.</td>
</tr>
<tr>
<td>8</td>
<td>2129 (851)</td>
<td>Either a parameter for building a TR-31 key block (a TR-31 key block or a component, such as a tag for an optional block) contains one or more ASCII characters that are not printable as described in TR-31, or a field contains ASCII characters that are not allowed for that field.</td>
</tr>
<tr>
<td>8</td>
<td>2130 (852)</td>
<td>The control vector carried in the optional blocks of the TR-31 key block is inconsistent with other attributes of the key.</td>
</tr>
<tr>
<td>8</td>
<td>2131 (853)</td>
<td>The TR-31 key-token failed the MAC validate step of the Key Block unwrap and verify steps (for either Key Block Version ID method). MAC validation failed for a parameter in a key block, such as a trusted block or a TR-31 key block. This might be the result of tampering, corruption, or using a validation key that is different from the one use to generate the MAC.</td>
</tr>
</tbody>
</table>
Table 147. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2134 (856)</td>
<td>No valid PIN decimalization tables are present.</td>
</tr>
<tr>
<td>8</td>
<td>2135 (857)</td>
<td>The PIN decimalization table provided as input is not allowed to be used because it does not match any of the active tables stored on the coprocessor.</td>
</tr>
<tr>
<td>8</td>
<td>2137 (859)</td>
<td>There is an error involving the PIN decimalization table input data. No PIN tables have been changed.</td>
</tr>
<tr>
<td>8</td>
<td>2138 (85A)</td>
<td>At least one of the PIN decimalization tables requested to be activated is empty or already in the active state (not in the loaded state). No PIN tables have been activated.</td>
</tr>
<tr>
<td>8</td>
<td>2139 (85B)</td>
<td>At least one PIN decimalization table provided as input to be activated does not match the corresponding table that is loaded on the coprocessor. No PIN tables have been changed from the loaded state to the active state.</td>
</tr>
<tr>
<td>8</td>
<td>2141 (84D)</td>
<td>The key verification pattern for the key-encrypting key is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>2142 (85E)</td>
<td>A key-usage field setting prevents operation.</td>
</tr>
<tr>
<td>8</td>
<td>2143 (85F)</td>
<td>A key-management field setting prevents operation.</td>
</tr>
<tr>
<td>8</td>
<td>2145 (861)</td>
<td>An attempt to wrap a stronger key with a weaker key was disallowed.</td>
</tr>
<tr>
<td>8</td>
<td>2147 (863)</td>
<td>The key type to be generated is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>2149 (865)</td>
<td>The key to be generated is stronger than the input material.</td>
</tr>
<tr>
<td>8</td>
<td>2151 (867)</td>
<td>At least one PIN decimalization table identifier provided as input is out of range or is a duplicate. No PIN tables have been changed.</td>
</tr>
<tr>
<td>8</td>
<td>2153 (869)</td>
<td>The input token is incompatible with the service (that is, clear key when encrypted key was expected).</td>
</tr>
<tr>
<td>8</td>
<td>2154 (86A)</td>
<td>At least one key token does not have the required key type for the specified function.</td>
</tr>
<tr>
<td>8</td>
<td>2158 (86E)</td>
<td>There is a mismatch between ECC key tokens of curve types, key lengths, or both. Curve types and key lengths must match.</td>
</tr>
<tr>
<td>8</td>
<td>2159 (86F)</td>
<td>A key-encrypting key is invalid.</td>
</tr>
<tr>
<td>8</td>
<td>2161 (871)</td>
<td>A wrap type, either requested or default, is in conflict with one or more input tokens.</td>
</tr>
<tr>
<td>8</td>
<td>3001 (BB9)</td>
<td>The RSA-OAEP block contains a PIN block and the verb did not request PINBLOCK processing.</td>
</tr>
<tr>
<td>8</td>
<td>3002 (BBA)</td>
<td>Specific to IBM System z - UDX already authorized.</td>
</tr>
<tr>
<td>8</td>
<td>3005 (BBD)</td>
<td>Specific to IBM System z - UDX not in UDX Authorization Table (UAT).</td>
</tr>
<tr>
<td>8</td>
<td>3006 (BBE)</td>
<td>Specific to IBM System z - UDX not authorized.</td>
</tr>
<tr>
<td>8</td>
<td>3007 (BBF)</td>
<td>Specific to IBM System z - Failed to obtain semaphore that guards the UAT.</td>
</tr>
<tr>
<td>8</td>
<td>3009 (BC1)</td>
<td>Specific to IBM System z - UDX Password hash mismatch.</td>
</tr>
<tr>
<td>8</td>
<td>3013 (BC5)</td>
<td>The longitudinal redundancy check (LRC) checksum in the AES key-token does not match the LRC checksum of the clear key.</td>
</tr>
<tr>
<td>8</td>
<td>3047 (BE7)</td>
<td>Use of clear key provided is not allowed. A secure key is required.</td>
</tr>
<tr>
<td>8</td>
<td>6000 (1770)</td>
<td>The specified device is already allocated.</td>
</tr>
<tr>
<td>8</td>
<td>6001 (1771)</td>
<td>No device is allocated.</td>
</tr>
<tr>
<td>8</td>
<td>6002 (1772)</td>
<td>The specified device does not exist.</td>
</tr>
<tr>
<td>8</td>
<td>6003 (1773)</td>
<td>The specified device is an improper type.</td>
</tr>
</tbody>
</table>
### Table 147. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6013 (177D)</td>
<td>The length of the cryptographic resource name is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>6014 (177E)</td>
<td>The cryptographic resource name is not valid or does not refer to a coprocessor that is available in the system.</td>
</tr>
<tr>
<td>8</td>
<td>6015 (177F)</td>
<td>An ECC curve type is invalid or its usage is inconsistent.</td>
</tr>
<tr>
<td>8</td>
<td>6017 (1781)</td>
<td>Curve size $p$ is invalid or its usage is inconsistent.</td>
</tr>
<tr>
<td>8</td>
<td>6018 (1782)</td>
<td>Error returned from CLiC module.</td>
</tr>
<tr>
<td>8</td>
<td>10028 (272C)</td>
<td>Specific to IBM System z - Invalid control vector in key token supplied.</td>
</tr>
<tr>
<td>8</td>
<td>10036 (2734)</td>
<td>Specific to IBM System z - Invalid control vectors (L-R) in key token supplied.</td>
</tr>
<tr>
<td>8</td>
<td>10044 (273C)</td>
<td>Specific to IBM System z - The key_type parameter and the CV key type for the supplied key token do not match.</td>
</tr>
<tr>
<td>8</td>
<td>10056 (2748)</td>
<td>Specific to IBM System z - The key_type parameter contains TOKEN, which is invalid for the requested operation.</td>
</tr>
<tr>
<td>8</td>
<td>10124 (278C)</td>
<td>Specific to IBM System z - The key id cannot be exported because of prohibit export restriction in the token supplied.</td>
</tr>
<tr>
<td>8</td>
<td>10128 (2790)</td>
<td>Specific to IBM System z - The NOCV-KEK rule_array keyword does not apply in this case.</td>
</tr>
<tr>
<td>8</td>
<td>10129 (2791)</td>
<td>Specific to IBM System z - The NOCV-KEK importer key or transport key is not allowed in the Remote Key Export operation requested.</td>
</tr>
</tbody>
</table>

### Reason codes that accompany return code 12

Reason codes that accompany return code 12

The codes are listed in Table 148.

### Table 148. Reason codes for return code 12

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>097 (061)</td>
<td>File space in key storage is insufficient to complete the operation.</td>
</tr>
<tr>
<td>12</td>
<td>196 (0C4)</td>
<td>The device driver, the security server, or the directory server is not installed or is not active. File permissions are not valid for your application.</td>
</tr>
<tr>
<td>12</td>
<td>197 (0C5)</td>
<td>There is a key-storage file I/O error or the file is not found.</td>
</tr>
<tr>
<td>12</td>
<td>206 (0CE)</td>
<td>The key-storage file is not valid or the master-key verification failed. There is an unlikely, but possible, synchronization problem with the Master Key Process verb.</td>
</tr>
<tr>
<td>12</td>
<td>207 (0CF)</td>
<td>The verification method flags in the profile are not valid.</td>
</tr>
<tr>
<td>12</td>
<td>319 (13F)</td>
<td>Passed to the CVV Verify or CVV Generate verb, the Verb Unique data corresponds to a PAN length of 19, but the overall length is wrong. This indicates that the host code is out of date.</td>
</tr>
<tr>
<td>12</td>
<td>324 (144)</td>
<td>There is insufficient memory available to process your request, either memory in the host computer or memory inside the coprocessor including the flash EPROM used to store keys, profiles, and other application data.</td>
</tr>
</tbody>
</table>
### Reason codes that accompany return code 12 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>338 (152)</td>
<td>This cryptographic hardware device driver is not installed or is not responding, or the CCA code is not loaded in the coprocessor.</td>
</tr>
<tr>
<td>12</td>
<td>764 (2FC)</td>
<td>The master keys are not loaded and, therefore, a key cannot be recovered or enciphered.</td>
</tr>
<tr>
<td>12</td>
<td>768 (300)</td>
<td>One or more paths for key-storage directory operations are improperly specified.</td>
</tr>
<tr>
<td>12</td>
<td>769 (301)</td>
<td>An internal error has occurred with the parameters to a cryptographic algorithm.</td>
</tr>
<tr>
<td>12</td>
<td>2007 (7D7)</td>
<td>The change type in the Pending Change Buffer is not recognized.</td>
</tr>
<tr>
<td>12</td>
<td>2015 (7DF)</td>
<td>The domain stored in the domain mask does not match what was included as the domain in the CPRB.</td>
</tr>
<tr>
<td>12</td>
<td>2017 (7E1)</td>
<td>The operation is attempting to call ‘SET’ for a master key, but has passed an invalid Master Key Verification Pattern.</td>
</tr>
<tr>
<td>12</td>
<td>2021 (7E5)</td>
<td>The card is disabled in the TKE path.</td>
</tr>
<tr>
<td>12</td>
<td>2037 (7F5)</td>
<td>Invalid domain specified.</td>
</tr>
<tr>
<td>12</td>
<td>2043 (7FB)</td>
<td>In the course of TKE communication through the host library to an adapter, a particular requested OA certificate was not found. A small number of these errors are typical when communication with a TKE is initiated.</td>
</tr>
<tr>
<td>12</td>
<td>2045 (7FD)</td>
<td>The CCA software is unable to claim a semaphore. The system might be short of resources.</td>
</tr>
<tr>
<td>12</td>
<td>2046 (7FE)</td>
<td>The CCA software is unable to list all the keys. The limit of 500,000 keys might have been reached.</td>
</tr>
<tr>
<td>12</td>
<td>2049 (801)</td>
<td>An error occurred while unlocking a semaphore in order to release the exclusive control of that semaphore.</td>
</tr>
<tr>
<td>12</td>
<td>2073 (819)</td>
<td>TKE command received when TKE disabled.</td>
</tr>
<tr>
<td>12</td>
<td>2074 (81A)</td>
<td>Invalid version found in Connectivity Programming Request/Reply Block (CPRB).</td>
</tr>
<tr>
<td>12</td>
<td>2101 (835)</td>
<td>Invalid AES flags in the function control vector (FCV).</td>
</tr>
<tr>
<td>12</td>
<td>2117 (845)</td>
<td>Thread specific CLiC objects are not in proper state.</td>
</tr>
<tr>
<td>12</td>
<td>2155 (86B)</td>
<td>The length of the fully qualified dataset name exceeds the maximum size that the verb can process.</td>
</tr>
<tr>
<td>12</td>
<td>3046 (BE6)</td>
<td>The wrong usage was attempted in an operation with a retained key.</td>
</tr>
</tbody>
</table>

### Reason codes that accompany return code 16

Reason codes that accompany return code 16.

These codes are listed in [Table 149](#).

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>099 (063)</td>
<td>An unrecoverable error occurred in the security server; contact the IBM support center.</td>
</tr>
</tbody>
</table>

Table 149. Reason codes for return code 16
<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>336 (150)</td>
<td>An error occurred in a cryptographic hardware or software component.</td>
</tr>
<tr>
<td>16</td>
<td>337 (151)</td>
<td>A device software error occurred.</td>
</tr>
<tr>
<td>16</td>
<td>339 (153)</td>
<td>A system error occurred in the interprocess communication routine.</td>
</tr>
<tr>
<td>16</td>
<td>444 (1BC)</td>
<td>The verb-unique-data has an invalid length.</td>
</tr>
<tr>
<td>16</td>
<td>556 (22C)</td>
<td>The request parameter block failed consistency checking.</td>
</tr>
<tr>
<td>16</td>
<td>708 (2C4)</td>
<td>The cryptographic engine is returning inconsistent data.</td>
</tr>
<tr>
<td>16</td>
<td>709 (2C5)</td>
<td>Cryptographic engine internal error. Could not access the master-key data.</td>
</tr>
<tr>
<td>16</td>
<td>710 (2C6)</td>
<td>An unrecoverable error occurred while attempting to update master-key data items.</td>
</tr>
<tr>
<td>16</td>
<td>712 (2C8)</td>
<td>An unexpected error occurred in the master-key manager.</td>
</tr>
<tr>
<td>16</td>
<td>800 (320)</td>
<td>A problem occurred in internal SHA operation processing.</td>
</tr>
<tr>
<td>16</td>
<td>2022 (7E6)</td>
<td>TKE-related internal file open error.</td>
</tr>
<tr>
<td>16</td>
<td>2047 (7FF)</td>
<td>Unable to transfer request data from host to coprocessor.</td>
</tr>
<tr>
<td>16</td>
<td>2057 (809)</td>
<td>Internal error: memory allocation failure.</td>
</tr>
<tr>
<td>16</td>
<td>2058 (80A)</td>
<td>Internal error: unexpected return code from OAEP routines.</td>
</tr>
<tr>
<td>16</td>
<td>2059 (80B)</td>
<td>Internal error: OAEP SHA-1 request failure.</td>
</tr>
<tr>
<td>16</td>
<td>2061 (80D)</td>
<td>Internal error in Symmetric Key Import, OAEP-decode: enciphered message too long.</td>
</tr>
<tr>
<td>16</td>
<td>2063 (80F)</td>
<td>The reply message too long for the requestor’s command reply buffer.</td>
</tr>
<tr>
<td>16</td>
<td>2107 (83B)</td>
<td>Internal files failed verification check when loading from encrypted storage.</td>
</tr>
<tr>
<td>16</td>
<td>2150 (866)</td>
<td>An error occurred while attempting to open or save the DECTABLE SRDI that is stored on the coprocessor.</td>
</tr>
</tbody>
</table>
Appendix B. Key token formats

For debugging purposes, this appendix provides the key token formats.

This appendix provides the formats for:

- “AES internal key token”
- “Token Validation Value” on page 610
- “DES internal key token” on page 612
- “DES external key token” on page 613
- “DES null key token” on page 615
- “RSA public key token” on page 615
- “RSA private external key token” on page 616
- “RSA private internal key token” on page 624
- “ECC key token” on page 636
- “PKA null key token” on page 640
- “HMAC key token” on page 640
- “TR-31 optional block data” on page 646
- “Trusted blocks” on page 647

AES internal key token

The format for an AES internal key token.

Table 150 shows the format for an AES internal key token.

CCA AES key-token data structures are 64 bytes in length, and are made up of an internal key-token identifier and a token version number, reserved fields, a flag byte containing various flag bits, and a token-validation value.

Depending on the flag byte, the key token either contains an encrypted key, a clear key, or the key is absent. An encrypted key is encrypted under an AES master key identified by a master-key verification pattern (MKVP) in the key token. The key token contains a two-byte integer that specifies the length of the clear-key value in bits, valued to 0, 128, 192, or 256, and a two-byte integer that specifies the length of the encrypted-key value in bytes, valued to 0 or 32. An LRC checksum byte of the clear-key value is also in the key token.

All AES keys are DATA keys. If the flag byte indicates a control vector (CV) is present, it must be all binary zeros. An all-zero CV represents the CV value of an AES DATA key. If a key is present without a control vector in a key token, that is accepted and the key is interpreted as an AES DATA key. The AES internal key-token is the structure used to hold AES keys that are either encrypted with the AES master-key, or in cleartext format.

Table 150. AES Internal key token format, version X’04’

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X’01’ (flag indicating this is an internal key token)</td>
</tr>
<tr>
<td>1 - 3</td>
<td>Implementation-dependent bytes, must be X’0000000’.</td>
</tr>
</tbody>
</table>
### Key token formats

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Key token version number, X'04'</td>
</tr>
<tr>
<td>5</td>
<td>Reserved (X'00')</td>
</tr>
<tr>
<td>6</td>
<td>Flag byte. See &quot;AES internal key-token flag byte.&quot;</td>
</tr>
<tr>
<td>7</td>
<td>Longitudinal redundancy check (LRC) checksum of clear-key value (LRC is the XOR of each byte in the clear-key value).</td>
</tr>
<tr>
<td>8 - 15</td>
<td>Master key verification pattern (MKVP)</td>
</tr>
<tr>
<td></td>
<td>Contains the master-key verification pattern of the AES master-key used to encrypt the key contained in the token, or binary zeros if the token does not contain a key or the key is in the clear. The MKVP is calculated as the leftmost eight bytes of the SHA-256 hash of the string formed by pre-pending the byte X'01' to the cleartext master-key value.</td>
</tr>
<tr>
<td>16 - 47</td>
<td>Key value, if present. Contains either:</td>
</tr>
<tr>
<td></td>
<td>• A 256-bit encrypted-key value. The clear key value is padded on the right with binary zeros, and the entire 256-bit value is encrypted under the AES master-key using AES CBC mode with an initialization vector of binary zeros.</td>
</tr>
<tr>
<td></td>
<td>• A 128-bit, 192-bit, or 256-bit clear-key value left-aligned and padded on the right with binary zeros for the entire 256-bit field.</td>
</tr>
<tr>
<td>48 - 55</td>
<td>Control Vector (CV)</td>
</tr>
<tr>
<td></td>
<td>This value must be binary zeros for all AES key tokens that have a control vector present.</td>
</tr>
<tr>
<td>56 - 57</td>
<td>Clear-key bit length</td>
</tr>
<tr>
<td></td>
<td>An integer specifying the length in bits of the clear-key value. If no key is present in a completed token, this length is zero. In a skeleton token, this is the length of the key to be created in the token when used as input to the Key Generate verb.</td>
</tr>
<tr>
<td>58 - 59</td>
<td>Encrypted-key byte length</td>
</tr>
<tr>
<td></td>
<td>An integer specifying the length in bytes of the encrypted-key value. This value is zero if the token does not contain a key or the key is in the clear.</td>
</tr>
<tr>
<td>60 - 63</td>
<td>Token validation value (TVV).</td>
</tr>
</tbody>
</table>

### AES internal key-token flag byte

The format for an AES internal key token flag byte.

Table 151 shows the format for an AES internal key token flag byte.

<table>
<thead>
<tr>
<th>Bits (MSB...LSB)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xxxx xxxx</td>
<td>Key is encrypted under the AES master-key (ignored if no key present).</td>
</tr>
<tr>
<td>0xxxx xxxx</td>
<td>Key is in the clear (ignored if no key present).</td>
</tr>
<tr>
<td>x1xx xxxx</td>
<td>Control vector (CV) is present.</td>
</tr>
<tr>
<td>xx1x xxxx</td>
<td>No key and no MKVP present.</td>
</tr>
<tr>
<td>xx0x xxxx</td>
<td>Encrypted or clear key present, MKVP present if key is encrypted.</td>
</tr>
</tbody>
</table>

**Note:** All undefined bits are reserved and must be 0.

### Token Validation Value

CCA uses the Token Validation Value (TVV) to verify that a token is valid.
Key token formats

The TVV prevents a key token that is not valid or that is overlaid from being accepted by CCA. It provides a checksum to detect a corruption in the key token.

When an CCA verb generates a key token, it generates a TVV and stores the TVV in bytes 60-63 of the key token. When an application program passes a key token to a verb, CCA checks the TVV. To generate the TVV, CCA performs a two's complement ADD operation (ignoring carries and overflow) on the key token, operating on four bytes at a time, starting with bytes 0-3 and ending with bytes 56-59.

Format of the clear key token

The format for a clear internal key token.

Table 152 shows the format for a clear internal key token.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X'01' (flag indicating this is an internal key token)</td>
</tr>
<tr>
<td>1 - 3</td>
<td>Implementation-dependent bytes (X'000000' for ICSF)</td>
</tr>
<tr>
<td>4</td>
<td>Key token version number (X'00' or X'01')</td>
</tr>
<tr>
<td>5</td>
<td>Reserved (X'00')</td>
</tr>
<tr>
<td>6</td>
<td>Flag byte</td>
</tr>
<tr>
<td></td>
<td><strong>Bit</strong></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2 - 7</td>
<td></td>
</tr>
<tr>
<td>7 - 15</td>
<td>Reserved (X'00')</td>
</tr>
<tr>
<td>16 - 23</td>
<td>A single-length key, the left half of a double-length key, or Part A of a triple-length key.</td>
</tr>
<tr>
<td>24 - 31</td>
<td>X'0000000000000000' if a single-length key, the right half of a double-length operational key, or Part B of a triple-length operational key.</td>
</tr>
<tr>
<td>32 - 47</td>
<td>Reserved for clear key tokens (X'00's)</td>
</tr>
<tr>
<td>48 - 55</td>
<td>X'0000000000000000' if a single-length key or double-length key, or Part C of a triple-length operational key.</td>
</tr>
<tr>
<td>56 - 58</td>
<td>Reserved (X'0000000')</td>
</tr>
<tr>
<td>59 bits 0 and 1</td>
<td>B'00' reserved</td>
</tr>
<tr>
<td>59 bits 2 and 3</td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td>B'00'</td>
</tr>
<tr>
<td></td>
<td>B'01'</td>
</tr>
<tr>
<td></td>
<td>B'10'</td>
</tr>
<tr>
<td>59 bits 4 - 7</td>
<td>B'0000'</td>
</tr>
<tr>
<td>60 - 63</td>
<td>Token validation value (TVV).</td>
</tr>
</tbody>
</table>
Key token formats

**DES internal key token**

The format for a DES internal key token.

Table 153 shows the format for a DES internal key token.

<table>
<thead>
<tr>
<th>Table 153. DES internal key token format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1-3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8-15</td>
</tr>
<tr>
<td>16-23</td>
</tr>
<tr>
<td>24-31</td>
</tr>
<tr>
<td>32-39</td>
</tr>
<tr>
<td>40-47</td>
</tr>
<tr>
<td>48-55</td>
</tr>
<tr>
<td>56-58</td>
</tr>
<tr>
<td>59 bits 0 and 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>59 bits 2 and 3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>59 bits 4-7</td>
</tr>
<tr>
<td>60-63</td>
</tr>
</tbody>
</table>

**Note:** AKEKs are not supported by this version of CCA. Key tokens from other CCA systems, however, could have the AKEK flag bits set in a key token.
### DES external key token

The format for a DES external key token.

Table 154 shows the format for a DES external key token.

#### Table 154. DES external key token format

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X'02' (flag indicating an external key token)</td>
</tr>
<tr>
<td>1</td>
<td>Reserved (X'00')</td>
</tr>
<tr>
<td>2 - 3</td>
<td>Implementation-dependent bytes (X'0000' for CCA)</td>
</tr>
<tr>
<td>4</td>
<td>Key token version number (X'00' or X'01')</td>
</tr>
<tr>
<td>5</td>
<td>Reserved (X'00')</td>
</tr>
<tr>
<td>6</td>
<td>Flag byte</td>
</tr>
<tr>
<td></td>
<td><strong>Bit</strong></td>
</tr>
<tr>
<td>0</td>
<td><strong>Meaning When Set On</strong></td>
</tr>
<tr>
<td>0</td>
<td>Encrypted key is present.</td>
</tr>
<tr>
<td>1</td>
<td>Control vector (CV) value has been applied to the key.</td>
</tr>
<tr>
<td></td>
<td>Other bits are reserved and are binary zeros.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved (X'00')</td>
</tr>
<tr>
<td>8 - 15</td>
<td>Reserved (X'0000000000000000000000')</td>
</tr>
<tr>
<td>16 - 23</td>
<td>Single-length key or left half of a double-length key, or Part A of a triple-length key. The value is encrypted under a transport key.</td>
</tr>
<tr>
<td>24 - 31</td>
<td>X'00000000000000000000000' if a single-length key or right half of a double-length key, or Part B of a triple-length key. The right half of a double-length key or Part B of a triple-length key is encrypted under a transport (key-encrypting key) for export or import.</td>
</tr>
<tr>
<td>32 - 39</td>
<td>Control vector (CV) for single-length key or left half of CV for double-length key</td>
</tr>
<tr>
<td>40 - 47</td>
<td>X'00000000000000000000000' if single-length key or right half of CV for double-length key</td>
</tr>
<tr>
<td>48 - 55</td>
<td>X'00000000000000000000000' if a single-length key, double-length key, or Part C of a triple-length key.</td>
</tr>
<tr>
<td>56 - 58</td>
<td>Reserved (X'000000')</td>
</tr>
<tr>
<td>59 bits 0 and 1</td>
<td>B'00'</td>
</tr>
<tr>
<td>59 bits 2 and 3</td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>B'00'</td>
<td>Indicates single-length key (version 0 only).</td>
</tr>
<tr>
<td>B'01'</td>
<td>Indicates double-length key (version 1 only).</td>
</tr>
<tr>
<td>B'10'</td>
<td>Indicates triple-length key (version 1 only).</td>
</tr>
<tr>
<td>59 bits 4 - 7</td>
<td>B'0000'</td>
</tr>
<tr>
<td>60 - 63</td>
<td>Token validation value (see &quot;Token Validation Value&quot; on page 610 for a description).</td>
</tr>
</tbody>
</table>

### External RKX DES key tokens

This topic defines an external fixed-length DES key-token called an **RKX key-token**.

Table 155 on page 614 defines an external fixed-length DES key-token called an **RKX key-token**. An RKX key-token is a special token used exclusively by the Remote Key Export (CSNDRKX) verb and DES key-storage verbs (for example, DES Key Record Write). No other verbs use or reference an RKX key-token or key-token record. For additional information about the usage of RKX key tokens,
Key token formats

see “Remote key loading” on page 46. Verbs other than Remote Key Export and the DES key-storage do not support RKX key tokens or RKX key token records.

As can be seen in the table, RKX key tokens are 64 bytes in length, have a token identifier flag (X'02'), a token version number (X'10'), and room for encrypted keys, same as normal fixed-length DES key tokens. Unlike normal fixed-length DES key tokens, RKX key tokens do not have a control vector, flag bits, and a token-validation value. In addition, RKX key tokens have a confounder value, a MAC value, and room for a third encrypted key.

Table 155. External RKX DES key-token format, version X'10'

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>X'02' (a token identifier flag that indicates an external key-token)</td>
</tr>
<tr>
<td>01</td>
<td>3</td>
<td>Reserved, binary zero</td>
</tr>
<tr>
<td>04</td>
<td>1</td>
<td>Token version number (X'10')</td>
</tr>
<tr>
<td>05</td>
<td>2</td>
<td>Reserved, binary zero</td>
</tr>
<tr>
<td>07</td>
<td>1</td>
<td>Key length in bytes, including confounder</td>
</tr>
<tr>
<td>08</td>
<td>8</td>
<td>Confounder</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>Key left</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>Key middle (binary zero if not used)</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>Key right (binary zero if not used)</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>Rule ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The trusted block rule identifier used to create this key token. A subsequent call to Remote Key Export (CSNDRKX) can use this token with a trusted block rule that references the rule ID that was used to create this token. The trusted block rule can be compared with this rule ID for verification purposes. The Rule ID is an 8-byte string of ASCII characters, left-aligned and padded on the right with space characters. Acceptable characters are A...Z, a...z, 0...9, - (X'2D'), and _ (X'5F'). All other characters are reserved for future use.</td>
</tr>
<tr>
<td>48</td>
<td>8</td>
<td>Reserved, binary zero</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>MAC value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISO 16609 CBC-mode Triple-DES MAC, computed over the 56 bytes starting at offset 0 and including the encrypted key value and the rule ID using the same MAC key that is used to protect the trusted block itself. This MAC value guarantees that the key and the rule ID cannot be modified without detection, providing integrity and binding the rule ID to the key itself. This MAC value must verify with the same trusted block used to create the key, thus binding the key structure to that specific trusted block.</td>
</tr>
</tbody>
</table>

Note:

1. A fixed, randomly derived variant is exclusive-ORed with the MAC key before it is used to encipher the generated or exported key and confounder.
2. The MAC key is located within a trusted block (internal format) and can be recovered by decipherment under a variant of the PKA master key.
3. The trusted block is originally created in external form by the Trusted Block Create verb, and then converted to internal form by the PKA Key Import verb prior to the Remote Key Export call.
## Key token formats

### DES null key token

The format for a DES null key token.

Table 156 shows the format for a DES null key token.

**Table 156. DES null key token format**

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X'00' (flag indicating this is a null key token).</td>
</tr>
<tr>
<td>1 - 15</td>
<td>Reserved (set to binary zeros).</td>
</tr>
<tr>
<td>16 - 23</td>
<td>Single-length encrypted key, left half of double-length encrypted key, or Part A of triple-length encrypted key.</td>
</tr>
<tr>
<td>24 - 31</td>
<td>X'0000000000000000' if a single-length encrypted key, the right half of double-length encrypted key, or Part B of triple-length encrypted key.</td>
</tr>
<tr>
<td>32 - 39</td>
<td>X'0000000000000000' if a single-length encrypted key or double-length encrypted key.</td>
</tr>
<tr>
<td>40 - 47</td>
<td>Reserved (set to binary zeros).</td>
</tr>
<tr>
<td>48 - 55</td>
<td>Part C of a triple-length encrypted key.</td>
</tr>
<tr>
<td>56 - 63</td>
<td>Reserved (set to binary zeros).</td>
</tr>
</tbody>
</table>

### RSA public key token

The sections of an RSA public key token.

An RSA public key token contains the following sections.

- A required token header, starting with the token identifier X'1E'
- A required RSA public key section, starting with the section identifier X'04'

Table 157 presents the format of an RSA public key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format).

**Table 157. RSA Public Key Token format**

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token Header (Required)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 001</td>
<td>Token identifier. X'1E' indicates an external token.</td>
<td></td>
</tr>
<tr>
<td>001 001</td>
<td>Version, X'00'.</td>
<td></td>
</tr>
<tr>
<td>002 002</td>
<td>Length of the key token structure.</td>
<td></td>
</tr>
<tr>
<td>004 004</td>
<td>Ignored. Should be 0.</td>
<td></td>
</tr>
<tr>
<td><strong>RSA Public Key Section (Required)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 001</td>
<td>X'04', section identifier, RSA public key.</td>
<td></td>
</tr>
<tr>
<td>001 001</td>
<td>X'00', version.</td>
<td></td>
</tr>
<tr>
<td>002 002</td>
<td>Section length, 12 + xxx + yyy</td>
<td></td>
</tr>
<tr>
<td>004 002</td>
<td>Reserved field.</td>
<td></td>
</tr>
<tr>
<td>006 002</td>
<td>RSA public key exponent field length in bytes, “xxx”.</td>
<td></td>
</tr>
<tr>
<td>008 002</td>
<td>Public key modulus length in bits.</td>
<td></td>
</tr>
<tr>
<td>010 002</td>
<td>RSA public key modulus field length in bytes, “yyy”.</td>
<td></td>
</tr>
</tbody>
</table>
Table 157. RSA Public Key Token format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>012</td>
<td>xxx</td>
<td>Public key exponent (this is generally a 1, 3, or 64 - 256-byte quantity), named e. e must be odd and 1 &lt; e &lt; n. (Frequently, the value of e is 2 ** 16 + 1). <strong>Note:</strong> You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation does not generate an RSA key with a public exponent valued to two (a Rabin key).</td>
</tr>
<tr>
<td>12 + xxx</td>
<td>yyy</td>
<td>Modulus, n.</td>
</tr>
</tbody>
</table>

RSA private external key token

The sections of an RSA private external key.

An RSA private external key token contains the following sections:

- A required PKA token header starting with the token identifier X'1E'.
- A required RSA private key section starting with one of the section identifiers shown in Table 158.
- A required RSA public key section, starting with the section identifier X'04'.
- An optional private key name section, starting with the section identifier X'10'.

Table 158 presents the basic record format of an RSA private external key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-aligned and padded with zeros to the left.

Table 158. RSA private external key token basic record format

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Token identifier. X'1E' indicates an external token. The private key is either in cleartext or enciphered with a transport key-encrypting key.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Version, X'00'.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the key token structure.</td>
</tr>
<tr>
<td>004</td>
<td>004</td>
<td>Ignored. Should be zero.</td>
</tr>
</tbody>
</table>

RSA Private Key Section (Required)

See the following sections:

- “RSA private key token, 1024-bit Modulus-Exponent external format” on page 617
- “RSA private key token, 4096-bit Modulus-Exponent external format” on page 618
- “RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section external form” on page 619
- “RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section external form” on page 621
- “RSA Private Key Token, 4096-bit Chinese Remainder Theorem External Form” on page 623

RSA Public Key Section (Required)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X'04', section identifier, RSA public key.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X'00', version.</td>
</tr>
</tbody>
</table>
### Key token formats

#### Table 158. RSA private external key token basic record format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>002</td>
<td>002</td>
<td>Section length, 12 + xxx.</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Reserved field.</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>RSA public key exponent field length in bytes, “xxx”.</td>
</tr>
<tr>
<td>008</td>
<td>002</td>
<td>Public key modulus length in bits.</td>
</tr>
<tr>
<td>010</td>
<td>002</td>
<td>RSA public key modulus field length in bytes, which is zero for a private token. <strong>Note</strong>: In an RSA private key token, this field should be zero. The RSA private key section contains the modulus.</td>
</tr>
<tr>
<td>012</td>
<td>xxx</td>
<td>Public key exponent, e (this is generally a 1, 3, or 64 - 256-byte quantity). e must be odd and 1 &lt; e &lt; n. (Frequently, the value of e is 2^16 + 1 (= 65,537). <strong>Note</strong>: You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation does not generate an RSA key with a public exponent valued to two (a Rabin key).</td>
</tr>
</tbody>
</table>

**Private Key Name** (Optional)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X’10’, section identifier, private key name.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X’00’, version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length, X’0044’ (68 decimal).</td>
</tr>
<tr>
<td>004</td>
<td>064</td>
<td>Private key name (in ASCII), left-aligned, padded with space characters (X’20’). An access control system can use the private key name to verify the calling application is entitled to use the key.</td>
</tr>
</tbody>
</table>

### RSA private key token, 1024-bit Modulus-Exponent external format

This RSA private key token and the external X’02’ token is supported starting with CEX3C.

Table 159 shows the format.

#### Table 159. RSA private key token, 1024-bit Modulus-Exponent external format

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X’02’, section identifier, RSA private key, Modulus-Exponent format (RSA-PRIV)</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X’00’, version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the RSA private key section X’016C’ (364 decimal).</td>
</tr>
<tr>
<td>004</td>
<td>020</td>
<td>SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.</td>
</tr>
<tr>
<td>024</td>
<td>004</td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>028</td>
<td>001</td>
<td>Key format and security: <strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’00’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’82’</td>
</tr>
<tr>
<td>029</td>
<td>001</td>
<td>Reserved, binary zero.</td>
</tr>
</tbody>
</table>
Key token formats

Table 159. RSA private key token, 1024-bit Modulus-Exponent external format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>030</td>
<td>020</td>
<td>SHA-1 hash of the optional key-name section. If there is no key-name section, then 20 bytes of X'00'.</td>
</tr>
</tbody>
</table>
| 050              | 004             | Key use flag bits.  
B'11xx xxxx'  
Only key unwrapping (KM-ONLY)  
B'10xx xxxx'  
Both signature generation and key unwrapping (KEY-MGMT)  
B'01xx xxxx'  
Undefined  
B'00xx xxxx'  
Only signature generation (SIG-ONLY)  
All other bits reserved, set to binary zero. |
| 054              | 006             | Reserved; set to binary zero. |
| 060              | 024             | Reserved; set to binary zero. |
| 084              | 024             | Start of the optionally-encrypted secure subsection. |
| 084              | 024             | Random number, confounder. |
| 108              | 128             | Private-key exponent, d. d = e^{-1} \mod((p-1)(q-1)), and 1 < d < n where e is the public exponent.  
End of the optionally-encrypted subsection; the confounder field and the private-key exponent field are enciphered for key confidentiality when the key format and security flags (offset 28) indicate the private key is enciphered. They are enciphered under a double-length transport key using the ede2 algorithm. |
| 236              | 128             | Modulus, n. n = pq where p and q are prime and 1 < n < 2^{1024}. |

RSA private key token, 4096-bit Modulus-Exponent external form

This RSA private key token and the external X'09' token is supported on a CCA Crypto Express coprocessor.

Table 160. RSA Private Key Token, 4096-bit Modulus-Exponent External Format

<table>
<thead>
<tr>
<th>Offset (Dec)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X'09', section identifier, RSA private key, modulus-exponent format (RSAMEVAR).</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X'00', version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the RSA private key section 132+ddd+nnn+xxx.</td>
</tr>
<tr>
<td>004</td>
<td>020</td>
<td>SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.</td>
</tr>
<tr>
<td>024</td>
<td>002</td>
<td>Length of the encrypted private key section 8+ddd+xxx.</td>
</tr>
<tr>
<td>026</td>
<td>002</td>
<td>Reserved; set to binary zero.</td>
</tr>
</tbody>
</table>
| 028          | 001             | Key format and security:  
X'00' Unencrypted RSA private key subsection identifier.  
X'82' Encrypted RSA private key subsection identifier. |
| 029          | 001             | Reserved, set to binary zero. |
### TABLE 160. RSA PRIVATE KEY, 4096-BIT MODULUS-EXONENT EXTERNAL FORM (CONTINUED)

<table>
<thead>
<tr>
<th>Offset (Dec)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>030 020</td>
<td></td>
<td>SHA-1 hash of the optional key-name section. If there is no key-name section, then 20 bytes of X’00’.</td>
</tr>
<tr>
<td>050 001</td>
<td></td>
<td>Key use flag bits.</td>
</tr>
<tr>
<td></td>
<td>Bit</td>
<td>Meaning When Set On</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Key management usage permitted.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Signature usage not permitted.</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>The key is translatable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other bits reserved, set to binary zero.</td>
</tr>
<tr>
<td>051 001</td>
<td></td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>052 048</td>
<td></td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>100 016</td>
<td></td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>116 002</td>
<td></td>
<td>Length of private exponent, ( d ), in bytes: ddd.</td>
</tr>
<tr>
<td>118 002</td>
<td></td>
<td>Length of modulus, ( n ), in bytes: nnn.</td>
</tr>
<tr>
<td>120 002</td>
<td></td>
<td>Length of padding field, in bytes: xxx.</td>
</tr>
<tr>
<td>122 002</td>
<td></td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>124 008</td>
<td></td>
<td>Start of the optionally-encrypted secure subsection.</td>
</tr>
<tr>
<td>132 ddd</td>
<td></td>
<td>Private-key exponent, ( d = e^{-1} \mod((p-1)(q-1)) ), and ( 1&lt;d&lt;n ) where ( e ) is the public exponent.</td>
</tr>
<tr>
<td>132+ddd.xxx</td>
<td></td>
<td>X’00’ padding of length xxx bytes such that the length from the start of the random number above to the end of the padding field is a multiple of eight bytes.</td>
</tr>
<tr>
<td>132+ddd+xxx</td>
<td>nnn</td>
<td>End of the optionally-encrypted subsection; the confounder field and the private-key exponent field are encrypted for key confidentiality when the key format and security flags (offset 28) indicate that the private key is encrypted. They are encrypted under a double-length transport key using the ede2 algorithm.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modulus, ( n = pq ) where ( p ) and ( q ) are prime and ( 1&lt;n&lt;2^{4096} ).</td>
</tr>
</tbody>
</table>

### RSA PRIVATE KEY, 4096-BIT MODULUS-EXONENT FORMAT WITH AES ENCRYPTED OPK SECTION EXTERNAL FORM

This RSA private key token is supported on the Crypto Express3 Coprocessor and Crypto Express4 Coprocessor.

### TABLE 161. RSA PRIVATE KEY, 4096-BIT MODULUS-EXONENT FORMAT WITH AES ENCRYPTED OPK SECTION (X’30’) EXTERNAL FORM

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 001</td>
<td></td>
<td>Section identifier:</td>
</tr>
<tr>
<td>X’30’</td>
<td></td>
<td>RSA private key, ME format with AES encrypted OPK.</td>
</tr>
<tr>
<td>001 001</td>
<td></td>
<td>Section version number (X’00’).</td>
</tr>
<tr>
<td>002 002</td>
<td></td>
<td>Section length: 122 + nnn + ppp</td>
</tr>
<tr>
<td>004 002</td>
<td></td>
<td>Length of “Associated Data” section</td>
</tr>
<tr>
<td>006 002</td>
<td></td>
<td>Length of payload data: ppp</td>
</tr>
</tbody>
</table>
### Key token formats

**Table 161. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (\textit{X'30'}) external form (continued)**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>008</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>010</td>
<td>001</td>
<td>Start of Associated Data</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>Associated Data Version:</td>
</tr>
<tr>
<td></td>
<td>X'02'</td>
<td>Version 2</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>Key format and security flag:</td>
</tr>
<tr>
<td></td>
<td>X'00'</td>
<td>Unencrypted ME RSA private-key subsection identifier</td>
</tr>
<tr>
<td></td>
<td>X'82'</td>
<td>Encrypted ME RSA private-key subsection identifier</td>
</tr>
<tr>
<td>012</td>
<td>001</td>
<td>Key source flag:</td>
</tr>
<tr>
<td></td>
<td>Reserved, binary zero.</td>
<td></td>
</tr>
<tr>
<td>013</td>
<td>001</td>
<td>Reserved, binary zeroes.</td>
</tr>
<tr>
<td>014</td>
<td>001</td>
<td>Hash type:</td>
</tr>
<tr>
<td></td>
<td>X'00'</td>
<td>Clear key</td>
</tr>
<tr>
<td></td>
<td>X'02'</td>
<td>SHA-256</td>
</tr>
<tr>
<td>015</td>
<td>032</td>
<td>SHA-256 hash of all optional sections that follow the public key section, if any; else 32 bytes of X'00'.</td>
</tr>
<tr>
<td>047</td>
<td>003</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>050</td>
<td>001</td>
<td>Key-usage flag:</td>
</tr>
<tr>
<td></td>
<td>B'11xx xxxx'</td>
<td>Only key unwrapping (KM-ONLY)</td>
</tr>
<tr>
<td></td>
<td>B'10xx xxxx'</td>
<td>Both signature generation and key unwrapping (KEY-MGMT)</td>
</tr>
<tr>
<td></td>
<td>B'01xx xxxx'</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td>B'00xx xxxx'</td>
<td>Only signature generation (SIG-ONLY)</td>
</tr>
<tr>
<td></td>
<td>Translation control:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B'xxxx xx1x'</td>
<td>Private key translation is allowed (XIMATE-OK)</td>
</tr>
<tr>
<td></td>
<td>B'xxxx xx0x'</td>
<td>Private key translation is not allowed (NO-XIMATE)</td>
</tr>
<tr>
<td>051</td>
<td>001</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>052</td>
<td>002</td>
<td>Length of modulus: nnn bytes</td>
</tr>
<tr>
<td>054</td>
<td>002</td>
<td>Length of private exponent: ddd bytes</td>
</tr>
<tr>
<td></td>
<td>End of Associated Data</td>
<td></td>
</tr>
<tr>
<td>056</td>
<td>048</td>
<td>16 byte confounder + 32-byte Object Protection Key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPK used as an AES key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encrypted with an AES KEK.</td>
</tr>
</tbody>
</table>
Table 161. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') external form (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>016</td>
<td>Key verification pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For an encrypted private key, KEK verification pattern (KVP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a clear private key, binary zeros</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a skeleton, binary zeros</td>
</tr>
<tr>
<td>120</td>
<td>002</td>
<td>Reserved, binary zeros.</td>
</tr>
<tr>
<td>122</td>
<td>nnn</td>
<td>Modulus</td>
</tr>
<tr>
<td>122+nnn</td>
<td>ppp</td>
<td>Payload starts here and includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When this section is unencrypted:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Clear private exponent d.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Length ppp bytes : ddd + 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When this section is encrypted:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Private exponent d within the AESKW-wrapped payload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Length ppp bytes : ddd + AESKW format overhead</td>
</tr>
</tbody>
</table>

RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section external form

This RSA private key token is supported on the Crypto Express3 Coprocessor and Crypto Express4 Coprocessor.

Table 162. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') external form

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'31' RSA private key, CRT format with AES encrypted OPK</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length: 134 + nnn + xxx</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Length of “Associated Data” section</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>Length of payload data: xxx</td>
</tr>
<tr>
<td>008</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Start of Associated Data</td>
</tr>
<tr>
<td>010</td>
<td>001</td>
<td>Associated Data Version:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'03' Version 3</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>Key format and security flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'40' Unencrypted RSA private-key subsection identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'42' Encrypted RSA private-key subsection identifier</td>
</tr>
<tr>
<td>012</td>
<td>001</td>
<td>Key source flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>013</td>
<td>001</td>
<td>Reserved, binary zeroes.</td>
</tr>
</tbody>
</table>
### Key token formats

Table 162. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X’31’)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>014</td>
<td>001</td>
<td>Hash type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’00’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’01’</td>
</tr>
<tr>
<td>015</td>
<td>032</td>
<td>SHA-256 hash of all optional sections that follow the public key section, if any; else 32 bytes of X’00’.</td>
</tr>
<tr>
<td>047</td>
<td>003</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>050</td>
<td>001</td>
<td>Key-usage flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’11xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’10xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’01xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’00xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translation control:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’xxxx xx1x’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’xxxx xx0x’</td>
</tr>
<tr>
<td>051</td>
<td>001</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>052</td>
<td>002</td>
<td>Length of the prime number, p, in bytes: ppp.</td>
</tr>
<tr>
<td>054</td>
<td>002</td>
<td>Length of the prime number, q, in bytes: qqq</td>
</tr>
<tr>
<td>056</td>
<td>002</td>
<td>Length of dp : rrr.</td>
</tr>
<tr>
<td>058</td>
<td>002</td>
<td>Length of dq : sss.</td>
</tr>
<tr>
<td>060</td>
<td>002</td>
<td>Length of U: uuu.</td>
</tr>
<tr>
<td>062</td>
<td>002</td>
<td>Length of modulus, nnn.</td>
</tr>
<tr>
<td>064</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>066</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End of Associated Data</td>
</tr>
<tr>
<td>068</td>
<td>048</td>
<td>16 byte confounder + 32-byte Object Protection Key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPK used as an AES key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>External tokens:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encrypted with an AES KEK.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Internal tokens:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encrypted with the ECC master key.</td>
</tr>
<tr>
<td>116</td>
<td>016</td>
<td>Key verification pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For an encrypted private key, KEK verification pattern (KVP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a clear private key, binary zeros</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a skeleton, binary zeros</td>
</tr>
</tbody>
</table>
### Key token formats

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>002</td>
<td>Reserved, binary zeros</td>
</tr>
<tr>
<td>134</td>
<td>nnn</td>
<td>Modulus, ( n, n=pq ), where ( p ) and ( q ) are prime.</td>
</tr>
</tbody>
</table>
| 134+nnn        | xxx            | Payload starts here and includes:  
  When this section is unencrypted:  
  - Clear prime number \( p \)  
  - Clear prime number \( q \)  
  - Clear \( dp \)  
  - Clear \( dq \)  
  - Clear \( U \)  
  - Length xxx bytes: \( ppp + qqq + rrr + sss + uuu + 0 \)  
  When this section is encrypted:  
  - prime number \( p \)  
  - prime number \( q \)  
  - \( dp \)  
  - \( dq \)  
  - \( U \)  
  - within the AESKW-wrapped payload.  
  Length xxx bytes: \( ppp + qqq + rrr + sss + uuu + AESKW \) format overhead |

### RSA Private Key Token, 4096-bit Chinese Remainder Theorem External Form

This RSA private key token with up to 2048-bit modulus is supported on all coprocessors. The modulus size is increased to 4096-bit on the z9 EC, z9 BC, z10 EC, z10 BC, or later machines with the Nov. 2007 or later version of the licensed internal code installed on the CCA Crypto Express coprocessor.

<table>
<thead>
<tr>
<th>Offset (Dec)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X’08’, section identifier, RSA private key, CRT format (RSA-CRT)</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X’00’, version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the RSA private-key section, 132 + ( ppp + qqq + rrr + sss + uuu + xxx + nnn ).</td>
</tr>
<tr>
<td>004</td>
<td>020</td>
<td>SHA-1 hash value of the private key subsection cleartext, offset 28 to the end of the modulus.</td>
</tr>
<tr>
<td>024</td>
<td>004</td>
<td>Reserved; set to binary zero.</td>
</tr>
</tbody>
</table>
| 028          | 001             | Key format and security:  
  - X’40’ Unencrypted RSA private-key subsection identifier, Chinese Remainder form.  
  - X’42’ Encrypted RSA private-key subsection identifier, Chinese Remainder form. |
| 029          | 001             | Reserved; set to binary zero. |
| 030          | 020             | SHA-1 hash of the optional key-name section and any following optional sections. If there are no optional sections, then 20 bytes of X’00’. |
### Key token formats

#### Table 163. RSA Private Key Token, 4096-bit Chinese Remainder Theorem External Format (continued)

<table>
<thead>
<tr>
<th>Offset (Dec)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>050</td>
<td>004</td>
<td>Key use flag bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Bit</strong> <strong>Meaning When Set On</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Key management usage permitted.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Signature usage not permitted.</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>The key is translatable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other bits reserved, set to binary zero.</td>
</tr>
<tr>
<td>054</td>
<td>002</td>
<td>Length of prime number, p, in bytes: ppp.</td>
</tr>
<tr>
<td>056</td>
<td>002</td>
<td>Length of prime number, q, in bytes: qqq.</td>
</tr>
<tr>
<td>058</td>
<td>002</td>
<td>Length of d_p, in bytes: rrr.</td>
</tr>
<tr>
<td>060</td>
<td>002</td>
<td>Length of d_q, in bytes: sss.</td>
</tr>
<tr>
<td>062</td>
<td>002</td>
<td>Length of U, in bytes: uuu.</td>
</tr>
<tr>
<td>064</td>
<td>002</td>
<td>Length of modulus, n, in bytes: nnn.</td>
</tr>
<tr>
<td>066</td>
<td>004</td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>070</td>
<td>002</td>
<td>Length of padding field, in bytes: xxx.</td>
</tr>
<tr>
<td>072</td>
<td>004</td>
<td>Reserved, set to binary zero.</td>
</tr>
<tr>
<td>076</td>
<td>016</td>
<td>Reserved, set to binary zero.</td>
</tr>
<tr>
<td>092</td>
<td>032</td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td>Start of the optionally-encrypted secure subsection.</td>
</tr>
<tr>
<td>124</td>
<td>008</td>
<td>Random number, confounder.</td>
</tr>
<tr>
<td>132</td>
<td>ppp</td>
<td>Prime number, p.</td>
</tr>
<tr>
<td>132 + ppp</td>
<td>qqq</td>
<td>Prime number, q</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr</td>
<td>d_p = d mod(p - 1)</td>
<td></td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr</td>
<td>d_q = d mod(q - 1)</td>
<td></td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss</td>
<td>U = q**-1 mod(p).</td>
<td></td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu</td>
<td>xxx</td>
<td>X'00' padding of length xxx bytes such that the length from the start of the random number above to the end of the padding field is a multiple of eight bytes.</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu + xxx</td>
<td>nnn</td>
<td>Modulus, n. n = pq where p and q are prime and 1**&lt;n&lt;2**2048.</td>
</tr>
</tbody>
</table>

### RSA private internal key token

The sections of the RSA private internal key.

An RSA private internal key token contains the following sections:

- A required PKA token header, starting with the token identifier X'1F'
Key token formats

- Basic record format of an RSA private internal key token. All length fields are in binary. All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, S/390 format). All binary fields (exponents, modulus, and so on) in the private sections of tokens are right-aligned and padded with zeros to the left.

Table 164 shows the format.

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token Header (Required)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>Token identifier. X'1F' indicates an internal token. The private key is enciphered with a PKA master key.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Version, X'00'.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the key token structure excluding the internal information section.</td>
</tr>
<tr>
<td>004</td>
<td>004</td>
<td>Ignored; should be zero.</td>
</tr>
</tbody>
</table>

**RSA Private Key Section and Secured Subsection (Required)**

See the following sections:

- “RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature” on page 626.
- “RSA private key token, 1024-bit Modulus-Exponent internal format starting with CEX3C” on page 627.
- “RSA private key token, 4096-bit Modulus-Exponent internal format” on page 628.
- “RSA private key token, 4096-bit Modulus-Exponent internal format starting with CEX3C” on page 629.
- “RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section internal form” on page 630.
- “RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section internal form” on page 632.
- “RSA Private Key Token, 4096-bit Chinese Remainder Theorem Internal Form” on page 634.

**RSA Public Key Section (Required)**

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X'04', section identifier, RSA public key.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X'00', version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length, 12 + xxx.</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Reserved field.</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>RSA public key exponent field length in bytes, “xxx”.</td>
</tr>
<tr>
<td>008</td>
<td>002</td>
<td>Public key modulus length in bits.</td>
</tr>
<tr>
<td>010</td>
<td>002</td>
<td>RSA public key modulus field length in bytes, which is zero for a private token.</td>
</tr>
</tbody>
</table>
| 012 | xxx | Public key exponent (this is generally a 1, 3, or 64 - 256-byte quantity), e. e must be odd and 1 < e < n. (Frequently, the value of e is 2^16 + 1 (= 65,537).

**Note:** You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation does not generate an RSA key with a public exponent valued to two (a Rabin key). |

**Private Key Name (Optional)**

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X'10', section identifier, private key name.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X'00', version.</td>
</tr>
</tbody>
</table>
### Key token formats

Table 164. RSA private internal key token basic record format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>002</td>
<td>002</td>
<td>Section length, X’0044‘ (68 decimal).</td>
</tr>
<tr>
<td>004</td>
<td>064</td>
<td>Private key name (in ASCII), left-aligned, padded with space characters (X’20’). An access control system can use the private key name to verify the calling application is entitled to use the key.</td>
</tr>
</tbody>
</table>

**Internal Information Section (Required)**

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>004</td>
<td>Eye catcher ‘PKTN’.</td>
</tr>
<tr>
<td>004</td>
<td>004</td>
<td>PKA token type.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning When Set On</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RSA key.</td>
</tr>
<tr>
<td>2</td>
<td>Private key.</td>
</tr>
<tr>
<td>3</td>
<td>Public key.</td>
</tr>
<tr>
<td>4</td>
<td>Private key name section exists.</td>
</tr>
<tr>
<td>5</td>
<td>Private key unenciphered.</td>
</tr>
<tr>
<td>6</td>
<td>Blinding information present.</td>
</tr>
<tr>
<td>7</td>
<td>Retained private key.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>008</td>
<td>004</td>
<td>Address of token header.</td>
</tr>
<tr>
<td>012</td>
<td>002</td>
<td>Total length of total structure including this information section.</td>
</tr>
<tr>
<td>014</td>
<td>002</td>
<td>Count of number of sections.</td>
</tr>
<tr>
<td>016</td>
<td>016</td>
<td>PKA master key hash pattern.</td>
</tr>
<tr>
<td>032</td>
<td>001</td>
<td>Domain of retained key.</td>
</tr>
<tr>
<td>033</td>
<td>008</td>
<td>Serial number of processor holding retained key.</td>
</tr>
<tr>
<td>041</td>
<td>007</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

**RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature**

The format of the RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature.

Table 165 shows the format of the RSA private key token, 1024-bit Modulus-Exponent internal format for cryptographic coprocessor feature.

Table 165. RSA private internal key token, 1024-bit Modulus-Exponent format for cryptographic coprocessor feature

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X’02’, section identifier, RSA private key.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X’00’, version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the RSA private key section X’016C’ (364 decimal).</td>
</tr>
<tr>
<td>004</td>
<td>020</td>
<td>SHA-1 hash value of the private key subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.</td>
</tr>
<tr>
<td>024</td>
<td>004</td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>028</td>
<td>001</td>
<td>Key format and security: X’02’ RSA private key.</td>
</tr>
</tbody>
</table>
### Key token formats

**Table 165. RSA private internal key token, 1024-bit Modulus-Exponent format for cryptographic coprocessor feature (continued)**

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>029</td>
<td>001</td>
<td>Format of external key from which this token was derived:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'21'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'22'</td>
</tr>
<tr>
<td>030</td>
<td>020</td>
<td>SHA-1 hash of the key token structure contents that follow the public key section. If no sections follow, this field is set to binary zeros.</td>
</tr>
<tr>
<td>050</td>
<td>001</td>
<td>Key use flag bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'11xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'10xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'01xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'00xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other bits reserved, set to binary zero.</td>
</tr>
<tr>
<td>051</td>
<td>009</td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>060</td>
<td>048</td>
<td>Object Protection Key (OPK) encrypted under a PKA master key—can be under the Signature Master Key (SMK) or Key Management Master Key (KMMK) depending on key use.</td>
</tr>
<tr>
<td>108</td>
<td>128</td>
<td>Secret key exponent d, encrypted under the OPK. d = e⁻¹ mod((p-1)(q-1))</td>
</tr>
<tr>
<td>236</td>
<td>128</td>
<td>Modulus, n. n = pq where p and q are prime and 1 &lt; n &lt; 2¹⁰²⁴.</td>
</tr>
</tbody>
</table>

**RSA private key token, 1024-bit Modulus-Exponent internal format starting with CEX3C**

The format for the RSA private key token, 1024-bit Modulus-Exponent internal format starting with CEX3C.

**Table 166 shows the format for the RSA private key token, 1024-bit Modulus-Exponent internal format starting with CEX3C.**

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X'06', section identifier, RSA private key Modulus-Exponent format (RSA-PRIV).</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X'00', version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the RSA private key section X'0198' (408 decimal) + rrr + iii + xxx.</td>
</tr>
<tr>
<td>004</td>
<td>020</td>
<td>SHA-1 hash value of the private key subsection cleartext, offset 28 to and including the modulus at offset 236.</td>
</tr>
<tr>
<td>024</td>
<td>004</td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>028</td>
<td>001</td>
<td>Key format and security:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02' RSA private key.</td>
</tr>
</tbody>
</table>
## RSA private key token, 4096-bit Modulus-Exponent internal format

The format of an RSA private key, 4096-bit Modulus-Exponent token.

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
</table>
| 029              | 001             | Format of external key from which this token was derived:  
|                  |                 | X'21' External private key was specified in the clear.  
|                  |                 | X'22' External private key was encrypted.  
|                  |                 | X'23' Private key was generated using regeneration data.  
|                  |                 | X'24' Private key was randomly generated.  
| 030              | 020             | SHA-1 hash of the optional key-name section and any following optional sections. If there are no optional sections, this field is set to binary zeros.  
| 050              | 004             | Key use flag bits.  
|                  |                 | B'11xx xxxx' Only key unwrapping (KM-ONLY)  
|                  |                 | B'10xx xxxx' Both signature generation and key unwrapping (KEY-MGMT)  
|                  |                 | B'01xx xxxx' Undefined  
|                  |                 | B'00xx xxxx' Only signature generation (SIG-ONLY)  
|                  |                 | All other bits reserved, set to binary zero.  
| 054              | 006             | Reserved; set to binary zero.  
| 060              | 048             | Object Protection Key (OPK) encrypted under the Asymmetric Keys Master Key using the ede3 algorithm.  
| 108              | 128             | Private key exponent $d$, encrypted under the OPK using the ede5 algorithm. $d = e^i \mod((p-1)(q-1))$, and $1 < d < n$ where $e$ is the public exponent.  
| 236              | 128             | Modulus, $n$. $n = pq$ where $p$ and $q$ are prime and $2^{312} < n < 2^{1024}$.  
| 364              | 016             | Asymmetric-Keys Master Key hash pattern.  
| 380              | 020             | SHA-1 hash value of the blinding information subsection cleartext, offset 400 to the end of the section.  
| 400              | 002             | Length of the random number $r$, in bytes: $rrr$  
| 402              | 002             | Length of the random number $r^i$, in bytes: $iii$  
| 404              | 002             | Length of the padding field, in bytes: $xxx$  
| 406              | 002             | Reserved; set to binary zeros.  
| 408              |                 | Start of the encrypted blinding subsection  
| $rrr$            |                 | Random number $r$ (used in blinding).  
| $rrr + iii$      |                 | Random number $r^i$ (used in blinding).  
| $rrr + iii + xxx$|                 | X'00' padding of length $xxx$ bytes such that the length from the start of the encrypted blinding subsection to the end of the padding field is a multiple of eight bytes.  

End of the encrypted blinding subsection; all of the fields starting with the random number $r$ and ending with the variable length pad field are encrypted under the OPK using TDES (CBC outer chaining) algorithm.
Table 167 shows the format of an RSA private key, 4096-bit Modulus-Exponent token.

**Table 167. Private key, 4096-bit Modulus-Exponent format section (X'09')**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier: X'09' RSA private key, 4096-bit maximum Modulus-Exponent format (RSAMEVAR). This format is used for a clear or an encrypted RSA private-key in an external key-token up to a modulus size of 4096 bits. This section type is not created in releases before Release 4.1.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length (132 + ddd + nnn + xxx).</td>
</tr>
<tr>
<td>004</td>
<td>020</td>
<td>SHA-1 hash value of the private-key subsection cleartext, offset 28 to the end of the modulus.</td>
</tr>
<tr>
<td>024</td>
<td>002</td>
<td>Length in bytes of the optionally encrypted secure subsection, or X'0000' if the subsection is not encrypted.</td>
</tr>
<tr>
<td>026</td>
<td>002</td>
<td>Reserved (X'0000').</td>
</tr>
<tr>
<td>028</td>
<td>001</td>
<td>Key format and security flags: External token: X'00' Unencrypted RSA private-key subsection identifier X'82' Encrypted RSA private-key subsection identifier</td>
</tr>
<tr>
<td>029</td>
<td>001</td>
<td>Private key source flag: X'00' Generation method unknown</td>
</tr>
<tr>
<td>030</td>
<td>020</td>
<td>SHA-1 hash of all optional sections that follow the public-key section, if any, else 20 bytes of X'00'.</td>
</tr>
<tr>
<td>050</td>
<td>001</td>
<td>Key-usage flag: B'11xx xxxx' Only key unwrapping (KM-ONLY) B'10xx xxxx' Both signature generation and key unwrapping (KEY-MGMT) B'01xx xxxx' Undefined B'00xx xxxx' Only signature generation (SIG-ONLY) All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>051</td>
<td>065</td>
<td>Reserved, binary zeros.</td>
</tr>
<tr>
<td>116</td>
<td>002</td>
<td>Private-key exponent field length, in bytes: ddd.</td>
</tr>
<tr>
<td>118</td>
<td>002</td>
<td>Private-key modulus field length, in bytes: nnn.</td>
</tr>
<tr>
<td>120</td>
<td>002</td>
<td>Length of padding field, in bytes: xxx. Padding of X'00' bytes for a length of xxx bytes such that the length from the start of the confounder at offset 124 to the end of the padding field is a multiple of 8 bytes.</td>
</tr>
<tr>
<td>122</td>
<td>002</td>
<td>Reserved, binary zeros.</td>
</tr>
</tbody>
</table>

Start of the (optionally) encrypted subsection; all of the fields starting with the confounder field and ending with the variable-length pad field are enciphered for key confidentiality when the key format and security flags (offset 28) indicate that the private key is enciphered.
### Key token formats

**Table 167. Private key, 4096-bit Modulus-Exponent format section (X'09') (continued)**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>008</td>
<td>Confounder.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is an eight-byte random number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data encrypted with two-part key-encrypting key.</td>
</tr>
<tr>
<td>132</td>
<td>ddd</td>
<td>Private-key exponent, $d$:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$d = e^{-1} \mod ((p - 1)(q - 1))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where $1 &lt; d &lt; n$, and $e$ is the public exponent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The transport key encrypts the private key exponent using the EDE2 algorithm.</td>
</tr>
<tr>
<td>132 + ddd</td>
<td>xxx</td>
<td>Pad of X'00' bytes.</td>
</tr>
<tr>
<td>132 + ddd + xxx</td>
<td>nnn</td>
<td>Private-key modulus.</td>
</tr>
</tbody>
</table>

**RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section internal form**

This RSA private key token is supported on the Crypto Express3 Coprocessor and Crypto Express4 Coprocessor.

**Table 168. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') internal form**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'30' RSA private key, ME format with AES encrypted OPK.</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length: 122 + nnn + ppp</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Length of “Associated Data” section</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>Length of payload data: ppp</td>
</tr>
<tr>
<td>008</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Start of Associated Data</td>
</tr>
<tr>
<td>010</td>
<td>001</td>
<td>Associated Data Version:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02' Version 2</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>Key format and security flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02' Encrypted ME RSA private-key subsection identifier</td>
</tr>
<tr>
<td>012</td>
<td>001</td>
<td>Key source flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Internal tokens:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'21' Imported from cleartext</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'22' Imported from ciphertext</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'23' Generated using regeneration data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'24' Randomly generated</td>
</tr>
<tr>
<td>013</td>
<td>001</td>
<td>Reserved, binary zeroes.</td>
</tr>
</tbody>
</table>
### Key token formats

Table 168. RSA private key, 4096-bit Modulus-Exponent format with AES encrypted OPK section (X'30') internal form (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>014</td>
<td>001</td>
<td>Hash type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00' Clear key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02' SHA-256</td>
</tr>
<tr>
<td>015</td>
<td>032</td>
<td>SHA-256 hash of all optional sections that follow the public key section, if any; else 32 bytes of X'00'.</td>
</tr>
<tr>
<td>047</td>
<td>003</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>050</td>
<td>001</td>
<td>Key-usage flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'11xx xxxx' Only key unwrapping (KM-ONLY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'10xx xxxx' Both signature generation and key unwrapping (KEY-MGMT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'01xx xxxx' Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'00xx xxxx' Only signature generation (SIG-ONLY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translation control:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx xx1x' Private key translation is allowed (XLATE-OK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx xx0x' Private key translation is not allowed (NO-XLATE)</td>
</tr>
<tr>
<td>051</td>
<td>001</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>052</td>
<td>002</td>
<td>Length of modulus: nnn bytes</td>
</tr>
<tr>
<td>054</td>
<td>002</td>
<td>Length of private exponent: ddd bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>End of Associated Data</strong></td>
</tr>
<tr>
<td>056</td>
<td>048</td>
<td>16 byte confounder + 32-byte Object Protection Key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPK used as an AES key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encrypted with the ECC master key.</td>
</tr>
<tr>
<td>104</td>
<td>016</td>
<td>Key verification pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For an encrypted private key, ECC master-key verification pattern (MKVP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a skeleton, binary zeros</td>
</tr>
<tr>
<td>120</td>
<td>002</td>
<td>Reserved, binary zeros.</td>
</tr>
<tr>
<td>122</td>
<td>nnn</td>
<td>Modulus</td>
</tr>
<tr>
<td>122+nnn</td>
<td>ppp</td>
<td>Payload starts here and includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When this section is unencrypted:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Clear private exponent d.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Length ppp bytes : ddd + 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When this section is encrypted:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Private exponent d within the AESKW-wrapped payload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Length ppp bytes : ddd + AESKW format overhead</td>
</tr>
</tbody>
</table>
Key token formats

**RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section internal form**

This RSA private key token is supported on the Crypto Express3 Coprocessor and Crypto Express4 Coprocessor.

**Table 169. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') external form**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'31' RSA private key, CRT format with AES encrypted OPK</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length: 134 + nnn + xxx</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Length of “Associated Data” section</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>Length of payload data: xxx</td>
</tr>
<tr>
<td>008</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
</tbody>
</table>

Start of Associated Data

| 010            | 001            | Associated Data Version:                                                    |
|                |                | X'03' Version 3                                                             |
| 011            | 001            | Key format and security flag:                                               |
|                |                | X'08' Unencrypted RSA private-key subsection identifier                      |
| 012            | 001            | Key source flag:                                                            |
|                |                | X'21' Imported from cleartext                                               |
|                |                | X'22' Imported from ciphertext                                              |
|                |                | X'23' Generated using regeneration data                                      |
|                |                | X'24' Randomly generated                                                    |
| 013            | 001            | Reserved, binary zeroes.                                                    |
| 014            | 001            | Hash type:                                                                  |
|                |                | X'00' Clear key                                                             |
|                |                | X'01' SHA-256                                                               |
| 015            | 032            | SHA-256 hash of all optional sections that follow the public key section, if |
|                |                | any; else 32 bytes of X'00'.                                                |
| 047            | 003            | Reserved, binary zero.                                                      |
Table 169. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31')

internal form (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>050</td>
<td>001</td>
<td>Key-usage flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'11xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only key unwrapping (KM-ONLY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'10xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Both signature generation and key unwrapping (KEY-MGMT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'01xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'00xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only signature generation (SIG-ONLY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translation control:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx xx1x'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Private key translation is allowed (XLATE-OK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx xx0x'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Private key translation is not allowed (NO-XLATE)</td>
</tr>
<tr>
<td>051</td>
<td>001</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>052</td>
<td>002</td>
<td>Length of the prime number, p, in bytes: ppp.</td>
</tr>
<tr>
<td>054</td>
<td>002</td>
<td>Length of the prime number, q, in bytes: qqq</td>
</tr>
<tr>
<td>056</td>
<td>002</td>
<td>Length of dp : rrr.</td>
</tr>
<tr>
<td>058</td>
<td>002</td>
<td>Length of dq : sss.</td>
</tr>
<tr>
<td>060</td>
<td>002</td>
<td>Length of U: uuu.</td>
</tr>
<tr>
<td>062</td>
<td>002</td>
<td>Length of modulus, nnn.</td>
</tr>
<tr>
<td>064</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>066</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End of Associated Data</td>
</tr>
<tr>
<td>068</td>
<td>048</td>
<td>16 byte confounder + 32-byte Object Protection Key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPK used as an AES key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encrypted with the ECC master key.</td>
</tr>
<tr>
<td>116</td>
<td>016</td>
<td>Key verification pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For an encrypted private key, ECC master-key verification pattern (MKVP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a skeleton, binary zeros</td>
</tr>
<tr>
<td>132</td>
<td>002</td>
<td>Reserved, binary zeros</td>
</tr>
<tr>
<td>134</td>
<td>nnn</td>
<td>Modulus, n, n=pq, where p and q are prime.</td>
</tr>
</tbody>
</table>
Key token formats

Table 169. RSA private key, 4096-bit Chinese Remainder Theorem format with AES encrypted OPK section (X'31') internal form (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
</table>
| 134+nnn xxx    | XXX            | Payload starts here and includes: When this section is unencrypted: 
• Clear prime number p 
• Clear prime number q 
• Clear dp 
• Clear dq 
• Clear U 
• Length xxx bytes: ppp + qqq + rrr + sss + uuu + 0 When this section is encrypted: 
• prime number p 
• prime number q 
• dp 
• dq 
• U 
• within the AESKW-wrapped payload. 
Length xxx bytes : ppp + qqq + rrr + sss + uuu + AESKW format overhead |

RSA Private Key Token, 4096-bit Chinese Remainder Theorem Internal Form

This RSA private key token with up to 2048-bit modulus is supported on all coprocessors. The modulus size is increased to 4096-bit on the z9 EC, z9 BC, z10 EC, z10 BC, or later machines with the Nov. 2007 or later version of the licensed internal code installed on the CCA Crypto Express coprocessor.

Table 170. RSA Private Internal Key Token, 4096-bit Chinese Remainder Theorem Internal Format

<table>
<thead>
<tr>
<th>Offset (Dec)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>X'08', section identifier, RSA private key, CRT format (RSA-CRT)</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>X'00', version.</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the RSA private-key section, 132 + ppp + qqq + rrr + sss + uuu + ttt + iii + xxx + nnn.</td>
</tr>
<tr>
<td>004</td>
<td>020</td>
<td>SHA-1 hash value of the private-key subsection cleartext, offset 28 to the end of the modulus.</td>
</tr>
<tr>
<td>024</td>
<td>004</td>
<td>Reserved; set to binary zero.</td>
</tr>
</tbody>
</table>
| 028          | 001             | Key format and security: 
X'08' Encrypted RSA private-key subsection identifier, Chinese Remainder form. |
| 029          | 001             | Key derivation method: 
X'21' External private key was specified in the clear. 
X'22' External private key was encrypted. 
X'23' Private key was generated using regeneration data. 
X'24' Private key was randomly generated. |
### Table 170. RSA Private Internal Key Token, 4096-bit Chinese Remainder Theorem Internal Format (continued)

<table>
<thead>
<tr>
<th>Offset (Dec)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>030</td>
<td>020</td>
<td>SHA-1 hash of the optional key-name section and any following sections. If there are no optional sections, then 20 bytes of X'00'.</td>
</tr>
<tr>
<td>050</td>
<td>004</td>
<td>Key use flag bits:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>054</td>
<td>002</td>
<td>Length of prime number, p, in bytes: ppp.</td>
</tr>
<tr>
<td>056</td>
<td>002</td>
<td>Length of prime number, q, in bytes: qqq.</td>
</tr>
<tr>
<td>058</td>
<td>002</td>
<td>Length of (d_p), in bytes: rrr.</td>
</tr>
<tr>
<td>060</td>
<td>002</td>
<td>Length of (d_q), in bytes: sss.</td>
</tr>
<tr>
<td>062</td>
<td>002</td>
<td>Length of U, in bytes: uuu.</td>
</tr>
<tr>
<td>064</td>
<td>002</td>
<td>Length of modulus, n, in bytes: nnn.</td>
</tr>
<tr>
<td>066</td>
<td>002</td>
<td>Length of the random number r, in bytes: ttt.</td>
</tr>
<tr>
<td>068</td>
<td>002</td>
<td>Length of the random number (r^{*-1}), in bytes: iii.</td>
</tr>
<tr>
<td>070</td>
<td>002</td>
<td>Length of padding field, in bytes: xxx.</td>
</tr>
<tr>
<td>072</td>
<td>004</td>
<td>Reserved, set to binary zero.</td>
</tr>
<tr>
<td>076</td>
<td>016</td>
<td>RSA Master Key hash pattern.</td>
</tr>
<tr>
<td>092</td>
<td>032</td>
<td>Object Protection Key (OPK) encrypted under the RSA Master Key using the TDES (CBC outer chaining) algorithm.</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td>Start of the encrypted secure subsection, encrypted under the OPK using TDES (CBC outer chaining).</td>
</tr>
<tr>
<td>132 + ppp</td>
<td></td>
<td>Random number, confounder.</td>
</tr>
<tr>
<td>132 + ppp + qqq</td>
<td></td>
<td>Prime number, p.</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr</td>
<td>x</td>
<td>Prime number, q</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss</td>
<td>x</td>
<td>(d_p = d \mod (p - 1))</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu</td>
<td>x</td>
<td>(d_q = d \mod (q - 1))</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu + ttt</td>
<td>x</td>
<td>(U = -1 \mod (p). q^{*-1} \mod (p)).</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu + ttt + iii</td>
<td>x</td>
<td>Random number r (used in blinding).</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu + ttt + xxx</td>
<td>x</td>
<td>Random number (-1 \cdot r^{*-1}) (used in blinding).</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu + ttt + iii + xxx</td>
<td>x</td>
<td>X'00' padding of length xxx bytes such that the length from the start of the confounder at offset 124 to the end of the padding field is a multiple of eight bytes.</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu + ttt + iii + xxx</td>
<td>x</td>
<td>End of the encrypted secure subsection; all of the fields starting with the confounder field and ending with the variable length pad field are encrypted under the OPK using TDES (CBC outer chaining) for key confidentiality.</td>
</tr>
<tr>
<td>132 + ppp + qqq + rrr + sss + uuu + ttt + iii + xxx + nnn</td>
<td>x</td>
<td>Modulus, n, n = pq where p and q are prime and (1 &lt; n &lt; 2^{2048}).</td>
</tr>
</tbody>
</table>
RSA variable Modulus-Exponent token

A description of the fields in the new variable length Modulus-Exponent token.

Table 171 describes the fields in the new variable length Modulus-Exponent token. Currently, only the external form of the token will be used. There are no blinding values for the token. The latest level hardware makes this unnecessary.

Table 171. RSA variable Modulus-Exponent token format

<table>
<thead>
<tr>
<th>Number</th>
<th>If External Key</th>
<th>New version '09' field</th>
<th>If Internal Key</th>
<th>Length in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'09'</td>
<td>sectionId</td>
<td>'09'</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>'00'</td>
<td>version</td>
<td>'00'</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>132 + dLength + nLength + padLength</td>
<td>sectionLength</td>
<td>132 + dLength + nLength + padLength</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Hash over fields 7 - end of section (clear values)</td>
<td>sha1Hash</td>
<td>Hash over fields 7 - end of section</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>8 + dLength + padLength</td>
<td>encrypted sectionLength</td>
<td>8 + dLength + padLength</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>This is actually a reserved field, not a pad '0000'</td>
<td>pad</td>
<td>'0000'</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>'82' encrypted external key or '00' clear external key</td>
<td>keyFormat</td>
<td>'02' encrypted operational key</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>'00'</td>
<td>pedigree</td>
<td>'21', '22', '23', or '24' as '06' token</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Hash over sections which follow the public key section, or '00'</td>
<td>sha1Key NameHash</td>
<td>Hash over sections which follow the public key section, or '00'</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>'02' indicates that the key is translatable</td>
<td>keyUsageFlag</td>
<td>same as in '06'</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>'00'</td>
<td>reserved1</td>
<td>'00'</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Binary zeroes</td>
<td>OPK</td>
<td>8 byte confounder + 40-byte (5-part) DES key, encrypted with the PKA master key</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>Binary zeroes</td>
<td>mkHash Pattern</td>
<td>16 byte MKVP</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>Length of private exponent</td>
<td>dLength</td>
<td>Length of private exponent</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>Length of modulus</td>
<td>nLength</td>
<td>Length of modulus</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>Length required to pad dLength to a multiple of 8</td>
<td>padLength</td>
<td>Length required to pad dLength to a multiple of 8</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>'0000'</td>
<td>reserved2</td>
<td>'0000'</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>Random value - encrypted data (with PKA MK) begins here</td>
<td>confounder</td>
<td>encrypted data (with 5-part OPK) begins here</td>
<td>8</td>
</tr>
<tr>
<td>19</td>
<td>&lt;d follows, then pad, then n&gt;</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

ECC key token

The format of ECC public and private key tokens.
Table 175 on page 639 and Table 174 show the format of ECC public and private key tokens.

CCA allows a choice between two types of elliptic curves when generating an ECC key. One is Brainpool, and the other is Prime. Table 172 and Table 173 show the size and name of each supported elliptic curve, along with its object identifier (OID) in dot notation.

**Table 172. Supported Prime elliptic curves by size, name, and object identifier**

<table>
<thead>
<tr>
<th>Size of prime $p$ in bits (key length)</th>
<th>OID in dot notation</th>
<th>ANSI X9.62 ECDSA prime curve ID</th>
<th>NIST-recommended elliptic curve ID</th>
<th>SEC 2 recommended elliptic curve domain parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>192</td>
<td>1.2.840.10045.3.1.1</td>
<td>prime192v1</td>
<td>P-192</td>
<td>secp192r1</td>
</tr>
<tr>
<td>224</td>
<td>1.3.132.0.33</td>
<td>N/A</td>
<td>P-224</td>
<td>secp224r1</td>
</tr>
<tr>
<td>256</td>
<td>1.2.840.10045.3.1.7</td>
<td>prime256v1</td>
<td>P-256</td>
<td>secp256r1</td>
</tr>
<tr>
<td>384</td>
<td>1.3.132.0.34</td>
<td>N/A</td>
<td>P-384</td>
<td>secp384r1</td>
</tr>
<tr>
<td>521</td>
<td>1.3.132.0.35</td>
<td>N/A</td>
<td>P-521</td>
<td>secp521r1</td>
</tr>
</tbody>
</table>

**Table 173. Supported Brainpool elliptic curves by size, name, and object identifier**

<table>
<thead>
<tr>
<th>Size of prime $p$ in bits (key length)</th>
<th>OID in dot notation</th>
<th>Brainpool elliptic curve ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>1.3.36.3.3.2.8.1.1.1</td>
<td>brainpoolP160r1</td>
</tr>
<tr>
<td>192</td>
<td>1.3.36.3.3.2.8.1.1.3</td>
<td>brainpoolP192r1</td>
</tr>
<tr>
<td>224</td>
<td>1.3.36.3.3.2.8.1.1.5</td>
<td>brainpoolP224r1</td>
</tr>
<tr>
<td>256</td>
<td>1.3.36.3.3.2.8.1.1.7</td>
<td>brainpoolP256r1</td>
</tr>
<tr>
<td>320</td>
<td>1.3.36.3.3.2.8.1.1.9</td>
<td>brainpoolP320r1</td>
</tr>
<tr>
<td>384</td>
<td>1.3.36.3.3.2.8.1.1.11</td>
<td>brainpoolP384r1</td>
</tr>
<tr>
<td>512</td>
<td>1.3.36.3.3.2.8.1.1.13</td>
<td>brainpoolP512r1</td>
</tr>
</tbody>
</table>

**Table 174. ECC private-key section**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier: X'20' ECC private key</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of section in bytes.</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Wrapping method: X'00' Section is unencrypted (clear) X'01' AESKW X'02' CBC</td>
</tr>
<tr>
<td>005</td>
<td>001</td>
<td>Hash method used for wrapping: X'00' None (clear key) X'01' SHA-224 X'02' SHA-256</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
</tbody>
</table>
### ECC private-key section (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>008</td>
<td>001</td>
<td>Key-usage flag.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Management of symmetric keys and generation of digital signatures:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'11xx xxxx' Only key establishment (KM-ONLY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'10xx xxxx' Both signature generation and key establishment (KEY-MGMT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'01xx xxxx' Undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'00xx xxxx' Only signature generation (SIG-ONLY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translation control:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx xx1x' Private key translation is allowed (XLATE-OK)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx xx0x' Private key translation is not allowed (NO-XLATE)</td>
</tr>
<tr>
<td>009</td>
<td>001</td>
<td>Curve type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00' Prime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01' Brainpool</td>
</tr>
<tr>
<td>010</td>
<td>001</td>
<td>Key format and security flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'08' Encrypted internal ECC private key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'40' Unencrypted external ECC private key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'42' Encrypted external ECC private key</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>Reserved, binary zero</td>
</tr>
<tr>
<td>012</td>
<td>002</td>
<td>Length of prime $p$ in bits. See Table 172 on page 637 and Table 173 on page 637</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00A0' 160 (Brainpool)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00C0' 192 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00E0' 224 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0100' 256 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0140' 320 (Brainpool)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0180' 384 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0200' 512 (Brainpool)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0209' 521 (Prime)</td>
</tr>
<tr>
<td>014</td>
<td>002</td>
<td>Length in bytes of IBM associated data.</td>
</tr>
<tr>
<td>016</td>
<td>008</td>
<td>Key verification pattern.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>External key-token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For an encrypted private key, KEK verification pattern (KVP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a clear private key, binary zeros</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a skeleton, binary zeros</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Internal key-token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For encrypted private key, master-key verification pattern (MKVP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For a skeleton, binary zeros</td>
</tr>
<tr>
<td>024</td>
<td>048</td>
<td>Object Protection Key (OPK).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For an internal key token: OPK, integrity check value (ICV), 8-byte confounder,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and a 256-bit AES key used with the AESKW algorithm to encrypt the ECC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>private key, otherwise binary zeros.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: The OPK is encrypted by the APKA master key using AESKW.</td>
</tr>
<tr>
<td>072</td>
<td>002</td>
<td>Length in bytes of associated data.</td>
</tr>
<tr>
<td>074</td>
<td>002</td>
<td>Length in bytes of formatted section, $bb$.</td>
</tr>
</tbody>
</table>

**Associated data section**
### Key token formats

#### Table 174. ECC private-key section (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of IBM associated data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>076</td>
<td>001</td>
<td>Associated data section version (X'00'). Includes IBM associated data and user-definable associated data.</td>
</tr>
<tr>
<td>077</td>
<td>001</td>
<td>Length in bytes of the key label: $kl$ (0 - 64).</td>
</tr>
<tr>
<td>078</td>
<td>002</td>
<td>Length in bytes of the IBM associated data, including key label and IBM extended associated data ($\geq 16$).</td>
</tr>
<tr>
<td>080</td>
<td>002</td>
<td>Length in bytes of the IBM extended associated data (0).</td>
</tr>
<tr>
<td>082</td>
<td>001</td>
<td>Length in bytes of the user-definable associated data: $uad$ (0 - 100).</td>
</tr>
<tr>
<td>083</td>
<td>001</td>
<td>Curve type (see offset 009).</td>
</tr>
<tr>
<td>084</td>
<td>002</td>
<td>Length of $p$ in bits (see offset 012).</td>
</tr>
<tr>
<td>086</td>
<td>001</td>
<td>Key-usage flag (see offset 008).</td>
</tr>
<tr>
<td>087</td>
<td>001</td>
<td>Key format and security flag (see 010).</td>
</tr>
<tr>
<td>088</td>
<td>004</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>092</td>
<td>$kl$</td>
<td>Optional key label.</td>
</tr>
<tr>
<td>092 + $kl$ + $iead$ + $uad$</td>
<td>$bb$</td>
<td>Formatted section (payload), which includes private key $d$:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Clear-key section contains $d$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Encrypted-key section contains $d$ within the AESKW-wrapped payload.</td>
</tr>
</tbody>
</table>

End of IBM associated data

End of associated data section

#### Table 175. ECC public-key section

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'21' ECC public key</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length.</td>
</tr>
<tr>
<td>004</td>
<td>004</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>008</td>
<td>001</td>
<td>Curve type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00' Prime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01' Brainpool</td>
</tr>
<tr>
<td>009</td>
<td>001</td>
<td>Reserved, binary zero</td>
</tr>
<tr>
<td>010</td>
<td>002</td>
<td>Length of prime $p$ in bits. See Table 172 on page 637 and Table 173 on page 637:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00A0' 160 (Brainpool)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00C0' 192 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00E0' 224 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0100' 256 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0140' 320 (Brainpool)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0180' 384 (Brainpool, Prime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0200' 512 (Brainpool)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0209' 521 (Prime)</td>
</tr>
<tr>
<td>012</td>
<td>002</td>
<td>Length of public key $q$ in bytes. Value includes key material length plus a one-byte flag to indicate if the key material is compressed.</td>
</tr>
</tbody>
</table>
### Key token formats

**Table 175. ECC public-key section (continued)**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>014</td>
<td>cc</td>
<td>Public key (q).</td>
</tr>
</tbody>
</table>

**PKA null key token**

The format for a PKA null key token.

*Table 176 shows the format for a PKA null key token.*

**Table 176. PKA null key token format**

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X'00' Token identifier (indicates that this is a null key token).</td>
</tr>
<tr>
<td>1</td>
<td>Version, X'00'.</td>
</tr>
<tr>
<td>2 - 3</td>
<td>X'0008' Length of the key token structure.</td>
</tr>
<tr>
<td>4 - 7</td>
<td>Ignored (should be zero).</td>
</tr>
</tbody>
</table>

**HMAC key token**

The two formats of the HMAC key token.

HMAC key tokens have two formats, "HMAC variable-length symmetric key token" and "HMAC symmetric null key token" on page 646.

**HMAC variable-length symmetric key token**

The format of the HMAC variable-length symmetric key-token.

*Table 177 shows the format of the HMAC variable-length symmetric key-token. An HMAC token is used by the HMAC Generate(CSNBHMG) and HMAC Verify(CSNBHMV) verbs to generate and verify keyed hash Message Authentication Codes.*

**Table 177. HMAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later)**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td></td>
<td>Token identifier:</td>
</tr>
<tr>
<td>000</td>
<td>01</td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00' Internal key-token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01' External key-token</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>002</td>
<td>02</td>
<td>Length in bytes of the overall token structure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(54 + kl + icad + uad + TLV lengths + ((pl + 7) / 8))</td>
</tr>
<tr>
<td>004</td>
<td>01</td>
<td>Token version number (X'05').</td>
</tr>
<tr>
<td>005</td>
<td>03</td>
<td>Reserved, binary zero.</td>
</tr>
</tbody>
</table>

**Wrapping information section (all data related to wrapping the token)**
### Key token formats

**Table 177. HMAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>008</td>
<td>01</td>
<td>Key material state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'03'</td>
</tr>
<tr>
<td>009</td>
<td>01</td>
<td>Key verification pattern (KVP) type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02'</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> Key-wrapping method X'03' (PKOAEP2) has no KVP.</td>
</tr>
<tr>
<td>010</td>
<td>16</td>
<td>KVP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value is left-aligned in the field and padded on the right with binary zeros.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> For key-wrapping method X'03' (PKOAEP2), this value is filled with binary zeros.</td>
</tr>
<tr>
<td>026</td>
<td>01</td>
<td>Encrypted section key-wrapping method:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'03'</td>
</tr>
<tr>
<td>027</td>
<td>01</td>
<td>Hash algorithm used for wrapping.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For clear key wrapping method (X'00' at offset 26):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For AESKW wrapping method (X'02' at offset 26):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For PKOAEP2 wrapping method (X'03' at offset 26):</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'04'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'08'</td>
</tr>
<tr>
<td>028</td>
<td>02</td>
<td>Reserved, binary zero.</td>
</tr>
</tbody>
</table>

End of wrapping information section

AESKW components: (1) associated data and (2) optional clear key or encrypted AESKW payload.

Associated data section

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>030</td>
<td>01</td>
<td>Associated data section version (X'01').</td>
</tr>
<tr>
<td>031</td>
<td>01</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>032</td>
<td>02</td>
<td>Length in bytes of the associated data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(24 + kl + iod + uad + TLV) lengths</td>
</tr>
<tr>
<td>034</td>
<td>01</td>
<td>Length in bytes of the key label: (kl) (0 or 64).</td>
</tr>
</tbody>
</table>
Table 177. HMAC variable-length symmetric key-token, version X’05’ (CCA 4.1.0 or later) (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>035</td>
<td>01</td>
<td>Length in bytes of the IBM extended associated data: \textit{iead} (0).</td>
</tr>
<tr>
<td>036</td>
<td>01</td>
<td>Length in bytes of the user-definable associated data: \textit{uad} (0 - 255).</td>
</tr>
<tr>
<td>037</td>
<td>01</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>038</td>
<td>02</td>
<td>Length \textit{in bits} of the payload: \textit{pl}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid values are 0 when no key is present, 80 - 2048 when key is clear, and 464 - 2432 when key is encrypted.</td>
</tr>
<tr>
<td>040</td>
<td>01</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>041</td>
<td>01</td>
<td>Type of algorithm for which the key can be used. X’03’ HMAC</td>
</tr>
<tr>
<td>042</td>
<td>02</td>
<td>Key type: X’0002’ MAC</td>
</tr>
<tr>
<td>044</td>
<td>01</td>
<td>Key-usage fields count: \textit{kuf} (2).</td>
</tr>
<tr>
<td>045</td>
<td>02</td>
<td>Key-usage field 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>High-order byte:</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>B’1xxx xxxx’</td>
<td>Key can be used for generate.</td>
</tr>
<tr>
<td></td>
<td>B’0xxx xxxx’</td>
<td>Key cannot be used for generate.</td>
</tr>
<tr>
<td></td>
<td>B’x1xx xxxx’</td>
<td>Key can be used for verify.</td>
</tr>
<tr>
<td></td>
<td>B’x0xx xxxx’</td>
<td>Key cannot be used for verify.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All unused bits are reserved and must be zero.</td>
</tr>
<tr>
<td></td>
<td>Low-order byte:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>B’xxx 1xxx’</td>
<td>The key can be used only in user-defined extensions (UDXs).</td>
</tr>
<tr>
<td></td>
<td>B’xxx 0xxx’</td>
<td>The key can be used in UDXs and CCA.</td>
</tr>
<tr>
<td></td>
<td>B’xxx xuuu’</td>
<td>Reserved for UDXs, where \textit{uuu} are UDX-defined bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All unused bits are reserved and must be zero.</td>
</tr>
</tbody>
</table>
### Key token formats

Table 177. HMAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>047</td>
<td>02</td>
<td>Key-usage field 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>High-order byte:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'1xxx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'0xxx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'x1xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'x0xx xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xx1x xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xx0x xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxx1 xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxx0 xxxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx 1xxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B'xxxx 0xxx'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All unused bits are reserved and must be zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low-order byte: All bits are reserved and must be zero.</td>
</tr>
<tr>
<td>049</td>
<td>01</td>
<td>Key-management fields count: kmf (2).</td>
</tr>
</tbody>
</table>
### Table 177. HMAC variable-length symmetric key-token, version X'05' (CCA 4.1.0 or later) (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>050</td>
<td>02</td>
<td>Key-management field 1.</td>
</tr>
</tbody>
</table>

**High-order byte:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B'1xxx xxxx'</td>
<td>Allow export using symmetric transport key.</td>
</tr>
<tr>
<td>B'0xxx xxxx'</td>
<td>Prohibit export using symmetric transport key.</td>
</tr>
<tr>
<td>B'x1xx xxxx'</td>
<td>Allow export using unauthenticated asymmetric transport key.</td>
</tr>
<tr>
<td>B'x0xx xxxx'</td>
<td>Prohibit export using unauthenticated asymmetric transport key.</td>
</tr>
<tr>
<td>B'xx1x xxxx'</td>
<td>Allow export using authenticated asymmetric transport key.</td>
</tr>
<tr>
<td>B'xx0x xxxx'</td>
<td>Prohibit export using authenticated asymmetric transport key.</td>
</tr>
<tr>
<td>B'xxx1 xxxx'</td>
<td>Allow export to TR-31 format.</td>
</tr>
<tr>
<td>B'xxx0 xxxx'</td>
<td>Prohibit export to TR-31 format.</td>
</tr>
<tr>
<td>B'xxx 1xxx'</td>
<td>Allow export in raw format.</td>
</tr>
<tr>
<td>B'xxx 0xxx'</td>
<td>Prohibit export in raw format.</td>
</tr>
</tbody>
</table>

All unused bits are reserved and must be zero.

**Low-order byte:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B'1xxx xxxx'</td>
<td>Prohibit export using a DES transport key.</td>
</tr>
<tr>
<td>B'0xxx xxxx'</td>
<td>Allow export using a DES transport key.</td>
</tr>
<tr>
<td>B'x1xx xxxx'</td>
<td>Prohibit export using an AES transport key.</td>
</tr>
<tr>
<td>B'x0xx xxxx'</td>
<td>Allow export using an AES transport key.</td>
</tr>
<tr>
<td>B'xxx 1xxx'</td>
<td>Prohibit export using an RSA transport key.</td>
</tr>
<tr>
<td>B'xxx 0xxx'</td>
<td>Allow export using an RSA transport key.</td>
</tr>
<tr>
<td>B'xxx x1xx'</td>
<td>Key cannot be derived using an ECC key.</td>
</tr>
<tr>
<td>B'xxx x0xx'</td>
<td>Key can be derived using an ECC key.</td>
</tr>
</tbody>
</table>

All unused bits are reserved and must be zero.
**Table 177. HMAC variable-length symmetric key-token, version X’05’ (CCA 4.1.0 or later) (continued)**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>052</td>
<td>02</td>
<td>Key-management field 2. High-order byte:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’11xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’10xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’01xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B’00xx xxxx’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All bits are reserved and must be zero.</td>
</tr>
<tr>
<td>054</td>
<td>kl</td>
<td>Optional key label.</td>
</tr>
<tr>
<td>054 + kl</td>
<td>iead</td>
<td>IBM extended associated data.</td>
</tr>
<tr>
<td>054 + kl + iead</td>
<td>uad</td>
<td>User-definable associated data.</td>
</tr>
</tbody>
</table>

End of associated data section

Optional clear key or encrypted AESKW payload

| 054 + kl + iead + uad [(pl + 7)/8] | Clear key or encrypted wrapped/encoded payload. |

End of optional clear key or encrypted AES KW payload

| 054 + kl + iead + uad + (pl + 7)/8 | End of AESKW components |

Unencrypted AESKW payload (This data will never appear in the clear outside of the cryptographic coprocessor)

| 000 | 6     | Integrity check value. Six byte constant: X’A6A6A6A6A6A6’. |
| 006 | 1     | Length of the padding in bits: pb |
| 007 | 1     | Length of the hash the associated data in bytes: 32 |
| 008 | 4     | Hash options |
| 012 | hoh - 4 | Hash of the associated data |
| 008 + hoh [(pl / 8) - 8 - hoh] | Key data and padding (key data is left-aligned). |

Note: All numbers are in big endian format.
Key token formats

**HMAC symmetric null key token**

The format of the HMAC symmetric null key token.

Table 178 shows the format of the HMAC symmetric null key token.

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X'00' Token identifier, which indicates that this is a null key token.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X'00' Version</td>
</tr>
<tr>
<td>2 - 3</td>
<td>2</td>
<td>X'0008' Length of the key token structure.</td>
</tr>
<tr>
<td>4 - 7</td>
<td>4</td>
<td>Ignored (zero).</td>
</tr>
</tbody>
</table>

**TR-31 optional block data**

This section describes the IBM-defined data for a TR-31 optional block that can be contained in a TR-31 key block.

See X9 TR-31 2010: Interoperable Secure Key Exchange Block Specification for Symmetric Algorithms for the definition of a TR-31 key block. As defined by X9 TR-31, a TR-31 key block can contain one or more optional blocks. A TR-31 key block contains at least one optional block when byte number 12 - 13 is a value other than ASCII string "00".

The data of an IBM-defined optional block contains ASCII string "00" (X'3130') in the first two bytes, and contains ASCII string "IBMC" (X'49424D43') beginning at offset 4 of the data. CCA treats an optional block with these characteristics as a proprietary container for a CCA control vector. See Table 179 on page 647. An optional block with different characteristics is ignored by CCA.

If a TR-31 key block contains an optional block as defined by Table 179 on page 647, the data contains a copy of the 8-byte or 16-byte DES control vector that was in the CCA key-token of the key being exported. The copied control vector is in hex-ASCII format.

The control vector is only copied from the CCA key-token when the user of the Key Export to TR31 verb specifies a control vector transport control keyword (INCL-CV or ATTR-CV):

1. If the optional block contains a control vector as the result of specifying the INCL-CV keyword during export, the key usage and mode of use fields indicate the key attributes, and these attributes are verified during export to be compatible with the ones in the included control vector.

2. If the optional block contains a control vector as the result of specifying the ATTR-CV keyword during export, the key usage field (byte number 5 - 6 of the TR-31 key block) is set to the proprietary value "10" (X'3130'), and the mode of use field (byte number 8) is set to the proprietary value "1" (X'31'). These proprietary values indicate that the key attributes are specified in the included control vector.

See Chapter 4, "TR-31 symmetric-key management,” on page 71 for additional information on how CCA uses an IBM-defined optional block in a TR-31 key block.
### Key token formats

**Table 179. IBM optional block data in a TR-31 key block**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>02</td>
<td>Proprietary ID of TR-31 optional block (alphanumeric-ASCII): X'3130' IBM proprietary optional block (ASCII string &quot;10&quot;)</td>
</tr>
<tr>
<td>02</td>
<td>02</td>
<td>Length of optional block (hex-ASCII): For TLV valued to '01': X'3143' &quot;1C&quot; for 8-byte (single-length) control vector X'3243' &quot;2C&quot; for 16-byte (double-length) control vector</td>
</tr>
</tbody>
</table>

**Beginning of optional block data**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>04</td>
<td>&quot;Magic&quot; value (alphanumeric-ASCII): X'49424D43' A constant value (&quot;IBMC&quot;) used to reduce ambiguity and the chance for false interpretation of the proprietary optional blocks of non-IBM vendors. An optional block that uses the same proprietary ID but does not include this magic value will be ignored.</td>
</tr>
<tr>
<td>08</td>
<td>02</td>
<td>Tag-length-value (TLV) ID (numeric-ASCII): X'3031' IBM CCA control vector ('01')</td>
</tr>
<tr>
<td>10</td>
<td>02</td>
<td>Length of TLV (hex-ASCII) For TLV valued to '01': X'3134' &quot;14&quot; (decimal 20) TLV of 8-byte control vector X'3234' &quot;24&quot; (decimal 36) TLV of 16-byte control vector</td>
</tr>
<tr>
<td>12</td>
<td>16 or 32</td>
<td>Control vector (hex-ASCII)</td>
</tr>
</tbody>
</table>

### Trusted blocks

A **key token** is a data structure that contains information about a key and usually contains a key or keys.

A trusted block is an extension of CCA key tokens using new section identifiers. A trusted block was introduced to CCA beginning with Release 3.25. Trusted blocks are an integral part of a remote key-loading process. See "Remote key loading" on page 46.

In general, a key that is available to an application program or held in key storage is multiply-enciphered by some other key. When a key is enciphered by the CCA node's master key, the key is designated an internal key and is held in an internal key-token structure. Therefore, an internal key token or internal trusted block is used to hold a key and its related information for use at a specific CCA node.

An external key token or external trusted block is used to communicate a key between nodes, or to hold a key in a form not enciphered by a CCA master key. DES keys and PKA private-keys contained in an external key-token or external trusted block are multiply-enciphered by a transport key. In a CCA-node, a transport key is a double-length DES key encrypting key (KEK).

Trusted blocks contain various items, some of which are optional, and some of which can be present in different forms. Tokens are composed of concatenated sections that, unlike CCA PKA key tokens, occur in no prescribed order.
Key token formats

As with other CCA key-tokens, both internal and external forms are defined:

- An external trusted block contains a randomly generated confounder and a triple-length MAC key enciphered under a DES IMP-PKA transport key. The MAC key is used to calculate an ISO 16609 CBC mode TDES MAC of the trusted block contents. An external trusted block is created by the Trusted Block Create verb. This verb can:
  1. Create an inactive external trusted block
  2. Change an external trusted block from inactive to active

- An internal trusted block contains a confounder and triple-length MAC key enciphered under a variant of the PKA master key. The MAC key is used to calculate a TDES MAC of the trusted block contents. A PKA master-key verification pattern is also included to enable determination that the proper master key is available to process the key. The Remote Key Export verb only operates on trusted blocks that are internal. An internal trusted block must be imported from an external trusted block that is active using the PKA Key Import verb.

Note: Trusted blocks do not contain a private key section.

Trusted block organization

A trusted block is a concatenation of a header followed by an unordered set of sections.

Some elements are required, while others are optional. The data structures of these sections are summarized in Table 180:

<table>
<thead>
<tr>
<th>Section</th>
<th>Reference</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>[Table 181 on page 650]</td>
<td>Trusted block token header</td>
</tr>
<tr>
<td>X'11'</td>
<td>[Table 182 on page 651]</td>
<td>Trusted block public key</td>
</tr>
<tr>
<td>X'12'</td>
<td>[Table 183 on page 652]</td>
<td>Trusted block rule</td>
</tr>
<tr>
<td>X'13'</td>
<td>[Table 190 on page 659]</td>
<td>Trusted block name (key label)</td>
</tr>
<tr>
<td>X'14'</td>
<td>[Table 191 on page 659]</td>
<td>Trusted block information</td>
</tr>
<tr>
<td>X'15'</td>
<td>[Table 195 on page 662]</td>
<td>Trusted block application-defined data</td>
</tr>
</tbody>
</table>

Every trusted block starts with a token header. The first byte of the token header determines the key form:

- An external header (first byte X'1E'), created by the Trusted Block Create verb
- An internal header (first byte X'1F'), imported from an active external trusted block by the PKA Key Import verb

Following the token header of a trusted block is an unordered set of sections. A trusted block is formed by concatenating these sections to a trusted block header:

- An optional public-key section (trusted block section identifier X'11')
  The trusted block trusted RSA public key section includes the key itself in addition to a key-usage flag. No multiple sections are allowed.
- An optional rule section (trusted block section identifier X'12')
  A trusted block can have zero or more rule sections.
  1. A trusted block with no rule sections can be used by the PKA Key Token Change and PKA Key Import verbs. A trusted block with no rule sections
can also be used by the Digital Signature Verify verb, provided there is an RSA public key section that has its key-usage flag bits set to allow digital signature operations.

2. At least one rule section is required when the Remote Key Export verb is used to:
   - Generate an RKX key-token
   - Export an RKX key-token
   - Export a CCA DES key-token
   - Encrypt the clear generated or exported key using the provided vendor certificate

3. If a trusted block has multiple rule sections, each rule section must have a unique 8-character Rule ID.

   - An optional name (key label) section (trusted block section identifier X'13')
     The trusted block name section provides a 64-byte variable to identify the trusted block, just as key labels are used to identify other CCA keys. This name, or label, enables a host access-control system such as RACF® to use the name to verify that the application has authority to use the trusted block. No multiple sections are allowed.

   - A required information section (trusted block section identifier X'14')
     The trusted block information section contains control and security information related to the trusted block. The information section is required while the others are optional. This section contains the cryptographic information that guarantees its integrity and binds it to the local system. No multiple sections are allowed.

   - An optional application-defined data section (trusted block section identifier X'15')
     The trusted block application-defined data section can be used to include application-defined data in the trusted block. The purpose of the data in this section is defined by the application. CCA does not examine or use this data in any way. No multiple sections are allowed.

**Trusted block integrity**

An enciphered confounder and triple-length MAC key contained within the required information section of the trusted block is used to protect the integrity of the trusted block.

The randomly generated MAC key is used to calculate an ISO 16609 CBC mode TDES MAC of the trusted block contents. Together, the MAC key and MAC value provide a way to verify that the trusted block originated from an authorized source, and binds it to the local system.

An external trusted block has its MAC key enciphered under an IMP-PKA key-encrypting key. An internal trusted block has its MAC key enciphered under a variant of the PKA master key, and the master-key verification pattern is stored in the information section.

**Number representation in trusted blocks**

The number format in trusted blocks.

- All length fields are in binary
- All binary fields (exponents, lengths, and so forth) are stored with the high-order byte first (left, low-address, z/OS format); thus the least significant bits are to the right and preceded with zero-bits to the width of a field
Key token formats

- In variable-length binary fields that have an associated field-length value, leading bytes that would otherwise contain X'00' can be dropped and the field shortened to contain only the significant bits

**Trusted block sections**

At the beginning of every trusted block is a trusted block header.

The header contains the following information:
- A token identifier, which specifies if the token contains an external or internal key-token
- A token version number to allow for future changes
- A length in bytes of the trusted block, including the length of the header

The trusted block header is defined in Table 181.

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Token identifier (a flag that indicates token type)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'1E'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'1F'</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Token version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Length of the key-token structure in bytes.</td>
</tr>
<tr>
<td>004</td>
<td>004</td>
<td>Reserved, binary zero.</td>
</tr>
</tbody>
</table>

**Note:** See “Number representation in trusted blocks” on page 649.

Following the header, in no particular order, are trusted block sections. There are five different sections defined, each identified by a one-byte section identifier (X'11' - X'15'). Two of the five sections have subsections defined. A subsection is a tag-length-value (TLV) object, identified by a two-byte subsection tag.

Only sections X'12' and X'14' have subsections defined; the other sections do not. A section and its subsections, if any, are one contiguous unit of data. The subsections are concatenated to the related section, but are otherwise in no particular order.

Section X'12' has five subsections defined (X'0001' - X'0005'). Section X'14' has two subsections, (X'0001' and X'0002'). Of all the subsections, only subsection X'0001' of section X'14' is required. Section X'14' is also required.

The trusted block sections and subsections are described in detail in the following topics.

**Trusted block section X'11'**

Trusted block section X'11' contains the trusted RSA public key in addition to a key-usage flag indicating whether the public key is usable in key-management operations, digital signature operations, or both.

Section X'11' is optional. No multiple sections are allowed. It has no subsections defined.
This section is defined in Table 182.

Table 182. Trusted block trusted RSA public key section (X'11')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier: X'11' Trusted block trusted RSA public key</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length (16 + xxx + yyy).</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>RSA public key exponent field length in bytes, xxx.</td>
</tr>
<tr>
<td>008</td>
<td>002</td>
<td>RSA public key modulus length in bits.</td>
</tr>
<tr>
<td>010</td>
<td>002</td>
<td>RSA publickey modulus field length in bytes, yyy.</td>
</tr>
<tr>
<td>012</td>
<td>xxx</td>
<td>Public key exponent, e (this field length is typically 1, 3, or 64 - 512 bytes). e must be odd and 1 \leq e &lt; n. (e is frequently valued to 3 or 2^{16}+1 (=65537), otherwise e is of the same order of magnitude as the modulus).</td>
</tr>
<tr>
<td>Note:</td>
<td></td>
<td>Although the current product implementation does not generate such a public key, you can import an RSA public key having an exponent valued to two (2). Such a public key (a Rabin key) can correctly validate an ISO 9796-1 digital signature.</td>
</tr>
<tr>
<td>012 + xxx + yyy</td>
<td>004</td>
<td>Flags:</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>X'00000000'</td>
<td>Trusted block public key can be used in digital signature operations only</td>
<td></td>
</tr>
<tr>
<td>X'80000000'</td>
<td>Trusted block public key can be used in both digital signature and key management operations</td>
<td></td>
</tr>
<tr>
<td>X'C0000000'</td>
<td>Trusted block public key can be used in key management operations only</td>
<td></td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 649.

**Trusted block section X'12'**

Trusted block section X'12' contains information that defines a rule.

TA trusted block can have zero or more rule sections.

1. A trusted block with no rule sections can be used by the PKA Key Token Change and PKA Key Import verbs. A trusted block with no rule sections can be used by the Digital Signature Verify verb, provided there is an RSA public key section that has its key-usage flag set to allow digital signature operations.

2. At least one rule section is required when the Remote Key Export verb is used to:
   - Generate an RXX key-token
   - Export an RXX key-token
   - Export a CCA DES key-token
Key token formats

- Generate or export a key encrypted by a public key. The public key is contained in a vendor certificate and is the root certification key for the ATM vendor. It is used to verify the digital signature on public-key certificates for specific individual ATMs.

3. If a trusted block has multiple rule sections, each rule section must have a unique 8-character Rule ID.

Section X'12' is the only section that can have multiple sections. Section X'12' is optional.

Note: The overall length of the trusted block cannot exceed its maximum size of 3500 bytes.

Five subsections (TLV objects) are defined.

This section is defined in Table 183.

Table 183. Trusted block rule section (X'12')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier: X'12' Trusted block rule</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length in bytes (20 + yyy).</td>
</tr>
<tr>
<td>004</td>
<td>008</td>
<td>Rule ID (in ASCII). An 8-byte character string that uniquely identifies the rule within the trusted block. Valid ASCII characters are: A - Z, a - z, 0 - 9, - (hyphen), and _ (underscore), left-aligned and padded on the right with space characters.</td>
</tr>
<tr>
<td>012</td>
<td>004</td>
<td>Flags (undefined flag bits are reserved and must be zero). Value Description X'00000000' Generate new key X'00000001' Export existing key</td>
</tr>
<tr>
<td>016</td>
<td>001</td>
<td>Generated key length. Length in bytes of key to be generated when flags value (offset 012) is set to generate a new key; otherwise ignore this value. Valid values are 8, 16, or 24; return an error if not valid.</td>
</tr>
</tbody>
</table>
| 017            | 001            | Key-check algorithm identifier (all others are reserved and must not be used): Value Description X'00' Do not compute key-check value. Set the key_check_value_length variable to zero. X'01' Encrypt an 8-byte block of binary zeros with the key. See “Encrypt zeros DES-key verification algorithm” on page 701. X'02' Compute the MDC-2 hash of the key. See “Modification Detection Code calculation” on page 701.
## Key token formats

### Table 183. Trusted block rule section (X’12’) (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>018</td>
<td>001</td>
<td>Symmetric encrypted output key format flag (all other values are reserved and must not be used).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Return the indicated symmetric key-token using the <code>sym_encrypted_key_identifier</code> parameter.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>X'00'</td>
<td>Return an RKX key-token encrypted under a variant of the MAC key.</td>
<td></td>
</tr>
<tr>
<td>Note:</td>
<td>This key format is permitted when the flags value (offset 012) is set to either:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Generate a new key</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Export an existing key</td>
<td></td>
</tr>
<tr>
<td>X'01'</td>
<td>Return a CCA DES key-token encrypted under a transport key.</td>
<td></td>
</tr>
<tr>
<td>Note:</td>
<td>This key format is not permitted if the flags value (offset 012) is set to generate a new key; it is only permitted when exporting an existing key.</td>
<td></td>
</tr>
</tbody>
</table>

| 019            | 001            | Asymmetric encrypted output key format flag (all other values are reserved and must not be used). |
|                |                | Return the indicated asymmetric key-token in the `asym_encrypted_key` variable. |
| Value          | Description    |             |
| X'00'          | Do not return an asymmetric key. Set the `asym_encrypted_key_length` variable to zero. |
| X'01'          | Output in PKCS-1.2 format. |
| X'02'          | Output in RSA-OAEP format. |

| 020            | yyy            | Rule section subsections (tag-length-value objects). A series of zero - five objects in TLV format. |

### Note:
See [“Number representation in trusted blocks” on page 649](#).

### Trusted block section X’12’ subsections:

Section X’12’ has five rule subsections (tag-length-value objects) defined.

These subsections are summarized in [Table 184](#).

### Table 184. Summary of trusted block X’12’ subsections

<table>
<thead>
<tr>
<th>Rule subsection tag</th>
<th>TLV object</th>
<th>Optional or required</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>X’0001</td>
<td>Transport key variant</td>
<td>Optional</td>
<td>Contains variant to be XORed into the cleartext transport key.</td>
</tr>
<tr>
<td>X’0002</td>
<td>Transport key rule reference</td>
<td>Optional; required to use an RKX key-token as a transport key</td>
<td>Contains the rule ID for the rule that must have been used to create the transport key.</td>
</tr>
<tr>
<td>X’0003</td>
<td>Common export key parameters</td>
<td>Optional for key generation; required for key export of an existing key</td>
<td>Contains the export key and source key minimum and maximum lengths, an output key variant length and variant, a CV length, and a CV to be XORed with the cleartext transport key to control usage of the key.</td>
</tr>
</tbody>
</table>
### Key token formats

#### Table 184. Summary of trusted block X'12' subsections (continued)

<table>
<thead>
<tr>
<th>Rule subsection tag</th>
<th>TLV object</th>
<th>Optional or required</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0004'</td>
<td>Source key reference</td>
<td>Optional; required if the source key is an RKX key-token</td>
<td>Contains the rule ID for the rule used to create the source key. <strong>Note:</strong> Include all rules that will ever be needed when a trusted block is created. A rule cannot be added to a trusted block after it has been created.</td>
</tr>
<tr>
<td>X'0005'</td>
<td>Export key CCA token parameters</td>
<td>Optional; used for export of CCA DES key tokens only</td>
<td>Contains mask length, mask, and CV template to limit the usage of the exported key. Also contains the template length and template that defines which source key labels are allowed. The key type of a source key input parameter can be &quot;filtered&quot; by using the export key CV limit mask (offset 005) and limit template (offset 005 + yyy) in this subsection.</td>
</tr>
</tbody>
</table>

**Note:** See "Number representation in trusted blocks" on page 649.

#### Trusted block section X'12' subsection X'0001'

Subsection X'0001' of the trusted block rule section (X'12') is the transport key variant TLV object. This subsection is optional. It contains a variant to be XORed into the cleartext transport key.

This subsection is defined in Table 185.

#### Table 185. Transport key variant subsection (X'0001') of trusted block rule section (X'12')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Subsection tag: X'0001' Transport key variant TLV object</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Subsection length in bytes (8 + nnn).</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Subsection version number (X'00').</td>
</tr>
<tr>
<td>005</td>
<td>002</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>007</td>
<td>001</td>
<td>Length of variant field in bytes (nnn). This length must be greater than or equal to the length of the transport key that is identified by the transport_key_identifier parameter. If the variant is longer than the key, truncate it on the right to the length of the key prior to use.</td>
</tr>
<tr>
<td>008</td>
<td>nnn</td>
<td>XOR this variant into the cleartext transport key, provided: (1) the length of the variant field value (offset 007) is not zero, and (2) the symmetric encrypted output key format flag (offset 018 in section X'12') is X'01'. <strong>Note:</strong> A transport key is not used when the symmetric encrypted output key is in RKX key-token format.</td>
</tr>
</tbody>
</table>

**Note:** See "Number representation in trusted blocks" on page 649.

#### Trusted block section X'12' subsection X'0002'

Subsection X'0002' of the trusted block rule section (X'12') is the transport key rule reference TLV object. This subsection is optional. It contains the rule ID for the rule...
that must have been used to create the transport key. This subsection must be present to use an RKX key-token as a transport key.

This subsection is defined in Table 186.

Table 186. Transport key rule reference subsection (X'0002') of trusted block rule section (X'12')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Subsection tag: X'0002' Transport key rule reference TLV object</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Subsection length in bytes (14).</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Subsection version number (X'00').</td>
</tr>
<tr>
<td>005</td>
<td>001</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>006</td>
<td>008</td>
<td>Rule ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the rule identifier for the rule that must have been used to create the RKX key-token used as the transport key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Rule ID is an 8-byte string of ASCII characters, left-aligned and padded on the right with space characters. Acceptable characters are A - Z, a - z, 0 - 9, -(X'2D'), and _ (X'5F'). All other characters are reserved for future use.</td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 649.

Trusted block section X'12' subsection X'0003'

Subsection X'0003' of the trusted block rule section X'12') is the common export key parameters TLV object. This subsection is optional, but is required for the key export of an existing source key (identified by the source_key_identifier parameter) in either RKX key-token format or CCA DES key-token format. For new key generation, this subsection applies the output key variant to the cleartext generated key, if such an option is desired. It contains the input source key and output export key minimum and maximum lengths, an output key variant length and variant, a CV length, and a CV to be XORed with the cleartext transport key.

This subsection is defined in Table 187.

Table 187. Common export key parameters subsection (X'0003') of trusted block rule section (X'12')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Subsection tag: X'0003' Common export key parameters TLV object</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Subsection length in bytes (12 + xxx + yyy).</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Subsection version number (X'00').</td>
</tr>
<tr>
<td>005</td>
<td>002</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>007</td>
<td>001</td>
<td>Flags (must be set to binary zero).</td>
</tr>
<tr>
<td>008</td>
<td>001</td>
<td>Export key minimum length in bytes. Length must be 0, 8, 16, or 24.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Also applies to the source key. Not applicable for key generation.</td>
</tr>
<tr>
<td>009</td>
<td>001</td>
<td>Export key maximum length in bytes (yyyy). Length must be 0, 8, 16, or 24.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Also applies to the source key. Not applicable for key generation.</td>
</tr>
</tbody>
</table>
### Key token formats

#### Table 187. Common export key parameters subsection (X'0003') of trusted block rule section (X'12') (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>001</td>
<td>Output key variant length in bytes (xxx). Valuables are 0 or 8 - 255. If greater than 0, the length must be at least as long as the longest key ever to be exported using this rule. If the variant is longer than the key, truncate it on the right to the length of the key prior to use. Note: The output key variant (offset 011) is not used if this length is zero.</td>
</tr>
<tr>
<td>011</td>
<td>xxx</td>
<td>Output key variant. The variant can be any value. XOR this variant into the cleartext value of the output key.</td>
</tr>
<tr>
<td>011 + xxx</td>
<td>001</td>
<td>CV length in bytes (yyy). - If the length is not 0, 8, or 16, return an error. - If the length is 0, and if the source key is a CCA DES key-token, preserve the CV in the symmetric encrypted output if the output is to be in the form of a CCA DES key-token. - If a nonzero length is less than the length of the key identified by the source_key_identifier parameter, return an error. - If the length is 16, and if the CV (offset 012 + xxx) is valued to 16 bytes of X'00' (ignoring the key-part bit), then: 1. Ignore all CV bit definitions 2. If CCA DES key-token format, set the flag byte of the symmetric encrypted output key to indicate a CV value is present. 3. If the source key is eight bytes in length, do not replicate the key to 16 bytes</td>
</tr>
<tr>
<td>012 + xxx</td>
<td>yyy</td>
<td>CV. (See “Control vector table” on page 667.) Place this CV into the output exported key-token, provided that the symmetric encrypted output key format selected (offset 018 in rule section) is CCA DES key-token. - If the symmetric encrypted output key format flag (offset 018 in section X'12') indicates return an RKX key-token (X'00'), then ignore this CV. Otherwise, XOR this CV into the cleartext transport key. - XOR the CV of the source key into the cleartext transport key if the CV length (offset 011 + xxx) is set to 0. If a transport key to encrypt a source key has equal left and right key halves, return an error. Replicate the key halves of the key identified by the source_key_identifier parameter whenever all of these conditions are met: 1. The Key Generate - SINGLE-R command (offset X'00DB') is enabled in the active role 2. The CV length (offset 011 + xxx) is 16, and both CV halves are nonzero 3. The source_key_identifier parameter (contained in either a CCA DES key-token or RKX key-token) identifies an 8-byte key 4. The key-form bits (40 - 42) of this CV do not indicate a single-length key (are not set to zero) 5. Key-form bit 40 of this CV does not indicate the key is to have guaranteed unique halves (is not set to B'1'). See “Key Form Bits, ‘fff’” on page 673. Note: A transport key is not used when the symmetric encrypted output key is in RKX key-token format.</td>
</tr>
</tbody>
</table>

**Note:** See “Number representation in trusted blocks” on page 649.

#### Trusted block section X'12' subsection X'0004'

Subsection X'0004' of the trusted block rule section (X'12') is the source key rule reference TLV object. This subsection is optional, but is required if using an RKX key-token as a source key (identified by source_key_identifier parameter). It contains
the rule ID for the rule used to create the export key. If this subsection is not present, an RKX key-token format source key will not be accepted for use.

This subsection is defined in Table 188.

Table 188. Source key rule reference subsection (X'0004') of trusted block rule section (X'12')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Subsection tag: Source key rule reference TLV object</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Subsection length in bytes (14).</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Subsection version number (X'00').</td>
</tr>
<tr>
<td>005</td>
<td>001</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>006</td>
<td>008</td>
<td>Rule ID. Rule identifier for the rule that must have been used to create the source key. The Rule ID is an 8-byte string of ASCII characters, left-aligned and padded on the right with space characters. Acceptable characters are A - Z, a - z, 0 - 9, - (X'2D'), and _ (X'5F'). All other characters are reserved for future use.</td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 649.

Trusted block section X'12' subsection X'0005'

Subsection X'0005' of the trusted block rule section (X'12') is the export key CCA token parameters TLV object. This subsection is optional. It contains a mask length, mask, and template for the export key CV limit. It also contains the template length and template for the source key label. When using a CCA DES key-token as a source key input parameter, its key type can be "filtered" by using the export key CV limit mask (offset 005) and limit template (offset 005+yyy) in this subsection.

This subsection is defined in Table 189.

Table 189. Export key CCA token parameters subsection (X'0005') of trusted block rule section (X'12')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Subsection tag: Export key CCA token parameters TLV object</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Subsection length in bytes (8 + yyy + yyy + zzz).</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Subsection version number (X'00').</td>
</tr>
<tr>
<td>005</td>
<td>002</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>007</td>
<td>001</td>
<td>Flags (must be set to binary zero).</td>
</tr>
<tr>
<td>008</td>
<td>001</td>
<td>Export key CV limit mask length in bytes (yyy). Do not use CV limits if this CV limit mask length (yyy) is zero. Use CV limits if yyy is nonzero, in which case yyy:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Must be 8 or 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Must not be less than the export key minimum length (offset 008 in subsection X'0003')</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Must be equal in length to the actual source key length of the key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: An export key minimum length of 16 and an export key CV limit mask length of 8 returns an error.</td>
</tr>
</tbody>
</table>
### Key token formats

#### Table 189. Export key CCA token parameters subsection (X'0005') of trusted block rule section (X'12') (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
</table>
| 009           | yyy           | Export key CV limit mask (does not exist if yyy=0).  
See “Control-vector-base bit maps” on page 669.  
Indicates which CV bits to check against the source key CV limit template (offset 009 + yyy).  
**Examples:** A mask of X'FF' means check all bits in a byte. A mask of X'FE' ignores the parity bit in a byte. |
| 009 + yyy     | yyy           | Export key CV limit template (does not exist if yyy = 0).  
Specifies the required values for those CV bits that are checked based on the export key CV limit mask (offset 009). (See “Control-vector-base bit maps” on page 669.)  
The export key CV limit mask and template have the same length, yyy. This is because these two variables work together to restrict the acceptable CVs for CCA DES key tokens to be exported. The checks work as follows:  
1. If the length of the key to be exported is less than yyy, return an error  
2. Logical AND the CV for the key to be exported with the export key CV limit mask  
3. Compare the result to the export key CV limit template  
4. Return an error if the comparison is not equal  
**Examples:** An export key CV limit mask of X'FF' for CV byte 1 (key type) along with an export key CV limit template of X'3F' (key type CVARENC) for byte 1 filters out all key types except CVARENC keys.  
**Note:** Using the mask and template to permit multiple key types is possible, but cannot consistently be achieved with one rule section. For example, setting bit 10 to B'1' in the mask and the template permits PIN processing keys and cryptographic variable encrypting keys, and only those keys. However, a mask to permit PIN-processing keys and key-encrypting keys, and only those keys, is not possible. In this case, multiple rule sections are required, one to permit PIN-processing keys and the other to permit key-encrypting keys. |
| 009 + yyy + yyy | 001 | Source key label template length in bytes (zzz).  
Valid values are 0 and 64. Return an error if the length is 64 and a source key label is not provided. |
| 010 + yyy + yyy | zzz | Source key label template (does not exist if zzz = 0).  
If a key label is identified by the source_key_identifier parameter, verify that the key label name matches this template. If the comparison fails, return an error.  
The source key label template must conform to the following rules:  
• The key label template must be 64 bytes in length  
• The first character cannot be in the range X'00' - X'1F', nor can it be X'FF'  
• The first character cannot be numeric (X'30' - X'39')  
• A key label name is terminated by a space character (X'20') on the right and must be padded on the right with space characters  
• The only special characters permitted are #, $, @, and * (X'23', X'24', X'40', and X'2A')  
• The wildcard X'2A' (*) is permitted only as the first character, the last character, or the only character in the template  
• Only alphanumeric characters (a - z, A - Z, 0 - 9), the four special characters (X'23', X'24', X'40', and X'2A'), and the space character (X'20') are allowed |

**Note:** See “Number representation in trusted blocks” on page 649.
Key token formats

**Trusted block section X'13'**
Trusted block section X'13' contains the name (key label).

**Trusted block section X'13'**
The trusted block name section provides a 64-byte variable to identify the trusted block, just as key labels are used to identify other CCA keys. This name, or label, enables a host access-control system such as RACF to use the name to verify that the application has authority to use the trusted block.

Section X'13' is optional. No multiple sections are allowed. It has no subsections defined. This section is defined in Table 190.

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'13' Trusted block name (key label)</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length in bytes (68).</td>
</tr>
<tr>
<td>004</td>
<td>064</td>
<td>Name (key label).</td>
</tr>
</tbody>
</table>

**Note:** See “Number representation in trusted blocks” on page 649.

**Trusted block section X'14'**
Trusted block section X'14' contains control and security information related to the trusted block.

This information section is separate from the public key and other sections because this section is required while the others are optional. This section contains the cryptographic information that guarantees its integrity and binds it to the local system.

Section X'14' is required. No multiple sections are allowed. Two subsections are defined. This section is defined in Table 191.

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'14' Trusted block information</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length in bytes (10+xxx).</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Reserved, binary zero.</td>
</tr>
<tr>
<td>006</td>
<td>004</td>
<td>Flags:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00000000'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'00000001'</td>
</tr>
<tr>
<td>010</td>
<td>xxx</td>
<td>Information section subsections (tag-length-value objects).</td>
</tr>
</tbody>
</table>
Key token formats

Note: See “Number representation in trusted blocks” on page 649.

Trusted block section X'14' subsections:

Section X'14' has two information subsections (tag-length-value objects) defined.

These subsections are summarized in Table 192.

Table 192. Summary of trusted block information subsections

<table>
<thead>
<tr>
<th>Rule subsection tag</th>
<th>TLV object</th>
<th>Optional or required</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0001'</td>
<td>Protection information</td>
<td>Required</td>
<td>Contains the encrypted 8-byte confounder and triple-length (24-byte) MAC key, the ISO-16609 TDES CBC MAC value, and the MKVP of the PKA master key (computed using MDC4).</td>
</tr>
<tr>
<td>X'0002'</td>
<td>Activation and expiration dates</td>
<td>Optional</td>
<td>Contains flags indicating whether or not the coprocessor is to validate dates, and contains the activation and expiration dates that are considered valid for the trusted block.</td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 649.

Trusted block section X'14' subsection X'0001'

Subsection X'0001' of the trusted block information section (X'14') is the protection information TLV object. This subsection is required. It contains the encrypted 8-byte confounder and triple-length (24-byte) MAC key, the ISO-16609 TDES CBC MAC value, and the MKVP of the PKA master key (computed using MDC4).

This subsection is defined in Table 193.

Table 193. Protection information subsection (X'0001') of trusted block information section (X'14')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Subsection tag: X'0001' Trusted block information TLV object</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Subsection length in bytes (62).</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Subsection version number (X'00').</td>
</tr>
<tr>
<td>005</td>
<td>001</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>006</td>
<td>032</td>
<td>Encrypted MAC key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the encrypted 8-byte confounder and triple-length (24-byte) MAC key in the following format:</td>
</tr>
<tr>
<td>00 - 07</td>
<td></td>
<td>Confounder</td>
</tr>
<tr>
<td>08 - 15</td>
<td></td>
<td>Left key</td>
</tr>
<tr>
<td>16 - 23</td>
<td></td>
<td>Middle key</td>
</tr>
<tr>
<td>24 - 31</td>
<td></td>
<td>Right key</td>
</tr>
<tr>
<td>038</td>
<td>008</td>
<td>MAC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the ISO-16609 TDES CBC Message Authentication Code value.</td>
</tr>
</tbody>
</table>
Table 193. Protection information subsection (X‘0001’) of trusted block information section (X‘14’) (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>046</td>
<td>016</td>
<td>MKVP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the PKA master-key verification pattern, computed using MDC4, when the trusted block is in internal form, otherwise contains binary zero.</td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 649.

Trusted block section X’14’ subsection X’0002’

Subsection X‘0002’ of the trusted block information section (X’14’) is the activation and expiration dates TLV object. This subsection is optional. It contains flags indicating whether or not the coprocessor is to validate dates, and contains the activation and expiration dates that are considered valid for the trusted block.

This subsection is defined in Table 194.

Table 194. Activation and expiration dates subsection (X‘0002’) of trusted block information section (X’14’)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>002</td>
<td>Subsection tag: X‘0002’ Activation and expiration dates TLV object</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Subsection length in bytes (16).</td>
</tr>
<tr>
<td>004</td>
<td>001</td>
<td>Subsection version number (X’00’).</td>
</tr>
<tr>
<td>005</td>
<td>001</td>
<td>Reserved, must be binary zero.</td>
</tr>
<tr>
<td>006</td>
<td>002</td>
<td>Flags:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X‘0000’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X‘0001’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>008</td>
<td>004</td>
<td>Activation date.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the first date that the trusted block can be used for generating or exporting keys. Format of the date is YYMDD, where:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>012</td>
<td>004</td>
<td>Expiration date.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the last date that the trusted block can be used. Same format as activation date (offset 008). Return an error if date is not valid.</td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 649.

Trusted block section X’15’

Trusted block section X’15’ contains application-defined data.
The trusted block application-defined data section can be used to include application-defined data in the trusted block. The purpose of the data in this section is defined by the application; it is neither examined nor used by CCA in any way.

Section X'15' is optional. No multiple sections are allowed. It has no subsections defined. This section is defined in Table 195.

Table 195. Trusted block application-defined data section (X'15')

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>Section identifier: X'15' Application-defined data</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>Section version number (X'00').</td>
</tr>
<tr>
<td>002</td>
<td>002</td>
<td>Section length (6 + xxx)</td>
</tr>
<tr>
<td>004</td>
<td>002</td>
<td>Application data length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value of xxx must be between 0 and N, where N does not cause the overall length of the trusted block to exceed its maximum size of 3500 bytes.</td>
</tr>
<tr>
<td>006</td>
<td>xxx</td>
<td>Application-defined data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Could be used to hold a public-key certificate for the trusted public key.</td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 649.
Appendix C. Key forms and types used in the Key Generate verb

This appendix provides examples of the key forms and key types used in the Key Generate verb.

The Key Generate verb is the most complex of all the CCA verbs.

Generating an operational key

There are different methods that you can use to generate an operational key.

Choose one of the following methods:

- For operational keys, call the Key Generate (CSNBKGN) verb. Table 32 on page 174 and Table 33 on page 174 show the key type and key form combinations for a single key and for a key pair.
- For data-encrypting keys, call the Random Number Generate (CSNBRNG) verb and specify the form parameter as ODD. Then pass the generated value to the Clear Key Import (CSNBCKI) verb or the Multiple Clear Key Import (CSNBCKM) verb. The DATA key type is now in operational form.

You cannot generate a PIN verification (PINVER) key in operational form because the originator of the PIN generation (PINGEN) key generates the PINVER key in exportable form, which is sent to you to be imported.

Generating an importable key

To generate an importable key form, call the Key Generate (CSNBKGN) verb.

If you want a DATA, MAC, PINGEN, DATAM, or DATAC key type in importable form, obtain it directly by generating a single key. If you want any other key type in importable form, request a key pair where either the first or second key type is importable (IM). Discard the generated key form that you do not need.

Generating an exportable key

To generate an exportable key form, call the Key Generate (CSNBKGN) verb.

If you want a DATA, MAC, PINGEN, DATAM, or DATAC key type in exportable form, obtain it directly by generating a single key. If you want any other key type in exportable form, request a key pair where either the first or second key type is exportable (EX). Discard the generated key form that you do not need.

Examples of single-length keys in one form only

An example of single-length keys.

<table>
<thead>
<tr>
<th>Key Form</th>
<th>Key Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP DATA</td>
<td>Encipher or Decipher data. Use Data Key Export or Key Export to send encrypted key to another cryptographic partner. Then communicate the ciphertext.</td>
<td>OP DATA</td>
</tr>
</tbody>
</table>
OP  MAC  MAC Generate. Because no MACVER key exists, there is no secure communication of the MAC with another cryptographic partner.

IM  DATA  Key Import, and then Encipher or Decipher. Then Key Export to communicate ciphertext and key with another cryptographic partner.

EX  DATA  You can send this key to a cryptographic partner, but you can do nothing with it directly. Use it for the key distribution service. The partner could then use Key Import to get it in operational form, and use it as in OP DATA above.

Examples of OPMI single-length, double-length, and triple-length keys in two forms

The first two letters of the key form indicate the form that key type 1 parameter is in, and the second two letters indicate the form that key type 2 parameter is in.

<table>
<thead>
<tr>
<th>Key Type Type</th>
<th>Form 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPMI DATA DATA</td>
<td>Use the OP form in Encipher. Use Key Export with the OP form to communicate ciphertext and key with another cryptographic partner. Use Key Import at a later time to use Encipher or Decipher with the same key again.</td>
</tr>
<tr>
<td>OPMI MAC MAC</td>
<td>Single-length MAC Generate key. Use the OP form in MAC Generate. You have no corresponding verb MACVER key, but you can call the MAC Verify verb with the MAC key directly. Use the Key Import verb and then compute the MAC again using the MAC Verify verb, which compares the MAC it generates with the MAC supplied with the message and issues a return code indicating whether they compare.</td>
</tr>
</tbody>
</table>

Examples of OPEX single-length, double-length, and triple-length keys in two forms

Examples of OPEX single-length, double-length, and triple-length keys in two forms.

<table>
<thead>
<tr>
<th>Key Type Type</th>
<th>Form 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEX DATA DATA</td>
<td>Use the OP form in Encipher. Send the EX form and the ciphertext to another cryptographic partner.</td>
</tr>
<tr>
<td>OPEX MAC MAC</td>
<td>Single-length MAC generation key. Use the OP form in both MAC Generate and MAC Verify. Send the EX form to a cryptographic partner to be used in the MAC Generate or MAC Verify verbs.</td>
</tr>
<tr>
<td>OPEX MAC MACVER</td>
<td>Single-length MAC generation and MAC verification keys. Use the OP form in MAC Generate. Send the EX form to a cryptographic partner where it will be put into Key Import, and then MAC Verify, with the message and MAC that you have also transmitted.</td>
</tr>
<tr>
<td>OPEX PINGEN PINVER</td>
<td>Use the OP form in Clear PIN Generate. Send the EX form to a cryptographic partner where it is put into Key Import, and then Encrypted PIN Verify, along with an IPINENC key.</td>
</tr>
<tr>
<td>OPEX IMPORTER EXPORTER</td>
<td>Use the OP form in Key Import or Key Generate. Send the EX form to a cryptographic partner where it is used in Key Export, Data Key Export, or Key Generate, or put in the CCA key storage file.</td>
</tr>
</tbody>
</table>
OPEX IMPORTER

Use the OP form in Key Export, Data Key Export, or Key Generate. Send the EX form to a cryptographic partner where it is put into the CCA Key storage file or used in Key Import or Key Generate.

When you and your partner have the OPEX IMPORTER EXPORTER, OPEX EXPORTER IMPORTER pairs of keys in "Examples of OPEX single-length, double-length, and triple-length keys in two forms" on page 664 installed, you can start key and data exchange.

Examples of IMEX single-length and double-length keys in two forms

Examples of IMEX single-length and double-length keys in two forms.

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form 1</td>
<td>Form 2</td>
<td></td>
</tr>
</tbody>
</table>

IMEX DATA DATA Use the Key Import verb to import IM form and use the OP form in Encipher. Send the EX form to a cryptographic partner.

IMEX MAC MACVER Use the Key Import verb to import the IM form and use the OP form in MAC Generate. Send the EX form to a cryptographic partner who can verify the MAC.

IMEX IMPORTER EXPORTER Use the Key Import verb to import the IM form and send the EX form to a cryptographic partner. This establishes a new IMPORTER/EXPORTER key between you and your partner.

IMEX PINGEN PINVER Use the Key Import verb to import the IM form and send the EX form to a cryptographic partner. This establishes a new PINGEN/PINVER key between you and your partner.

Examples of EXEX single-length and double-length keys in two forms

Examples of IMEX single-length and double-length keys in two forms.

For the keys shown in the following list, you are providing key distribution services for other nodes in your network, or other cryptographic partners. Neither key type can be used in your installation.

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form 1</td>
<td>Form 2</td>
<td></td>
</tr>
</tbody>
</table>

EXEX DATA DATA Send the first EX form to a cryptographic partner with the corresponding IMPORTER and

EXEX MAC MACVER send the second EX form to another cryptographic partner with the corresponding IMPORTER. This exchange establishes a key between two partners.
Appendix D. Control vectors and changing control vectors with the Control Vector Translate verb

This appendix contains a control vector table, which displays the default value of the control vector associated with each type of key.

This appendix also describes how to change control vectors with the Control Vector Translate verb

Control vector table

This topic describes the default control vector values.

**Note:** The control vectors descriptions here build on the descriptions used for earlier IBM products supporting CCA, each in turn: 4765, 4764, 4758, and TSS.

The master key enciphers all keys operational on your system. A transport key enciphers keys distributed off your system. Before a master key or transport key enciphers a key, CCA XORs both halves of the master key or transport key with a control vector. The same control vector is XORed to the left and right half of a master key or transport key.

Also, if you are entering a key part, CCA XORs each half of the key part with a control vector before placing the key part into the key storage file.

Each type of CCA key (except the master key) has either one or two unique control vectors associated with it. The master key or transport key CCA XORs with the control vector depending on the type of key the master key or transport key is enciphering. For double-length keys, a unique control vector exists for each half of a specific key type. For example, there is a control vector for the left half of an input PIN-encrypting key, and a control vector for the right half of an input PIN-encrypting key.

If you are entering a cleartext key part, CCA XORs the key part with the unique control vector(s) associated with the key type. CCA also enciphers the key part with two master key variants for a key part. One master key variant enciphers the left half of the key part and another master key variant enciphers the right half of the key part. CCA creates the master key variants for a key part by XORing the master key with the control vectors for key parts. These procedures protect key separation.

Table 196 displays the default value of the control vector associated with each type of key. Some key types do not have a default control vector. For keys that are double-length, CCA enciphers using a unique control vector on each half.

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Control Vector Value (Hex) Value for Single-length Key or Left Half of Double-length Key</th>
<th>Control Vector Value (Hex) Value for Right Half of Double-length Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>Key Type</td>
<td>Control Vector Value (Hex) Value for Single-length Key or Left Half of Double-length Key</td>
<td>Control Vector Value (Hex) Value for Right Half of Double-length Key</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>AESTOKEN</td>
<td>00 00 00 00 00 00 00 00 00 00 00</td>
<td>00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>CIPHER</td>
<td>00 03 71 00 03 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CIPHER (double length)</td>
<td>00 03 71 00 03 41 00 00 00 00 03 71 00 03 21 00 00</td>
<td></td>
</tr>
<tr>
<td>CVARDEC</td>
<td>00 3F 42 00 03 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CVARENC</td>
<td>00 3F 48 00 03 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CVARPINE</td>
<td>00 3F 41 00 03 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CVARXCVL</td>
<td>00 3F 44 00 03 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CVARXCVR</td>
<td>00 3F 47 00 03 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>*DATA</td>
<td>00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>DATAC</td>
<td>00 00 71 00 03 41 00 00 00 00 00</td>
<td>00 00 71 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*DATAM generation key (external)</td>
<td>00 00 4D 00 03 41 00 00 00 00 00 00 4D 00 03 21 00 00</td>
<td>00 00 4D 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*DATAM key (internal)</td>
<td>00 05 4D 00 03 00 00 00 00 00 00 05 4D 00 03 21 00 00</td>
<td>00 05 4D 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*DATAMV MAC verification key (external)</td>
<td>00 00 44 00 03 41 00 00 00 00 00 44 00 03 21 00 00</td>
<td>00 00 44 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*DATAMV MAC verification key (internal)</td>
<td>00 05 44 00 03 00 00 00 00 00 05 44 00 03 00 00 00</td>
<td>00 05 44 00 03 00 00 00 00 00</td>
</tr>
<tr>
<td>*DATAXLAT</td>
<td>00 06 71 00 03 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>DECRIPHER</td>
<td>00 03 50 00 03 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>DECRIPHER (double-length)</td>
<td>00 03 50 00 03 41 00 00 00 00 03 50 00 03 21 00 00</td>
<td>00 03 50 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>DKEYGENKY</td>
<td>00 71 44 00 00 03 41 00 00 00 00 71 44 00 03 21 00 00</td>
<td>00 71 44 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>DKEYL0</td>
<td>This control vector has the DKEYL0 set by default.</td>
<td></td>
</tr>
<tr>
<td>DKEYL1</td>
<td>00 72 44 00 00 03 41 00 00 00 00 72 44 00 00 03 21 00 00</td>
<td>00 72 44 00 00 03 21 00 00 00</td>
</tr>
<tr>
<td>DKEYL2</td>
<td>00 74 44 00 00 03 41 00 00 00 00 74 44 00 00 03 21 00 00</td>
<td>00 74 44 00 00 03 21 00 00 00</td>
</tr>
<tr>
<td>DKEYL3</td>
<td>00 77 44 00 00 03 41 00 00 00 00 77 44 00 00 03 21 00 00</td>
<td>00 77 44 00 00 03 21 00 00 00</td>
</tr>
<tr>
<td>DKEYL4</td>
<td>00 78 44 00 00 03 41 00 00 00 00 78 44 00 00 03 21 00 00</td>
<td>00 78 44 00 00 03 21 00 00 00</td>
</tr>
<tr>
<td>DKEYL5</td>
<td>00 7B 44 00 00 03 41 00 00 00 00 7B 44 00 00 03 21 00 00</td>
<td>00 7B 44 00 00 03 21 00 00 00</td>
</tr>
<tr>
<td>DKEYL6</td>
<td>00 7D 44 00 00 03 41 00 00 00 00 7D 44 00 00 03 21 00 00</td>
<td>00 7D 44 00 00 03 21 00 00 00</td>
</tr>
<tr>
<td>DKEYL7</td>
<td>00 7E 44 00 00 03 41 00 00 00 00 7E 44 00 00 03 21 00 00</td>
<td>00 7E 44 00 00 03 21 00 00 00</td>
</tr>
<tr>
<td>ENCIPHER</td>
<td>00 03 60 00 03 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>ENCIPHER (double-length)</td>
<td>00 03 60 00 03 41 00 00 00 00 03 60 00 03 21 00 00</td>
<td>00 03 60 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*EXPORTER</td>
<td>00 41 7D 00 03 41 00 00 00 00 41 7D 00 03 21 00 00</td>
<td>00 41 7D 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>00 42 42 00 03 41 00 00 00 00 02 42 42 00 03 21 00 00</td>
<td>00 42 42 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*IMP-PKA</td>
<td>00 42 05 00 03 41 00 00 00 02 42 05 00 03 21 00 00</td>
<td>00 42 05 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*IMPORTER</td>
<td>00 42 7D 00 03 41 00 00 00 00 02 42 7D 00 03 21 00 00</td>
<td>00 42 7D 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*IPINENC</td>
<td>00 21 5F 00 03 41 00 00 00 00 21 5F 00 03 21 00 00</td>
<td>00 21 5F 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*MAC</td>
<td>00 05 4D 00 03 00 00 00 00 05 4D 00 03 21 00 00</td>
<td>00 05 4D 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>MAC (double-length)</td>
<td>00 05 4D 00 03 41 00 00 00 05 4D 00 03 21 00 00</td>
<td>00 05 4D 00 03 21 00 00 00 00</td>
</tr>
<tr>
<td>*MACVER</td>
<td>00 05 44 00 03 00 00 00 00 05 44 00 03 21 00 00</td>
<td>00 05 44 00 03 21 00 00 00 00</td>
</tr>
</tbody>
</table>
### Table 196. Default control vector values (continued)

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Control Vector Value (Hex) Value for Single-length Key or Left Half of Double-length Key</th>
<th>Control Vector Value (Hex) Value for Right Half of Double-length Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACVER (double-length)</td>
<td>00 05 44 00 03 41 00 00</td>
<td>00 05 44 00 03 21 00 00</td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>00 41 42 00 03 41 00 00</td>
<td>00 41 42 00 03 21 00 00</td>
</tr>
<tr>
<td>*OPINENC</td>
<td>00 24 77 00 03 41 00 00</td>
<td>00 24 77 00 03 21 00 00</td>
</tr>
<tr>
<td>*PINGEN</td>
<td>00 22 7E 00 03 41 00 00</td>
<td>00 22 7E 00 03 21 00 00</td>
</tr>
<tr>
<td>*PINVER</td>
<td>00 22 42 00 03 41 00 00</td>
<td>00 22 42 00 03 21 00 00</td>
</tr>
<tr>
<td>SECMSG with SMPIN set</td>
<td>00 0A 50 00 03 41 00 00</td>
<td>00 0A 50 00 03 21 00 00</td>
</tr>
<tr>
<td>SECMSG with SMKEY set</td>
<td>00 0A 60 00 03 41 00 00</td>
<td>00 0A 60 00 03 21 00 00</td>
</tr>
</tbody>
</table>

**Note:** The external control vectors for DATAC, DATAM MAC generation, and DATAMV MAC verification keys are also referred to as data compatibility control vectors.

### Control-vector-base bit maps

This topic describes the control vector base bit maps.
Figure 5. Control vector base bit map (common bits and key-encrypting keys)
### Control-Vector Base Bits

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 1 1 1</td>
<td>1 1 2 2</td>
<td>2 2 3 3</td>
<td>3 3 3 3</td>
<td>4 4 4 4</td>
<td>4 5 5 5</td>
<td>5 5 6 6</td>
<td></td>
</tr>
<tr>
<td>0 2 4 6</td>
<td>0 8 0 2</td>
<td>4 6 0 2</td>
<td>2 4 6 8</td>
<td>0 2 4 6</td>
<td>8 0 2 4</td>
<td>6 8 0 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Data Operation Keys**

- **DATA**
  - 00000000 00000000 00000000 00000011 fff00000 00000000 00000000
- **DATAC**
  - 00000000 00000000 00110000 00000000 00000011 fff00000 00000000 00000000
- **DATAM**
  - 00000000 00000000 00001000 00000000 00000011 fff00000 00000000 00000000
- **DATAMV**
  - 00000000 00000000 00000100 00000000 00000011 fff00000 00000000 00000000
- **CIPHER**
  - 00000000 00000011 00110000 00000000 00000011 fff00000 00000000 00000000
- **DECLIPHER**
  - 00000000 00000011 00010000 00000000 00000011 fff00000 00000000 00000000
- **ENCIPHER**
  - 00000000 00000011 00100000 00000000 00000011 fff00000 00000000 00000000
- **SECMGS**
  - 00000000 00010100 00000000 00000011 fff00000 00000000 00000000
- **MAC**
  - 00000000 00000101 00000100 00000000 00000011 fff00000 00000000 00000000
- **MACVER**
  - 00000000 00000101 00000100 00000000 00000011 fff00000 00000000 00000000

**Legend**

- **Key-Form**
- **PIN encryption**
- **10 Key encryption**

---

**Figure 6. Control vector base bit map (data operation keys)**

---

Appendix D. Control vectors and changing control vectors with the Control Vector Translate verb 671
**Control-Vector Base Bits**

<table>
<thead>
<tr>
<th>0 0 0</th>
<th>0 1 1</th>
<th>1 1 2</th>
<th>2 2 2</th>
<th>3 3 3</th>
<th>4 4 4</th>
<th>5 5 5</th>
<th>5 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 4</td>
<td>6 8 0</td>
<td>2 6 8</td>
<td>0 2 4</td>
<td>8 0 2</td>
<td>6 8 0</td>
<td>2 6 8</td>
<td>0 2 4</td>
</tr>
</tbody>
</table>

Most Significant Bit

**PIN Processing Keys**

- 0000 NG-SPEC
- 0001 BM-PIN/BM-PINO
- 0010 VISA-PVV
- 0011 INBK-PIN
- 0100 GBP-PIN/GBP-PINO
- 0101 NL-PIN-1

**PIN Generation (PINGEN)**
- 000P
- 00100010

**PIN Verify (PINVER)**
- 000P
- 00100010

**IPIN Encode (IPINENC)**
- 00000000
- 00100001

**OPIN Encode (OPINENC)**
- 00000000
- 00100100

**Cryptographic Variable-Encrypting Keys**

| 0 0 0 0 0 | 0 1 1 1 1 | 0 0 0 0 0 | 0 0 0 0 1 | 0 0 0 0 0 | f f f 0 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 |

Key-form

---

*Figure 7. Control vector base bit map (PIN processing keys and cryptographic variable-encrypting keys)*
Key Form Bits, 'fff'

The key form bits, 40-42, and for a double-length key, bits 104-106, are designated 'fff' in the preceding illustration.

These bits can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Single length key</td>
</tr>
<tr>
<td>010</td>
<td>Double length key, left half</td>
</tr>
<tr>
<td>001</td>
<td>Double length key, right half</td>
</tr>
</tbody>
</table>

The following values could exist in some CCA implementations:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>Double-length key, left half, halves guaranteed unique</td>
</tr>
<tr>
<td>101</td>
<td>Double-length key, right half, halves guaranteed unique</td>
</tr>
</tbody>
</table>

Specifying a control-vector-base value

You can determine the value of a control vector by working through a series of questions.

About this task

Work through this series of questions:

Procedure

1. Begin with a field of 64 bits (eight bytes) set to B'0'. The most significant bit is referred to as bit 0. Define the key type and subtype (bits 8 - 14) as follows:
• The main key type bits (bits 8 - 11). Set bits 8 - 11 to one of the following values:

Table 197. Main key type bits

<table>
<thead>
<tr>
<th>Bits 8 - 11</th>
<th>Main Key Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Data operation keys</td>
</tr>
<tr>
<td>0010</td>
<td>PIN keys</td>
</tr>
<tr>
<td>0011</td>
<td>Cryptographic variable-encrypting keys</td>
</tr>
<tr>
<td>0100</td>
<td>Key-encrypting keys</td>
</tr>
<tr>
<td>0101</td>
<td>Key-generating keys</td>
</tr>
<tr>
<td>0111</td>
<td>Diversified key-generating keys</td>
</tr>
</tbody>
</table>

• The key subtype bits (bits 12 - 14). Set bits 12 - 14 to one of the following values:

Note: For Diversified Key Generating Keys, the subtype field specifies the hierarchical level of the DKYGENKY. If the subtype is nonzero, the DKYGENKY can generate only another DKYGENKY key with the hierarchy level decremented by one. If the subtype is zero, the DKYGENKY can generate only the final diversified key (a non-DKYGENKY key) with the key type specified by the usage bits.

Table 198. Key subtype bits

<table>
<thead>
<tr>
<th>Bits 12 - 14</th>
<th>Key Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Operation Keys</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>Compatibility key (DATA)</td>
</tr>
<tr>
<td>001</td>
<td>Confidentiality key (CIPHER, DECIPHER, or ENCIPHER)</td>
</tr>
<tr>
<td>010</td>
<td>MAC key (MAC or MACVER)</td>
</tr>
<tr>
<td>101</td>
<td>Secure messaging keys</td>
</tr>
<tr>
<td>Key-Encrypting Keys</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>Transport-sending keys (EXPORTER and OKEYXLAT)</td>
</tr>
<tr>
<td>001</td>
<td>Transport-receiving keys (IMPORTER and IKEYXLAT)</td>
</tr>
<tr>
<td>PIN Keys</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>PIN-generating key (PINGEN, PINVER)</td>
</tr>
<tr>
<td>000</td>
<td>Inbound PIN-block decrypting key (IPINENC)</td>
</tr>
<tr>
<td>010</td>
<td>Outbound PIN-block encrypting key (OPINENC)</td>
</tr>
<tr>
<td>Cryptographic Variable-Encrypting Keys</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>Cryptographic variable-encrypting key (CVAR....)</td>
</tr>
<tr>
<td>Diversified Key Generating Keys</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>DKY Subtype 0</td>
</tr>
<tr>
<td>001</td>
<td>DKY Subtype 1</td>
</tr>
<tr>
<td>010</td>
<td>DKY Subtype 2</td>
</tr>
<tr>
<td>011</td>
<td>DKY Subtype 3</td>
</tr>
<tr>
<td>100</td>
<td>DKY Subtype 4</td>
</tr>
<tr>
<td>101</td>
<td>DKY Subtype 5</td>
</tr>
<tr>
<td>110</td>
<td>DKY Subtype 6</td>
</tr>
</tbody>
</table>
Table 198. Key subtype bits (continued)

<table>
<thead>
<tr>
<th>Bits 12 - 14</th>
<th>Key Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>DKY Subtype 7</td>
</tr>
</tbody>
</table>

2. For key-encrypting keys, set the following bits:
   - The key-generating usage bits (gks, bits 18 - 20). Set the gks bits to B’111’ to indicate the Key Generate verb can use the associated key-encrypting key to encipher generated keys when the Key Generate verb is generating various key-pair key-form combinations (see the Key-Encrypting Keys section of Figure 5 on page 670). Without any of the gks bits set to B’1’, the Key Generate verb cannot use the associated key-encrypting key. The Key Token Build verb can set the gks bits to B’1’ when you supply the OPIM, IMEX, IMIM, OPEX, and EXEX keywords.
   - The IMPORT and EXPORT bit and the XLATE bit (ix, bits 21 and 22). If the ‘i’ bit is set to B’1’, the associated key-encrypting key can be used in the Data Key Import, Key Import, Data Key Export, and Key Export verbs. If the ‘x’ bit is set to B’1’, the associated key-encrypting key can be used in the Key Translate and Key Translate2 verbs.
   - The key-form bits (fff, bits 40 - 42). The key-form bits indicate how the key was generated and how the control vector participates in multiple-enciphering. To indicate the parts can be the same value, set these bits to B’010’. For information about the value of the key-form bits in the right half of a control vector, see Step 8 on page 676.

3. For MAC and MACVER keys, set the following bits:
   - The MAC control bits (bits 20 and 21). For a MAC-generate key, set bits 20 and 21 to B’11’. For a MAC-verify key, set bits 20 and 21 to B’01’.
   - The key-form bits (fff, bits 40 - 42). For a single-length key, set the bits to B’000’. For a double-length key, set the bits to B’010’.

4. For PINGEN and PINVER keys, set the following bits:
   - The PIN calculation method bits (aaaa, bits 0 - 3). Set these bits to one of the following values:

Table 199. Calculation method keyword bits

<table>
<thead>
<tr>
<th>Bits 0 - 3</th>
<th>Calculation Method Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NO-SPEC</td>
<td>A key with this control vector can be used with any PIN calculation method.</td>
</tr>
<tr>
<td>0001</td>
<td>IBM-PIN or IBM-PINO</td>
<td>A key with this control vector can be used only with the IBM PIN or PIN Offset calculation method.</td>
</tr>
<tr>
<td>0010</td>
<td>VISA-PVV</td>
<td>A key with this control vector can be used only with the VISA-PVV calculation method.</td>
</tr>
<tr>
<td>0100</td>
<td>GBP-PIN or GBP-PINO</td>
<td>A key with this control vector can be used only with the German Banking Pool PIN or PIN Offset calculation method.</td>
</tr>
<tr>
<td>0011</td>
<td>INBK-PIN</td>
<td>A key with this control vector can be used only with the Interbank PIN calculation method.</td>
</tr>
</tbody>
</table>
The prohibit-offset bit (o, bit 37) to restrict operations to the PIN value. If set to B'1', this bit prevents operation with the IBM 3624 PIN Offset calculation method and the IBM German Bank Pool PIN Offset calculation method.

5. For PINGEN, IPINENC, and OPINENC keys, set bits 18 - 22 to indicate whether the key can be used with the following verbs:

<table>
<thead>
<tr>
<th>Service Allowed</th>
<th>Bit Name</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear PIN Generate</td>
<td>CPINGEN</td>
<td>18</td>
</tr>
<tr>
<td>Encrypted PIN Generate Alternate</td>
<td>EPINGENA**</td>
<td>19</td>
</tr>
<tr>
<td>Encrypted PIN Generate</td>
<td>EPINGEN</td>
<td>20 for PINGEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19 for OPINENC</td>
</tr>
<tr>
<td>Clear PIN Generate Alternate</td>
<td>CPINGENA</td>
<td>21 for PINGEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20 for OPINENC</td>
</tr>
<tr>
<td>Encrypted PIN Verify</td>
<td>EPINVER</td>
<td>19</td>
</tr>
<tr>
<td>Clear PIN Encrypt</td>
<td>CPINENC</td>
<td>18</td>
</tr>
</tbody>
</table>

** EPINGENA is no longer supported, although the bit retains this definition for compatibility. There is no Encrypted Pin Generate Alternate verb.

6. For the IPINENC (inbound) and OPINENC (outbound) PIN-block ciphering keys, do the following:
   - Set the TRANSLAT bit (t, bit 21) to B'1' to permit the key to be used in the PIN Translate verb. The Control Vector Generate verb can set the TRANSLAT bit to B'1' when you supply the TRANSLAT keyword.
   - Set the REFORMAT bit (r, bit 22) to B'1' to permit the key to be used in the PIN Translate verb. The Control Vector Generate verb can set the REFORMAT bit and the TRANSLAT bit to B'1' when you supply the REFORMAT keyword.

7. For the cryptographic variable-encrypting keys (bits 18 - 22), set the variable-type bits (bits 18 - 22) to one of the following values:

<table>
<thead>
<tr>
<th>Bits 18 - 22</th>
<th>Generic Key Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>CVARPINE</td>
<td>Used in the Encrypted PIN Generate Alternate verb to encrypt a clear PIN.</td>
</tr>
<tr>
<td>00010</td>
<td>CVARXCVL</td>
<td>Used in the Control Vector Translate verb to decrypt the left mask array.</td>
</tr>
<tr>
<td>00011</td>
<td>CVARXCVR</td>
<td>Used in the Control Vector Translate verb to decrypt the right mask array.</td>
</tr>
</tbody>
</table>

8. For key-generating keys, set the following bits:
   - For KEYGENKY, set bit 18 for UKPT usage and bit 19 for CLR8-ENC usage.
   - For DKYGENKY, bits 12–14 will specify the hierarchical level of the DKYGENKY key. If the subtype CV bits are nonzero, the DKYGENKY can generate only another DKYGENKY key with the hierarchical level
decremented by one. If the subtype CV bits are zero, the DKYGENKY can generate only the final diversified key (a non-DKYGENKY key) with the key type specified by usage bits.

To specify the subtype values of the DKYGENKY, keywords DKYL0, DKYL1, DKYL2, DKYL3, DKYL4, DKYL5, DKYL6, and DKYL7 will be used.

- For DKYGENKY, bit 18 is reserved and must be zero.
- Usage bits 18-22 for the DKYGENKY key type are defined as follows. They will be encoded as the final key type that the DKYGENKY key generates.

<table>
<thead>
<tr>
<th>Bits 19 - 22</th>
<th>Keyword</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>DDATA</td>
<td>DATA, DATAC, single or double length</td>
</tr>
<tr>
<td>0010</td>
<td>DMAC</td>
<td>MAC, DATAM</td>
</tr>
<tr>
<td>0011</td>
<td>DMV</td>
<td>MACVER, DATAMV</td>
</tr>
<tr>
<td>0100</td>
<td>DIMP</td>
<td>IMPORTER, IKEYXLAT</td>
</tr>
<tr>
<td>0101</td>
<td>DEXP</td>
<td>EXPORTER, OKEYXLAT</td>
</tr>
<tr>
<td>0110</td>
<td>DPVR</td>
<td>PINVER</td>
</tr>
<tr>
<td>1000</td>
<td>DMKEY</td>
<td>Secure message key for encrypting keys</td>
</tr>
<tr>
<td>1001</td>
<td>DMPIN</td>
<td>Secure message key for encrypting PINs</td>
</tr>
<tr>
<td>1111</td>
<td>DALL</td>
<td>All key types can be generated except DKYGENKY and KEYGENKY keys. Usage of the DALL keyword is controlled by a separate access control point.</td>
</tr>
</tbody>
</table>

9. For secure messaging keys, set the following bits:
   - Set bit 18 to B'1' if the key will be used in the secure messaging for PINs service. Set bit 19 to B'1' if the key will be used in the secure messaging for keys service.

10. For all keys, set the following bits:
   - The export bit (E, bit 17). If set to B'0', the export bit prevents a key from being exported. By setting this bit to B'0', you can prevent the receiver of a key from exporting or translating the key for use in another cryptographic subsystem. After this bit is set to B'0', it cannot be set to B'1' by any service other than Control Vector Translate. The Prohibit Export verb can reset the export bit.
   - The key-part bit (K, bit 44). Set the key-part bit to B'1' in a control vector associated with a key part. When the final key part is combined with previously accumulated key parts, the key-part bit in the control vector for the final key part is set to B'0'. The Control Vector Generate verb can set the key-part bit to B'1' when you supply the KEY-PART keyword.
   - The anti-variant bits (bit 30 and bit 38). Set bit 30 to B'0' and bit 38 to B'1'. Many cryptographic systems have implemented a system of variants where a 7-bit value is XORed with each 7-bit group of a key-encrypting key before enciphering the target key. By setting bits 30 and 38 to opposite values, control vectors do not produce patterns that can occur in variant-based systems.
Changing control vectors with the Control Vector Translate verb

What you need to do when you use the Control Vector Translate verb.

About this task

Do the following when using the verb:

- Provide the control information for testing the control vectors of the source, target, and key-encrypting keys to ensure that only sanctioned changes can be performed
- Select the key-half processing mode.

Providing the control information for testing the control vectors

To minimize your security exposure, the Control Vector Translate verb requires control information (mask array information) to limit the range of allowable control vector changes.

To ensure that this verb is used only for authorized purposes, the source-key control vector, target-key control vector, and key-encrypting key (KEK) control vector must pass specific tests. The tests on the control vectors are performed within the secured cryptographic engine.

The tests consist of evaluating four logic expressions, the results of which must be a string of binary zeros. The expressions operate bitwise on information that is contained in the mask arrays and in the portions of the control vectors associated with the key or key-half that is being processed. If any of the expression evaluations do not result in all zero bits, the verb is ended with a control vector violation return and reason code (8/39). See Figure 9 on page 680. Only the 56-bit positions that are associated with a key value are evaluated. The low-order bit that is associated with key parity in each key byte is not evaluated.

Mask array preparation

A mask array consists of seven 8-byte elements: A1, B1, A2, B2, A3, B3, and B4.

You choose the values of the array elements such that each of the following four expressions evaluates to a string of binary zeros. (See Figure 9 on page 680.) Set the A bits to the value you require for the corresponding control vector bits. In expressions 1 on page 679 through 3 on page 679, set the B bits to select the control vector bits to be evaluated. In expression 4 on page 679, set the B bits to select the source and target control vector bits to be evaluated. Also, use the following control vector information:

- C1 is the control vector associated with the left half of the KEK.

- Control vector bits 64 - 127. If bits 40 - 42 are B'000' (single-length key), set bits 64 - 127 to B'0'. Otherwise, copy bits 0 - 63 into bits 64 - 127 and set bits 105 and 106 to B'01'.

- Set the parity bits (low-order bit of each byte, bits 7, 15, ..., 127). These bits contain the parity bits (P) of the control vector. Set the parity bit of each byte so the number of zero-value bits in the byte is an even number.

- For secure messaging keys, usage bit 18 on will enable the encryption of keys in a secure message and usage bit 19 on will enable the encryption of PINs in a secure message.
C2 is the control vector associated with the source key or selected source-key half/halves.
C3 is the control vector associated with the target key or selected target-key half/halves.

1. \((C_1 \text{ XOR } A_1) \text{ logical-AND } B_1\)
   This expression tests whether the KEK used to encipher the key meets your criteria for the desired translation.

2. \((C_2 \text{ XOR } A_2) \text{ logical-AND } B_2\)
   This expression tests whether the control vector associated with the source key meets your criteria for the desired translation.

3. \((C_3 \text{ XOR } A_3) \text{ logical-AND } B_3\)
   This expression tests whether the control vector associated with the target key meets your criteria for the desired translation.

4. \((C_2 \text{ XOR } C_3) \text{ logical-AND } B_4\)
   This expression tests whether the control vectors associated with the source key and the target key meet your criteria for the desired translation.

Encipher two copies of the mask array, each under a different cryptographic-variable key (key type CVARENC). Use two different keys so the enciphered-array copies are unique values. When using the Control Vector Translate verb, the mask_array_left parameter and the mask_array_right parameter identify the enciphered mask arrays. The array_key_left parameter and the array_key_right parameter identify the internal keys for deciphering the mask arrays. The array_key_left parameter must have a key type of CVARXCVL and the array_key_right parameter must have a key type of CVARXCVR. The cryptographic process deciphers the arrays and compares the results; for the service to continue, the deciphered arrays must be equal. If the results are not equal, the service returns the return and reason code for data that is not valid (8/385).

Use the Key Generate verb to create the key pairs CVARENC-CVARXCVL and CVARENC-CVARXCVR. Each key in the key pair must be generated for a different node. The CVARENC keys are generated for, or imported into, the node where the mask array will be enciphered. After enciphering the mask array, you should destroy the enciphering key. The CVARXCVL and CVARXCVR keys are generated for, or imported into, the node where the Control Vector Translate verb will be performed.

If using the BOTH keyword to process both halves of a double-length key, remember that bits 41, 42, 104, and 105 are different in the left and right halves of the CCA control vector and must be ignored in your mask-array tests (that is, make the corresponding B2 and/or B3 bits equal to zero).

When the control vectors pass the masking tests, the verb does the following:
• Deciphers the source key. In the decipher process, the service uses a key that is formed by the XOR of the KEK and the control vector in the key token variable the source_key_token parameter identifies.
• Enciphers the deciphered source key. In the encipher process, the verb uses a key that is formed by the XOR of the KEK and the control vector in the key token variable the target_key_token parameter identifies.
• Places the enciphered key in the key field in the key token variable the target_key_token parameter identifies.
Selecting the key-half processing mode

Use the Control Vector Translate verb to change a control vector associated with a key.

*rule_array* keywords determine which key halves are processed in the call, as shown in Figure 10 on page 681.
Keyword  
Description

**SINGLE**  
This keyword causes the control vector of the left half of the source key to be changed. The updated key half is placed into the left half of the target key in the target key token. The right half of the target key is unchanged.

The **SINGLE** keyword is useful when processing a single-length key or when first processing the left half of a double-length key (to be followed by processing the right half).

**RIGHT**  
This keyword causes the control vector of the right half of the source key to be changed. The updated key half is placed into the right half of the target key of the target key token. The left half of the source key is copied unchanged into the left half of the target key in the target key token.

**BOTH**  
This keyword causes the control vector of both halves of the source key to be changed. The updated key is placed into the target key in the target key token.

A single set of control information must permit the control vector changes applied to each key half. Normally, control vector bit positions 41, 42, 105, and 106 are different for each key half. Therefore, set bits 41 and 42 to B'00' in mask array elements B₁, B₂, and B₃.

You can verify that the source and target key tokens have control vectors with matching bits in bit positions 40-42 and 104-106, the “form field” bits. Ensure bits 40-42 of mask array B₄ are set to B'111'.

**LEFT**  
This keyword enables you to supply a single-length key and obtain a double-length key. The source key token must contain:

- The KEK-enciphered single-length key
- The control vector for the single-length key (often this is a null value)
- A control vector, stored in the source token where the right-half control vector is normally stored, used in decrypting the single-length source key when the key is being processed for the target right half of the key.

The verb first processes the source and target tokens as with the **SINGLE** keyword. Then the source token is processed using the single-length enciphered key and the source token right-half control vector to obtain the actual key value. The key value is then enciphered using the KEK and the control vector in the target token for the right-half of the key.
This approach is frequently of use when you must obtain a double-length CCA key from a system that supports only a single-length key, for example when processing PIN keys or key-encrypting keys received from non-CCA systems.

To prevent the verb from ensuring each key byte has odd parity, you can specify the NOADJUST keyword. If you do not specify the NOADJUST keyword, or if you specify the ADJUST keyword, the verb ensures each byte of the target key has odd parity.

**When the target key-token CV is null**

When you use any of the LEFT, BOTH, or RIGHT keywords, and when the control vector in the target key token is null (all B'0'), bit 3 in byte 59 will be set to B'1' to indicate this is a double-length DATA key.

**Control vector translate example**

As an example, consider the case of receiving a single-length PIN-block encrypting key from a non-CCA system.

Often such a key will be encrypted by an unmodified transport key (no control vector or variant is used). In a CCA system, an inbound PIN encrypting key is double-length.

First use the Key Token Build verb to insert the single-length key value into the left-half key-space in a key token. Specify USE-CV as a key type and a control vector value set to 16 bytes of X'00'. Also specify EXTERNAL, KEY, and CV keywords in the rule_array. This key token will be the source key key-token.

Second, the target key token can also be created using the Key Token Build verb. Specify a key type of IPINENC and the NO-EXPORT rule_array keyword.

Then call the Control Vector Translate verb and specify a rule_array keyword of LEFT. The mask arrays can be constructed as follows:

- A1 is set to the value of the KEK's control vector, most likely the value of an IMPORTER key, perhaps with the NO-EXPORT bit set. B1 is set to eight bytes of X'FF' so all bits of the KEK's control vector will be tested.
- A2 is set to eight bytes of X'00', the (null) value of the source key control vector. B2 is set to eight bytes of X'FF' so all bits of the source-key “control vector” will be tested.
- A3 is set to the value of the target key's left-half control vector. B3 is set to X'FFFF FFFF FF9F FFFF'. This will cause all bits of the control vector to be tested except for the two (“fff”) bits used to distinguish between the left-half and right-half target-key control vector.
- B4 is set to eight bytes of X'00' so no comparison is made between the source and target control vectors.
Appendix E. PIN formats and algorithms

This appendix describes the personal identification number (PIN) notation, PIN block formats, PIN extraction rules, and PIN algorithms.

For PIN calculation procedures, see IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference.

PIN notation

This section describes various PIN block formats.

The following notations describe the contents of PIN blocks:

- **P** = A 4-bit decimal digit that is one digit of the PIN value.
- **C** = A 4-bit hexadecimal control value. The valid values are X'0', X'1', and X'2'.
- **L** = A 4-bit hexadecimal value that specifies the number of PIN digits. This value ranges from 4 - 12, inclusive.
- **F** = A 4-bit field delimiter of value X'F'.
- **f** = A 4-bit delimiter filler that is either P or F, depending on the length of the PIN.
- **D** = A 4-bit decimal padding value. All pad digits in the PIN block have the same value.
- **X** = A 4-bit hexadecimal padding value. All pad digits in the PIN block have the same value.
- **x** = A 4-bit hexadecimal filler that is either P or X, depending on the length of the PIN.
- **R** = A 4-bit hexadecimal random digit. The sequence of R digits can each take a different value.
- **r** = A 4-bit random filler that is either P or R, depending on the length of the PIN.
- **Z** = A 4-bit hexadecimal zero (X'0').
- **z** = A 4-bit zero filler that is either P or Z, depending on the length of the PIN.
- **S** = A 4-bit hexadecimal digit that constitutes one digit of a sequence number.
- **A** = A 4-bit decimal digit that constitutes one digit of a user-specified constant.

PIN block formats

This section describes the PIN block formats and assigns a code to each format.

**ANSI X9.8**

This format is also named ISO format 0, VISA format 1, VISA format 4, and ECI format 1.

- **P1** = CLPPPPppppppppppFFFF
- **P2** = ZZZZaaaaaaaaaaaa
PIN Block = P1 XOR P2
where C = X'0'
   L = X'4' to X'C'

Programming Note: The rightmost 12 digits (excluding the check digit) in P2 are
the rightmost 12 digits of the account number for all formats except VISA format 4.
For VISA format 4, the rightmost 12 digits (excluding the check digit) in P2 are the
leftmost 12 digits of the account number.

ISO Format 1
Example code for ISO Format 2.
This format is also named ECI format 4.
   PIN Block = C L P P P P P P P P P P P
where C = X'1'
   L = X'4' to X'C'

ISO Format 2
Example code for ISO Format 2.
   PIN Block = C L P P P P P P P P P P P P
where C = X'2'
   L = X'4' to X'C'

ISO Format 3
The formats of the intermediate PIN-block, the PAN block, and the ISO-3
PIN-block.

An ISO-3 PIN-block format is equivalent to the ANSI X9.8, VISA-1, and ECI-1
PIN-block formats in length. A PIN that is longer than 12 digits is truncated on the
right.

```
<table>
<thead>
<tr>
<th>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 L P P P P P P P P P P P P P P P R R</td>
</tr>
</tbody>
</table>
```

Intermediate PIN-Block = IPB

```
| 0 0 0 0 PAN PAN PAN PAN PAN PAN PAN PAN |
```
PAN Block

```
<table>
<thead>
<tr>
<th>3 L P P P P P P P P P P P P P X X X X X X</th>
</tr>
</thead>
</table>
```
PIN Block = IPB XOR PAN Block

Figure 11. ISO-3 PIN-block format
3 Is the value X'3' for ISO-3.
L Is the length of the PIN, which is a 4-bit value from X'4' - X'C'.
P Is a PIN digit, which is a 4-bit value from X'0' - X'9'. The values of the PIN digits are independent.
P/R Is a PIN digit or pad value. A PIN digit has a 4-bit value from X'0' - X'9'. A pad value has a random 4-bit value of X'A' - X'F'. The number of pad values in the intermediate PIN block (IPB) is from 2 - 10.
R Is the random value X'A' - X'F' for the pad value.
PAN Is twelve 4-bit digits that represent one of the following:
  • The rightmost 12 digits of the primary account-number (excluding the check digit) if the format of the PIN block is ISO-3, ANSI X9.8, VISA-1, or ECI-1.
  • The leftmost 12 digits of the primary account-number (excluding the check digit) if the format of the PIN block is VISA-4.
  Each PAN digit has a value from X'0' - X'9'.

The PIN block is the result of XORing the 64-bit IPB with the 64-bit PAN block.

Example:
L = 6, PIN = 123456, Personal Account Number = 11122233444555
36123456AFBECCDC : IPB
0000222333444555 : PAN block for ISO-3 (ANSI X9.8, VISA-1, ECI-1) format
361216759CFA8889 : PIN block for ISO-3 (ANSI X9.8, VISA-1, ECI-1) format

VISA Format 2
Example code for VISA Format 2.
PIN Block = LPPPPzzDDDDDDDDD

where L = X'4' to X'6'

VISA Format 3
Example code for VISA Format 3.
This format specifies that the PIN length can be 4 - 12 digits, inclusive. The PIN starts from the leftmost digit and ends by the delimiter ('F'), and the remaining digits are padding digits.

An example of a 6-digit PIN:
PIN Block = PPPPPPFXxxxxxxxx

IBM 3624 Format
This format requires the program to specify the delimiter, X, for determining the PIN length.
PIN Block = PPPPxxxxxxxx

IBM 3621 Format
This format requires the program to specify the delimiter, X, for determining the PIN length.
PIN Block = SSSSSPPPxxxxxxxx
ECI Format 2
This format defines the PIN to be 4 digits.

PIN Block = PPPPRRRRRRRRRRRR

ECI Format 3
Example code for ECI Format 3.

PIN Block = LPPPPzzRRRRRRRRRR

where L = X'4' to X'6'

PIN extraction rules
This section describes the PIN extraction rules for the Encrypted PIN Verify and Encrypted PIN Translate verbs.

Encrypted PIN Verify verb
This verb extracts the customer-entered PIN from the input PIN block.

It extracts the PIN according to the following rules:

- If the input PIN block format is ANSI X9.8, ISO format 0, VISA format 1, VISA format 4, ECI format 1, ISO format 2, IS0 format 3, VISA format 2, IBM Encrypting PINPAD format, or ECI format 3, the verb extracts the PIN according to the length specified in the PIN block.
- If the input PIN block format is VISA format 3, the specified delimiter (padding) determines the PIN length. The search starts at the leftmost digit in the PIN block. If the input PIN block format is 3624, the specification of a PIN extraction method for the 3624 is supported through rule_array keywords. If no PIN extraction method is specified in the rule_array, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is 3621, the specification of a PIN extraction method for the 3621 is supported through rule_array keywords. If no PIN extraction method is specified in the rule_array, the specified delimiter (padding) determines the PIN length.
- If the input PIN block format is ECI format 2, the PIN is the leftmost 4 digits.

For the VISA algorithm, if the extracted PIN length is less than 4, the verb sets a reason code that indicates verification failed. If the length is greater than or equal to 4, the verb uses the leftmost 4 digits as the referenced PIN.

For the IBM German Banking Pool algorithm, if the extracted PIN length is not 4, the verb sets a reason code that indicates verification failed.

For the IBM 3624 algorithm, if the extracted PIN length is less than the PIN check length, the verb sets a reason code that indicates verification failed.

Clear PIN Generate Alternate verb
This verb extracts the customer-entered PIN from the input PIN block.

It extracts the PIN from the input PIN block according to the following rules:

- This verb supports the specification of a PIN extraction method for the 3624 and 3621 PIN block formats through the use of the rule_array keyword. The rule_array points to an array of one or two 8-byte elements. The first element in the rule_array specifies the PIN calculation method. The second element in the
Encrypted PIN Translate verb

This verb extracts the customer-entered PIN from the input PIN block.

It extracts the PIN from the input PIN block according to the following rules:

- If the input PIN block format is ANSI X9.8, ISO format 0, VISA format 1, VISA format 4, ECI format 1, ISO format 1, ISO format 2, ISO format 3, VISA format 2, IBM Encrypting PINPAD format, or ECI format 3 and, if the specified PIN length is less than 4, the verb sets a reason code to reject the operation. If the specified PIN length is greater than 12, the operation proceeds to normal completion with unpredictable contents in the output PIN block. Otherwise, the verb extracts the PIN according to the specified length.

- If the input PIN block format is VISA format 3, the specified delimiter (padding) determines the PIN length. The search starts at the leftmost digit in the PIN block. If the input PIN block format is 3624, the specification of a PIN extraction method for the 3624 is supported through rule_array keywords. If no PIN extraction method is specified in the rule_array, the specified delimiter (padding) determines the PIN length.

- If the input PIN block format is 3621, the specification of a PIN extraction method for the 3621 is supported through rule_array keywords. If no PIN extraction method is specified in the rule_array, the specified delimiter (padding) determines the PIN length.

- If the input block format is ECI format 2, the PIN is always the leftmost 4 digits.

If the maximum PIN length allowed by the output PIN block is shorter than the extracted PIN, only the leftmost digits of the extracted PIN that form the allowable maximum length are placed in the output PIN block. The PIN length field in the output PIN block, if it exists, specifies the allowable maximum length.
value of the digit of the enciphered validation data. The result is an intermediate PIN. The leftmost n digits of the intermediate PIN are the generated PIN, where n is specified by the assigned PIN length.

Figure 12 illustrates the 3624 PIN generation algorithm.

**German Banking Pool PIN Generation algorithm**

This algorithm generates a 4-digit PIN based on account-related data or person-related data, namely the validation data.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN-generation key

The validation data is enciphered using the PIN generation key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data. The result is an intermediate PIN. The rightmost 4 digits of the leftmost 6 digits of the intermediate PIN are extracted. The leftmost digit of the extracted 4 digits is checked for zero. If the digit is zero, the digit is changed to one; otherwise, the digit remains unchanged. The resulting four digits is the generated PIN.

Figure 13 on page 689 illustrates the German Banking Pool (GBP) PIN generation algorithm.
PIN Offset Generation algorithm

To allow the customer to select his own PIN, a PIN offset is used by the IBM 3624 and GBP PIN generation algorithms to relate the customer-selected PIN to the generated PIN.

The PIN offset generation algorithm requires two parameters in addition to those used in the 3624 PIN generation algorithm. They are a customer-selected PIN and a 4-bit PIN check length. The length of the customer-selected PIN is equal to the assigned-PIN length, n.

The 3624 PIN generation algorithm described in the previous section is performed. The offset data value is the result of subtracting (modulo 10) the leftmost n digits of the intermediate PIN from the customer-selected PIN. The modulo 10 subtraction ignores borrows. The rightmost m digits of the offset data form the PIN offset, where m is specified by the PIN check length. Note that n cannot be less than m. To generate a PIN offset for a GBP PIN, m is set to 4 and n is set to 6.

Figure 14 on page 690 illustrates the PIN offset generation algorithm.

Figure 13. GBP PIN generation algorithm

PIN Offset Generation algorithm

To allow the customer to select his own PIN, a PIN offset is used by the IBM 3624 and GBP PIN generation algorithms to relate the customer-selected PIN to the generated PIN.

The PIN offset generation algorithm requires two parameters in addition to those used in the 3624 PIN generation algorithm. They are a customer-selected PIN and a 4-bit PIN check length. The length of the customer-selected PIN is equal to the assigned-PIN length, n.

The 3624 PIN generation algorithm described in the previous section is performed. The offset data value is the result of subtracting (modulo 10) the leftmost n digits of the intermediate PIN from the customer-selected PIN. The modulo 10 subtraction ignores borrows. The rightmost m digits of the offset data form the PIN offset, where m is specified by the PIN check length. Note that n cannot be less than m. To generate a PIN offset for a GBP PIN, m is set to 4 and n is set to 6.

Figure 14 on page 690 illustrates the PIN offset generation algorithm.
3624 PIN Verification algorithm

This algorithm generates an intermediate PIN based on the specified validation data. A part of the intermediate PIN is adjusted by adding an offset data. A part of the result is compared with the corresponding part of the customer-entered PIN.

The algorithm requires the following input parameters:
- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN-verification key
- A 4-bit PIN-check length
- An offset data
- A customer-entered PIN

The rightmost m digits of the offset data form the PIN offset, where m is the PIN check length.

1. The validation data is enciphered using the PIN verification key. Each digit of the enciphered validation data is replaced by the digit in the decimalization
table whose displacement from the leftmost digit of the table is the same as the
value of the digit of enciphered validation data.

2. The leftmost \( n \) digits of the result is added (modulo 10) to the offset data value,
where \( n \) is the length of the customer-entered PIN. The modulo 10 addition
ignores carries.

3. The rightmost \( m \) digits of the result of the addition operation form the PIN
check number. The PIN check number is compared with the rightmost \( m \) digits
of the customer-entered PIN. If they match, PIN verification is successful;
otherwise, verification is unsuccessful.

When a nonzero PIN offset is used, the length of the customer-entered PIN is
equal to the assigned PIN length.

**Figure 15** illustrates the PIN verification algorithm.

**German Banking Pool PIN Verification algorithm**

This algorithm generates an intermediate PIN based on the specified validation
data.
A part of the intermediate PIN is adjusted by adding an offset data. A part of the result is extracted. The extracted value might or might not be modified before it compares with the customer-entered PIN.

The algorithm requires the following input parameters:
- A 64-bit validation data
- A 64-bit decimalization table
- A 128-bit PIN verification key
- An offset data
- A customer-entered PIN

The rightmost 4 digits of the offset data form the PIN offset.
1. The validation data is enciphered using the PIN verification key. Each digit of the enciphered validation data is replaced by the digit in the decimalization table whose displacement from the leftmost digit of the table is the same as the value of the digit of enciphered validation data.
2. The leftmost 6 digits of the result is added (modulo 10) to the offset data. The modulo 10 addition ignores carries.
3. The rightmost 4 digits of the result of the addition (modulo 10) are extracted.
4. The leftmost digit of the extracted value is checked for zero. If the digit is zero, the digit is set to one; otherwise, the digit remains unchanged. The resulting four digits are compared with the customer-entered PIN. If they match, PIN verification is successful; otherwise, verification is unsuccessful.

Figure 16 on page 693 illustrates the GBP PIN verification algorithm.
VISA PIN algorithms

The VISA PIN verification algorithm performs a multiple encipherment of a value, called the transformed security parameter (TSP), and a extraction of a 4-digit PIN verification value (PVV) from the ciphertext.

The calculated PVV is compared with the referenced PVV and stored on the plastic card or data base. If they match, verification is successful.

**PVV Generation algorithm**

The algorithm generates a 4-digit PIN verification value (PVV) based on the transformed security parameter (TSP).

The algorithm requires the following input parameters:

- A 64-bit TSP
- A 128-bit PVV generation key
1. A multiple encipherment of the TSP using the double-length PVV generation key is performed.
2. The ciphertext is scanned from left to right. Decimal digits are selected during the scan until four decimal digits are found. Each selected digit is placed from left to right according to the order of selection. If four decimal digits are found, those digits are the PVV.
3. If, at the end of the first scan, less than four decimal digits have been selected, a second scan is performed from left to right. During the second scan, all decimal digits are skipped and only non-decimal digits can be processed. Non-decimal digits are converted to decimal digits by subtracting 10. The process proceeds until four digits of PVV are found.

Figure 17 illustrates the PVV generation algorithm.

**Programming Note:** For VISA PVV algorithms, the leftmost 11 digits of the TSP are the personal account number (PAN), the leftmost 12th digit is a key table index to select the PVV generation key, and the rightmost 4 digits are the PIN. The key table index should have a value between 1 and 6, inclusive.

**PVV Verification algorithm**

The PVV verification algorithm requires specific parameters.

The algorithm requires the following input parameters:
- A 64-bit TSP
- A 16-bit referenced PVV
- A 128-bit PVV verification key
A PVV is generated using the PVV generation algorithm, except a PVV verification key rather than a PVV generation key is used. The generated PVV is compared with the referenced PVV. If they match, verification is successful.

**Interbank PIN Generation algorithm**

A description of the Interbank PIN calculation method.

The Interbank PIN calculation method consists of the following steps:

1. Let X denote the *transaction_security* parameter element converted to an array of 16 4-bit numeric values. This parameter consists of (in the following sequence) the 11 rightmost digits of the customer PAN (excluding the check digit), a constant of 6, a 1-digit key indicator, and a 3-digit validation field.
2. Encrypt X with the double-length PINGEN (or PINVER) key to get 16 hexadecimal digits (64 bits).
3. Perform decimalization on the result of the previous step by scanning the 16 hexadecimal digits from left to right, skipping any digit greater than X'9' until 4 decimal digits (for example, digits that have values from X'0' - X'9') are found. If all digits are scanned but 4 decimal digits are not found, repeat the scanning process, skipping all digits that are X'9' or less and selecting the digits that are greater than X'9'. Subtract 10 (X'A') from each digit selected in this scan. If the 4 digits that were found are all zeros, replace the 4 digits with 0100.
4. Concatenate and use the resulting digits for the Interbank PIN. The 4-digit PIN consists of the decimal digits in the sequence in which they are found.
Appendix F. Cryptographic algorithms and processes

This appendix provides processing details for specific aspects of the CCA design.

These processing details are described for these aspects:
- “Cryptographic key-verification techniques”
- “Modification Detection Code calculation” on page 700
- “Ciphering methods” on page 702
- “MAC calculation methods” on page 709
- “RSA key-pair generation” on page 711
- “Multiple decipherment and encipherment” on page 712
- “PKA92 key format and encryption process” on page 718
- “Formatting hashes and keys in public-key cryptography” on page 720

Cryptographic key-verification techniques

The key-verification implementations described in this document employ several mechanisms for assuring the integrity and value of the key.

Information is presented about these topics:
- “Master-key verification algorithms”
- “CCA DES-key verification algorithm” on page 699
- “Encrypt zeros AES-key verification algorithm” on page 699
- “Encrypt zeros DES-key verification algorithm” on page 700

Master-key verification algorithms

The CEX*C implementations employ triple-length DES and PKA master keys (three DES keys) that are internally represented in 24 bytes (168 bits).

Beginning with Release 3.30, the CEX2C implementation employs an AES master key represented in 32 bytes (256 bits). Beginning with Release 4.1.0, the CAA employs an APKA master key represented in 32 bytes (256 bits). Verification patterns on the contents of the new, current, and old master-key registers can be generated and verified when the selected register is not in the empty state. For the AES master key, the SHA-256 verification method is used.

The CEX*C employ several verification pattern generation methods.

SHA-1 based master-key verification method

A SHA-1 hash algorithm is calculated on the quantity X’01’ prepended to the 24-byte register contents.

The resulting 20-byte hash value is used in the following ways:
- The Key Test and Key Test2 verb2 uses the first eight bytes of the 20-byte hash value as the random_number variable, and uses the second eight bytes as the verification_pattern.
- A SHA-1 based master-key verification pattern stored in a two-byte or an eight-byte master-key verification pattern field in a key token consists of the first two or the first eight bytes of the calculated SHA-1 value, respectively.
**z/OS-based master-key verification method**

When the first and third portions of the symmetric master key have the same value, the master key is effectively a double-length DES key.

In this case, the master-key verification pattern (MKVP) is based on this algorithm:
- \( C = X'4545454545454545' \)
- \( IR = MK_{\text{first-part}} \text{ XOR } e_x(MK_{\text{first-part}}) \)
- \( MKVP = MK_{\text{second-part}} \text{ XOR } e_{IR}(MK_{\text{second-part}}) \)

where:
- \( e_x(Y) \) is the DES encoding of \( Y \) using \( x \) as a key
- XOR means bitwise exclusive OR

Version \( X'00' \) internal CCA DES key tokens use this eight-byte master-key verification pattern.

**SHA-256 based master-key verification method**

A SHA-256 hash algorithm is calculated on the quantity \( X'01' \) prepended to the 24-byte register contents.

For AES, there will be verification patterns for both the AES master key and for AES operational keys that are used to encipher or decipher data. The verification pattern on the master key is called the MKVP. The verification pattern on operational keys is referred to as a key-verification pattern (KVP).

Both the MKVP and KVP for AES will use the same algorithm. Both will be computed with the following process.
1. Compute the SHA-256 hash of the string formed by prepending the byte \( X'01' \) to the cleartext key value.
2. Take the leftmost eight bytes of the hash as the verification pattern.

This value is truncated to eight bytes because this is the length allocated for the verification in several CCA structures and APIs. For example, the AES key token has eight bytes for the MKVP, and the Key Test and Key Test2 verbs have an eight-byte parameter for the verification pattern.

**Asymmetric master key MDC-based verification method**

The verification pattern for the asymmetric master keys is based on hashing the value of the master-key using the MDC-4 hashing algorithm.

The master key is not parity adjusted.

The RSA private key sections \( X'06' \) and \( X'08' \) use this 16-byte master-key version number.

**Key-token verification patterns**

The verification pattern techniques used in the several types of CCA key tokens are described in this topic.

The techniques are:
- AES and ECC key tokens: leftmost 8 bytes of SHA-256 hash of the string formed by pre-pending \( X'01' \) to the cleartext key value.
- DES key tokens:
- Triple-length master key, key token version X'00': leftmost 8 bytes of SHA-1 hash
- Triple-length master key, key token version X'03': leftmost 2 bytes of SHA-1 hash
- Double-length master key, key token version X'00': leftmost 8 bytes of z/OS hash
- Double-length master key, key token version X'03': leftmost 2 bytes of SHA-1 hash

• RSA key tokens:
  - Private-key section types X'06' and X'08': 16-byte MDC-4 value
  - Private-key section types X'02' and X'05': leftmost 2 bytes of SHA-1 hash

• Trusted blocks: 16-byte MDC-4 value

CCA DES-key verification algorithm

The cryptographic engines provide a method for verifying the value of a DES cryptographic key or key part without revealing information about the value of the key or key part.

The CCA verification method first creates a random number. A one-way cryptographic function combines the random number with the key or key part. The verification method returns the result of this one-way cryptographic function (the verification pattern) and the random number.

Note: A one-way cryptographic function is a function in which it is easy to compute the output from a given input, but it is not computationally feasible to compute the input given an output.

For information about how you can use an application program to invoke this verification method, see “Key Test (CSNBKYT)” on page 195.

The CCA DES key verification algorithm does the following:
1. Sets $KKR' = KKR \text{ XOR } RN$
2. Sets $K1 = X'4545454545454545'$
3. Sets $XI = \text{DES encoding of } KKL \text{ using key } K1$
4. Sets $K2 = XI \text{ XOR } KKL$
5. Sets $X2 = \text{DES encoding of } KKR' \text{ using key } K2$
6. Sets $VP = X2 \text{ XOR } KKR'$

where:

$RN$ is the random number generated or provided

$KKL$ is the value of the single-length key, or is the left half of the double-length key

$KKR$ is XL8'00' if the key is a single-length key, or is the value of the right half of the double-length key

$VP$ is the verification pattern

Encrypt zeros AES-key verification algorithm

The cryptographic engine provides a method for verifying the value of an AES cryptographic key or key part without revealing information about the value of the key or key part.
In this method, the AES key data encryption algorithm encodes a 128-bit value that is all zero bits. The leftmost 32 bits of the result are compared to the trial input value or returned from the Key Test2 verb in an 8-byte variable that is padded with bits valued to zero.

**Encrypt zeros DES-key verification algorithm**

The cryptographic engine provides a method for verifying the value of a DES cryptographic key or key part without revealing information about the value of the key or key part.

In this method the single-length or double-length key DEA encodes a 64-bit value that is all zero bits. The leftmost 32 bits of the result are compared to the trial input value or returned from the Key Test and Key Test2 verbs.

For a single-length key, the key DEA encodes an 8-byte, all-zero-bits value.

For a double-length key, the key DEA triple-encodes an 8-byte, all-zero-bits value.

The left half (high-order half) key encodes the zero-bit value, this result is DEA decoded by the right key half, and that result is DEA encoded by the left key half.

**SHAVP1 algorithm**

This algorithm is used by the Key Test2 callable service to generate and verify the verification pattern.

\[ VP = \text{Trunc128} \left( \text{SHA256} ( KA \ |\ |\ KT \ |\ |\ KL \ |\ |\ K ) \right) \]

where:

- \( VP \) is the 128-bit verification pattern
- \( \text{TruncN}(x) \) is truncation of the string \( x \) to the left most \( N \) bits
- \( \text{SHA256}(x) \) is the SHA-256 hash of the string \( x \)
- \( KA \) is the one-byte CCA variable-length key token constant for the algorithm of key (HMAC X'03')
- \( KT \) is the two-byte CCA variable-length key token constant for the type of key (MAC X'0002')
- \( KL \) is the two-byte bit length of the clear key value
- \( K \) is the clear key value left-aligned and padded on the right with binary zeros to byte boundary
- | | is string concatenation

**Modification Detection Code calculation**

The Modification Detection Code (MDC) calculation method defines a one-way cryptographic function.

A one-way cryptographic function is a function in which it is easy to compute the input into output (a digest) but very difficult to compute the output into input. MDC uses DES encryption only and a default key of X'5252 5252 5252 5252 2525 2525 2525 2525'.

700  Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
The MDC Generate verb supports four versions of the MDC calculation method that you specify by using one of the keywords shown in Table 203. All versions use the MDC-1 calculation.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Version of the MDC calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDC-2, PADMDC-2</td>
<td>Specifies two encipherments for each 8-byte input data block. These versions use the MDC-2 calculation procedure described in Table 204.</td>
</tr>
<tr>
<td>MDC-4, PADMDC-4</td>
<td>Specifies four encipherments for each 8-byte input data block. These versions use the MDC-4 calculation procedure described in Table 204.</td>
</tr>
</tbody>
</table>

When the keywords PADMDC-2 and PADMDC-4 are used, the supplied text is always padded as follows:

- If the total supplied text is less than 16 bytes in length, pad bytes are appended to make the text length equal to 16 bytes. A length of zero is allowed.
- If the total supplied text is a minimum of 16 bytes in length, pad bytes are appended to make the text length equal to the next-higher multiple of eight bytes. One or more pad bytes are always added.
- All appended pad bytes, other than the last pad byte, are set to X'FF'.
- The last pad byte is set to a binary value equal to the count of all appended pad bytes (X'01' - X'10').

Use the resulting pad text in the Table 204. The MDC Generate verb uses these MDC calculation methods. See “MDC Generate (CSNBMDG)” on page 332 for more information.

### Table 204. MDC calculation procedures

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MDC-1</strong></td>
<td>MDC-1(KD1, KD2, IN1, IN2, OUT1, OUT2);&lt;br&gt;Set KD1mod := set KD1 bit 1 to B'1' and bit 2 to B'0' (bits 0-7)&lt;br&gt;Set KD2mod := set KD2 bit 1 to B'0' and bit 2 to B'1' (bits 0-7)&lt;br&gt;Set F1 := IN1 XOR eKD1mod(IN1)&lt;br&gt;Set F2 := IN2 XOR eKD2mod(IN2)&lt;br&gt;Set OUT1 := (bits 0..31 of F1)</td>
</tr>
<tr>
<td><strong>MDC-2</strong></td>
<td>MDC-2(n, text, KEY1, KEY2, MDC);&lt;br&gt;For i := 1, 2, ..., n do&lt;br&gt;Call MDC-1(KEY1, KEY2, T8&lt;i&gt;., T8&lt;i&gt;., OUT1, OUT2)&lt;br&gt;Set KEY1 := OUT1&lt;br&gt;Set KEY2 := OUT2&lt;br&gt;End do&lt;br&gt;Set output MDC := (KEY1</td>
</tr>
<tr>
<td><strong>MDC-4</strong></td>
<td>MDC-4(n, text, KEY1, KEY2, MDC);&lt;br&gt;For i := 1, 2, ..., n do&lt;br&gt;Call MDC-1(KEY1, KEY2, T8&lt;i&gt;., T8&lt;i&gt;., OUT1, OUT2)&lt;br&gt;Set KEY1int := OUT1&lt;br&gt;Set KEY2int := OUT2&lt;br&gt;Call MDC-1(KEY1int, KEY2int, KEY2, KEY1, OUT1, OUT2)&lt;br&gt;Set KEY1 := OUT1&lt;br&gt;Set KEY2 := OUT2&lt;br&gt;End do&lt;br&gt;Set output MDC := (KEY1</td>
</tr>
</tbody>
</table>
Table 204. MDC calculation procedures (continued)

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>eK(X)</td>
<td>DES encryption of plaintext X using key K</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive-OR operation</td>
</tr>
<tr>
<td>:=</td>
<td>Assignment operation</td>
</tr>
<tr>
<td>T8&lt;1&gt;</td>
<td>First 8-byte block of text</td>
</tr>
<tr>
<td>T8&lt;2&gt;</td>
<td>Second 8-byte block of text</td>
</tr>
<tr>
<td>KD1, KD2</td>
<td>64-bit quantities</td>
</tr>
<tr>
<td>IN1, IN2</td>
<td>64-bit quantities</td>
</tr>
<tr>
<td>OUT1, OUT2</td>
<td>64-bit quantities</td>
</tr>
<tr>
<td>n</td>
<td>Number of 8-byte blocks</td>
</tr>
</tbody>
</table>

**Ciphering methods**

The Data Encryption Standard (DES) algorithm defines operations on 8-byte data strings.

The DES algorithm is used in many different processes within CCA:
- Encrypting and decrypting general data
- Triple-encrypting and triple-decrypting PIN blocks
- Triple-encrypting and triple-decrypting CCA DES keys
- Triple-encrypting and triple-decrypting RSA private keys with several processes
- Deriving keys, hashing data, generating CVV values, and so forth

The Encipher and Decipher verbs describe how you can request encryption or decryption of application data. See the following topic: "General data-encryption processes" for a description of the two standardized processes you can use.

In CCA, PIN blocks are encrypted with double-length keys. The PIN block is encrypted with the left-half key, for which the result is decrypted with the right-half key and this result is encrypted with the left-half key.

See "Triple-DES ciphering algorithms" on page 706 and "Ciphering methods," which describe how CCA DES keys are enciphered.

**General data-encryption processes**

Although the fundamental concepts of enciphering and deciphering data are simple, different methods exist to process data strings that are not a multiple of eight bytes in length.

Two widely used methods for enciphering general data are defined in these ANSI standards:
- ANSI X3.106 cipher block chaining (CBC)
- ANSI X9.23

These methods also differ in how they define the initial chaining value (ICV).
This section describes how the Encipher and Decipher verbs implement these methods.

**Single-DES and Triple-DES encryption algorithms for general data**

Using the CEX®C, you can use the triple-DES algorithm in addition to the classical single-DES algorithm.

In the subsequent descriptions of the CBC method and ANSI X9.23 method, the actions of the Encipher and Decipher verbs encompass both single-DES and triple-DES algorithms. The triple-DES processes are depicted in Figure 18 where “left key” and “right key” refer to the two halves of a double-length DES key.

![Figure 18. Triple-DES data encryption and decryption](image)

**ANSI X3.106 Cipher Block Chaining (CBC) method**

ANSI standard X3.106 defines four modes of operation for ciphering, and one of these modes, Cipher Block Chaining (CBC), defines the basic method for ciphering multiple 8-byte data strings.

Figure 19 on page 704 and Figure 20 on page 704 show CBC using the Encipher and Decipher verbs. A plaintext data string that must be a multiple of eight bytes is processed as a series of 8-byte blocks. The ciphered result from processing an 8-byte block is XORed with the next block of 8 input bytes. The last 8-byte ciphered result is defined as an output chaining value (OCV). The security server stores the OCV in bytes 0 - 7 of the chaining_vector variable.

An ICV is XORed with the first block of eight bytes. When you call the Encipher or Decipher verb, specify the **INITIAL** or **CONTINUE** keywords. If you specify the **INITIAL** keyword, the default, the initialization vector from the verb parameter is XORed with the first eight bytes of data. If you specify the **CONTINUE** keyword, the OCV identified by the chaining_vector parameter is
XORed with the first eight bytes of data.

**ANSI X9.23 cipher block chaining**

ANSI X9.23 defines an enhancement to the basic cipher block chaining (CBC) mode of ANSI X3.106 so that the system can process data with a length that is not an exact multiple of eight bytes.
The ANSI X9.23 method always appends from 1 - 8 bytes to the plaintext before encipherment. The last appended byte is the count of the added bytes and is in the range of X'01' - X'08'. The standard defines that any other added bytes, or pad characters, be random.

When the coprocessor enciphers the plaintext, the resulting ciphertext is always 1 - 8 bytes longer than the plaintext. See Figure 21. This is true even if the length of the plaintext is a multiple of eight bytes. When the coprocessor deciphers the ciphertext, it uses the last byte of the deciphered data as the number of bytes to remove from the end (pad bytes, if any, and count byte). The result is the original plaintext. See Figure 22 on page 706.

The output chaining vector can be used as feedback with this method in the same way as with the X3.106 method.

The ANSI X9.23 method requires the caller to supply an initialization vector, and it does not allow specification of a pad character.

Note: The ANSI X9.23 standard has been withdrawn, but the X9.23 padding method is retained in CCA for compatibility with applications that rely on this method.

![Diagram](image-url)
**Triple-DES ciphering algorithms**

A triple-DES (TDES) algorithm is used to encrypt keys, PIN blocks, and general data.

Several techniques are employed:

**TDES ECB**

DES keys, when triple encrypted under a double-length DES key, are ciphered using an e-d-e scheme without feedback.

**TDES CBC**

Encryption of general data, and RSA section type X'08' CRT-format private keys and OPK keys, employs the scheme depicted in Figure 23 on page 707 and Figure 24 on page 707. This is often referred to as “outer CBC mode.”

This CCA supports double-length DES keys for triple-DES data encryption using the Encipher and Decipher verbs. The triple-length asymmetric master key is used to CBC encrypt CRT-format OPK keys.

**EDEx / DEDx**

CCA employs EDEx processes for encrypting several of the RSA private key formats (section types X'02', X'05', and X'06') and the OPK key in section type X'06'. The EDEx processes make successive use of single-key DES CBC processes. EDE2, EDE3, and EDE5 processes have been defined, based on the number of keys and initialization vectors used in the process. See Figure 25 on page 708 and Figure 26 on page 709. K1, K2, and K3 are true keys while “K4” and “K5” are initialization vectors. See Figure 25 on page 708 and Figure 26 on page 709.

---

![Diagram](image-url)  

*Figure 22. Deciphering using the ANSI X9.23 method*
For 2-key triple-DES, $K_c = K_a$  

*Figure 23. Triple-DES CBC encryption process*

For 2-key triple-DES, $K_c = K_a$  

*Figure 24. Triple-DES CBC decryption process*
Figure 25. EDE algorithm
MAC calculation methods

Four variations of DES-based message authentication can be used by the MAC Generate and MAC Verify verbs.

These variations are:
- "ANSI X9.9 MAC"
- "ANSI X9.19 Optional Procedure 1 MAC" on page 710
- "EMV MAC" on page 710
- "ISO 16609 TDES MAC" on page 711

A keyed-hash MAC (HMAC) based message authentication can be used by the HMAC Generate and HMAC Verify verbs.

ANSI X9.9 MAC

The Financial Institution (Wholesale) Message Authentication Standard (ANSI X9.9-1986) defines a process for the authentication of messages from originator to recipient, and this process is called the Message Authentication Code (MAC) calculation method.
Figure 27 shows the MAC calculation for binary data. In this figure, KEY is a 64-bit key, and T₁ - Tₙ are 64-bit data blocks of text. If Tₙ is less than 64 bits long, binary zeros are appended to the right of Tₙ. Data blocks T₁...Tₙ are DES CBC-encrypted with all output discarded except for the final output block, Oₙ.

**ANSI X9.19 Optional Procedure 1 MAC**


The CCA “X9.19OPT” process employs a double-length DES key. After calculating the 64-bit MAC as above with the left half of the double-length key, the result is decrypted using the right half of the double-length key. This result is then encrypted with the left half of the double-length key. The resulting MAC value is processed according to other specifications supplied to the verb call.

**EMV MAC**

The EMV smart card standards define MAC generation and verification processes that are the same as ANSI X9.9 and ANSI X9.19 Optional Procedure 1 (ISO/IEC 9797-1, Algorithm 3), except for padding added to the end of the message.

Append one byte of X’80’ to the original message. Then append additional bytes, as required, of X’00’ to form an extended message, which is a multiple of eight bytes in length.

In the ANSI X9.9 and ANSI X9.19 Optional Procedure 1 standards, the leftmost 32 bits (4 bytes) of Oₙ are taken as the MAC. In the EMV standards, the MAC value is between four and eight bytes in length. CCA provides support for the leftmost four, six, and eight bytes of MAC value.

---

3. The ANSI X9.9 standard defines five options. The MAC Generate and MAC Verify verbs implement option 1, binary data.
ISO 16609 TDES MAC

ISO 16609 defines a process for protecting the integrity of transmitted banking messages and for verifying that a message has originated from an authorized source and this process is called the ISO 16609 TDES MAC method.

The ISO 16609 TDES MAC method corresponds to ISO/IEC 9797-1, algorithm 1 using T-DEA (ANSI X9.52:1998). ISO/FDIS 16609 identifies this method as one of the recommended ways to generate a MAC using symmetric techniques.

The ISO 16609 TDES MAC method uses a double-length DES key and operates on data blocks that are a multiple of eight bytes. If the last input data block is not a multiple of eight bytes, binary zeros are appended to the right of the block. A CBC mode triple-DES (TDES) encryption operation is performed on the data, with all output discarded except for the final output block.

The resulting MAC value is processed according to other specifications supplied to the verb call.

Keyed-hash MAC (HMAC)

The Keyed-Hash Message Authentication Code (HMAC) standard (FIPS PUB 198-1) describes a mechanism for message authentication using cryptographic hash functions.

HMAC can be used with a hash function in combination with a shared secret key.

To see how to compute a MAC over the data, see FIPS PUB 198-1 available at: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf.

RSA key-pair generation

RSA key-pair generation is determined based on user input of the modulus bit length, public exponent, and key type.

The output is based on creating primes \( p \) and \( q \) in conformance with ANSI X9.31 requirements as follows:

- prime \( p \) bit length = ((modulus_bit_length +1)/2)
- prime \( q \) bit length = modulus_bit_length - \( p \) bit length
- \( p \) and \( q \) are randomly chosen prime numbers
- \( p > q \)
- The Rabin-Miller Probabilistic Primality Test is iterated 8 times for each prime. This test determines that a false prime is produced with probability no greater than \( 1/4^c \), where \( c \) is the number of iterations. Refer to the ANSI X9.31 standard and see the section entitled “Miller-Rabin Probabilistic Primality Test.”
- Primes \( p \) and \( q \) are relatively prime with the public exponent.
- Primes \( p \) and \( q \) are different in at least one of the first 100 most significant bits, that is, \( |p-q| > 2^{(prime \ bit \ length \ - \ 100)} \). For example, when the modulus bit length is 1024, then both primes bit length are 512 bits and the difference of the two primes is \( |p-q| > 2^{412} \).
- For each key generation, and for any size of key, the PKA manager seeds an internal FIPS-approved, SHA-1 based pseudo random number generator (PRNG) with the first 24 bytes of information that it receives from three successive calls to the random number generator (RNG) manager’s PRNG interface.
2. The RNG manager can supply random number in two ways, but with the CCA Support Program only one way is used, namely, the PRNG method. The PKA manager seeds an internal FIPS-approved, SHA-1 based PRNG with 24 bytes obtained.

The RNG manager can respond to requests for random numbers from other processes with such responses interspersed between responses to PKA manager requests. An RSA key is generated from random information obtained from two cascaded SHA-1 PRNGs.

3. An RSA key is based on one or more 24-byte seeds from the RNG manager source, depending on the dynamic mix of tasks running inside the coprocessor.

There exists a system RNG manager (ANSI X9.31 compliant) that is used as the source for pseudo random numbers. The PKA manager also has a PRNG that is DSA compliant for generating primes. The PKA manager PRNG is re-seeded from the system RNG manager, for every new key pair generation, which is for every generation of a public/private key pair.

### Multiple decipherment and encipherment

This section explains multiple encipherment and decipherment and their equations.

CCA uses multiple encipherment whenever it enciphers a key under a key-encrypting key such as the master key or the transport key and in triple-DES encipherment for data privacy. Multiple encipherment is superior to single encipherment because multiple encipherment increases the work needed to “break” a key. CCA provides extra protection for a key by enciphering it under an enciphering key multiple times rather than once. The multiple encipherment method for keys enciphered under a key-encrypting key uses a double-length (128-bit) key split into two 64-bit halves. Like single encipherment, multiple encipherment uses a DES based on the electronic code book (ECB) mode of encipherment.

Keys can either be double-length or single-length depending on the installation and their cryptographic function. When a single-length key is encrypted under a double-length key, multiple encipherment is performed on the key. In the multiple encipherment method, the key is encrypted under the left half of the enciphering key. The result is then decrypted under the right half of the enciphering key. Finally, this result is encrypted under the left half of the enciphering key again.

When a double-length key is encrypted with multiple encipherment, the method is similar, except CCA uses two enciphering keys. One enciphering key encrypts each half of the double-length key. Double-length keys active on the system have two master key variants used when enciphering them.

Multiple encipherment and decipherment is not only used to protect or retrieve a cryptographic key, but they are also used to protect or retrieve 64-bit data in the area of PIN applications. For example, the following two sections use a double-length *KEK as an example to cipher a single-length key even though the same algorithms apply to cipher 64-bit data by a double-length PIN-related cryptographic key.

CCA also supports triple-DES encipherment for data privacy using double-length and triple-length DATA keys. For this procedure the data is first enciphered using the first DATA key. The result is then deciphered using the second DATA key. This
second result is then enciphered using the third DATA key when a triple-length key is provided or reusing the first DATA key when a double-length key is provided.

Note that an asterisk (*) preceding the key means the key is double-length. Notations in this chapter have the following meaning:

- $e_K(x)$, where $x$ is enciphered under $K$
- $d_K(y)$ represents plaintext, where $K$ is the key and $y$ is the ciphertext

Therefore, $d_K(e_K(x))$ equals $x$ for any 64-bit key $K$ and any 64-bit plaintext $x$.

When a key (*K) to be protected is double-length, two double-length *KEKs are used. One *KEK is used for protecting the left half of the key (*K); another is for the right half. Multiple encipherment is used with the appropriate *KEK for protecting each half of the key.

**Multiple encipherment of single-length keys**

Definition of the multiple encipherment of a single-length key ($K$) using a double-length *KEK.

The multiple encipherment of a single-length key ($K$) using a double-length *KEK is defined as follows:

$$e_{*KEK}(K) = e_{KEKL}(d_{KEKR}(e_{KEKL}(K)))$$

where $KEKL$ is the left 64 bits of *KEK and $KEKR$ is the right 64 bits of *KEK.

**Figure 28** illustrates the definition.

**Multiple decipherment of single-length keys**

Definition of the multiple encipherment of an encrypted single-length key ($Y = e_{*KEK}(K)$) using a double-length *KEK.
The multiple encipherment of an encrypted single-length key \( Y = e^{*}\text{KEK}(K) \) using a double-length \(*\text{KEK}\) is defined as follows:

\[
d^{*}\text{KEK}(Y) = d\text{KEKL}(e\text{KEKR}(d\text{KEKL}(Y)))
\]

\[
= d^{*}\text{KEK}(e^{*}\text{KEK}(K))
\]

\[
= K
\]

where \( \text{KEKL} \) is the left 64 bits of \(*\text{KEK}\) and \( \text{KEKR} \) is the right 64 bits of \(*\text{KEK}\).

**Figure 29** illustrates the definition.

**Multiple encipherment of single-length keys**

**Multiple encipherment of double-length keys**

Definition of the multiple encipherment of a double-length key \((*K)\) using two double-length \(*\text{KEKs}\), \(*\text{KEKa}\), and \(*\text{KEKb}\).

The multiple encipherment of a double-length key \((*K)\) using two double-length \(*\text{KEKs}\), \(*\text{KEKa}\), and \(*\text{KEKb}\) is defined as follows:

\[
e^{*}\text{KEKa}(KL) || e^{*}\text{KEKb}(KR) = e\text{KEKaL}(d\text{KEKaR}(e\text{KEKaL}(KL))) || e\text{KEKbL}(d\text{KEKbR}(e\text{KEKbL}(KR)))
\]

where:

- \( KL \) is the left 64 bits of \(*K\)
- \( KR \) is the right 64 bits of \(*K\)
- \( \text{KEKaL} \) is the left 64 bits of \(*\text{KEKa}\)
- \( \text{KEKaR} \) is the right 64 bits of \(*\text{KEKa}\)
- \( \text{KEKbL} \) is the left 64 bits of \(*\text{KEKb}\)
- \( \text{KEKbR} \) is the right 64 bits of \(*\text{KEKb}\)
- \(||\) means concatenation

**Figure 30 on page 715** illustrates the definition.
Multiple decipherment of double-length keys

Definition of the multiple decipherment of an encrypted double-length key, \( Y = e^{*KEK_a}(KL) \| e^{*KEK_b}(KR) \), using two double-length \(*KEKs, *KEK_a, and *KEK_b.\)

The multiple decipherment of an encrypted double-length key, \( Y = e^{*KEK_a}(KL) \| e^{*KEK_b}(KR) \), using two double-length \(*KEKs, *KEK_a, and *KEK_b, is defined as follows:

\[
D^{*KEK_a}(YL) \| d^{*KEK_b}(YR) = d^{*KEK_a}(e^{*KEK_a}(KL)) \| d^{*KEK_b}(e^{*KEK_b}(KR)) = *K
\]

where

- \( YL \) is the left 64 bits of \( Y \)
- \( YR \) is the right 64 bits of \( Y \)
- \( KEK_aL \) is the left 64 bits of \( *KEK_a \)
- \( KEK_aR \) is the right 64 bits of \( *KEK_a \)
- \( KEK_bL \) is the left 64 bits of \( *KEK_b \)
- \( KEK_bR \) is the right 64 bits of \( *KEK_b \)
- \( \| \) means concatenation

Figure 31 on page 716 illustrates the definition.
Multiple encipherment of triple-length keys

Definition of the multiple encipherment of a triple-length key (**K) using two double-length *KEKs, *KEKa, and *KEKb.

The multiple encipherment of a triple-length key (**K) using two double-length *KEKs, *KEKa, and *KEKb is defined as follows:

\[
e^{*\text{KEKa}}(\text{KL}) \ || \ e^{*\text{KEKb}}(\text{KM}) \ || \ e^{*\text{KEKa}}(\text{KR}) = \\]

\[
e^{\text{KEKaL}}(d^{\text{KEKaR}}(e^{\text{KEKaL}}(\text{KL}))) \ || \ e^{\text{KEKbL}}(d^{\text{KEKbR}}(e^{\text{KEKbL}}(\text{KM}))) \ || \ e^{\text{KEKaL}}(d^{\text{KEKaR}}(e^{\text{KEKaL}}(\text{KR})))
\]

where:
- KL is the left 64 bits of **K
- KM is the next 64 bits of **K
- KR is the right 64 bits of **K
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- || means concatenation

Figure 31 on page 717 illustrates the definition.
Multiple decipherment of triple-length keys

Definition of the multiple decipherment of an encrypted triple-length key $**Y = e^{*KEKa}^{(KL)} \ || \ e^{*KEKb}^{(KM)} \ || \ e^{*KEKa}^{(KR)}$, using two double-length *KEKs, *KEKa, and *KEKb.

The multiple decipherment of an encrypted triple-length key $**Y = e^{*KEKa}^{(KL)} \ || \ e^{*KEKb}^{(KM)} \ || \ e^{*KEKa}^{(KR)}$, using two double-length *KEKs, *KEKa, and *KEKb, is defined as follows:

$$
\begin{align*}
\text{d}^{*KEKa}(YL) \ || \ \text{d}^{*KEKb}(YM) \ || \ \text{d}^{*KEKa}(YR) \\
= \text{d}^{KEKaL}(e^{KEKaR}(d^{KEKaL}(YL))) \ || \ \\
= \text{d}^{KEKbL}(e^{KEKbR}(d^{KEKbL}(YM))) \\
= \text{d}^{KEKaL}(e^{KEKaR}(d^{KEKaL}(YR))) \\
= \text{d}^{*KEKa}(e^{*KEKa}(KL)) \ || \ \\
= \text{d}^{*KEKb}(e^{*KEKb}(KM)) \ || \ \\
= \text{d}^{*KEKa}(e^{*KEKa}(KR)) \\
= **K
\end{align*}
$$

where:
- YL is the left 64 bits of **Y
- YM is the next 64 bits of **Y
- YR is the right 64 bits of **Y
- KEKaL is the left 64 bits of *KEKa
- KEKaR is the right 64 bits of *KEKa
- KEKbL is the left 64 bits of *KEKb
- KEKbR is the right 64 bits of *KEKb
- || means concatenation
Figure 33 illustrates the definition.

The Symmetric Key Generate and the Symmetric Key Import verbs optionally support a PKA92 method of encrypting a DES key with an RSA public key.

This format is adapted from the IBM Transaction Security System (TSS) 4753 and 4755 product's implementation of "PKA92". The verbs do not create or accept the complete PKA92 AS key token as defined for the TSS products. Rather, the verbs support only the actual RSA-encrypted portion of a TSS PKA92 key token, the AS External Key Block.

Forming an external key block - The PKA96 implementation forms an AS External Key Block by RSA-encrypting a key block using a public key. The key block is formed by padding the key record detailed in Table 205 with zero bits on the left, high-order end of the key record. The process completes the key block with three sub-processes: masking, overwriting, and RSA encrypting.

Table 205. PKA96 clear DES key record

<table>
<thead>
<tr>
<th>Offset (Bytes)</th>
<th>Length (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>005</td>
<td>Header and flags: X'01 0000 0000.'</td>
</tr>
<tr>
<td>005</td>
<td>016</td>
<td>Environment Identifier (EID), encoded in ASCII.</td>
</tr>
</tbody>
</table>
### Table 205. PKA96 clear DES key record (continued)

<table>
<thead>
<tr>
<th>Offset (Bytes)</th>
<th>Length (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>021</td>
<td>008</td>
<td>Control vector base for the DES key.</td>
</tr>
<tr>
<td>029</td>
<td>008</td>
<td>Repeat of the CV data at offset 021.</td>
</tr>
<tr>
<td>037</td>
<td>008</td>
<td>The single-length DES key or the left half of a double-length DES key.</td>
</tr>
<tr>
<td>045</td>
<td>008</td>
<td>The right half of a double-length DES key or a random number. This value is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>locally designated &quot;K.&quot;</td>
</tr>
<tr>
<td>053</td>
<td>008</td>
<td>Random number, &quot;IV.&quot;</td>
</tr>
<tr>
<td>061</td>
<td>001</td>
<td>Ending byte, X'00.'</td>
</tr>
</tbody>
</table>

**Masking Sub-process** - Create a mask by CBC encrypting a multiple of eight bytes of binary zeros using K as the key and IV as the initialization vector as defined in the key record at offsets 45 and 53. XOR the mask with the key record and call the result PKR.

**Overwriting Sub-process** - Set the high-order bits of PKR to B'01' and set the low-order bits to B'0110'.

XOR K and IV and write the result at offset 45 in PKR.

Write IV at offset 53 in PKR. This causes the masked and overwritten PKR to have IV at its original position.

**Encrypting Sub-process** - RSA encrypt the overwritten PKR masked key record using the public key of the receiving node.

**Recovering a key from an external key block** - Recover the encrypted DES key from an AS External Key Block by performing decrypting, validating, unmasking, and extraction sub-processes.

**Decrypting Sub-process** - RSA decrypt the AS External Key Block using an RSA private key and call the result of the decryption PKR. The private key must be usable for key management purposes.

**Validating Sub-process** - Verify the high-order two bits of the PKR record are valued to B'01' and the low-order four bits of the PKR record are valued to B'0110'.

**Unmasking Sub-process** - Set IV to the value of the eight bytes at offset 53 of the PKR record. Note that there is a variable quantity of padding prior to offset 0. See Table 205 on page 718.

Set K to the XOR of IV and the value of the eight bytes at offset 45 of the PKR record.

Create a mask equal in length to the PKR record by CBC encrypting a multiple of eight bytes of binary zeros using K as the key and IV as the initialization vector. XOR the mask with PKR and call the result the key record.

Copy K to offset 45 in the PKR record.

**Extraction Sub-process.** Confirm that:
• The four bytes at offset 1 in the key record are valued to X'0000 0000'.
• The two control vector fields at offsets 21 and 29 are identical.
• If the control vector is an IMPORTER or EXPORTER key class, the Environment Identifier (EID) in the key record is not the same as the EID stored in the cryptographic engine.

The control vector base of the recovered key is the value at offset 21. If the control vector base bits 40 - 42 are valued to B'010' or B'110', the key is double length. Set the right half of the received key's control vector equal to the left half and reverse bits 41 and 42 in the right half.

The recovered key is at offset 37 and is either 8 or 16 bytes long based on the control vector base bits 40 - 42. If these bits are valued to B'000', the key is single length. If these bits are valued to B'010' or B'110', the key is double length.

---

**Formatting hashes and keys in public-key cryptography**

The Digital Signature Generate and Digital Signature Verify verbs support several methods for formatting a hash and, in some cases, a descriptor for the hashing method, into a bit-string to be processed by the cryptographic algorithm.

This section provides information about the ANSI X9.31 and PKCS #1 methods. The ISO 9796-1 method can be found in the ISO standard.

This section also describes the PKCS #1, version 1, 1.5, and 2.0, methods for placing a key in a bit string for RSA ciphering as part of a key exchange.

**ANSI X9.31 hash format**

With ANSI X9.31, the string that is processed by the RSA algorithm is formatted by the concatenation of a header, padding, the hash value and a trailer, from the most significant bit to the least significant bit, so that the resulting string is the same length as the modulus of the key.

For CCA, the modulus length must be a multiple of 8 bits.
• The header consists of the value X'6B'.
• The padding consists of the value X'BB', repeated as many times as required, and ended with X'BA'.
• The hash value follows the padding.
• The trailer consists of a hashing mechanism specifier and final byte. The hashing mechanism specifier is defined as one of the following values:
  - X'31' RIPEMD-160
  - X'32' RIPEMD-128
  - X'33' SHA-1
  - X'34' SHA-256 (Release 3.30.05 or later)
• The final byte is X'CC'.

**PKCS #1 formats**

Version 2.0 of the PKCS #1 standard defines methods for formatting keys and hashes prior to RSA encryption of the resulting data structures.
The earlier versions of the PKCS #1 standard defined block types 0, 1, and 2, but in the current standard that terminology is dropped.

CCA implemented these processes using the terminology of the Version 2.0 standard:

- For formatting keys for secured transport Symmetric Key Export, Digital Signature Generate, Symmetric Key Import):
  - RSAES-OAEP, the preferred method for key-encipherment when exchanging DATA keys between systems. Keyword PKCSOAEP (Version 2.0) and PKOAEP2 (Version 2.1) is used to invoke this formatting technique. The P parameter described in the standard is not used and its length is set to zero.
  - RSAES-PKCS1-v1_5, is an older method for formatting keys. It is included for compatibility with existing applications. Keyword PKCS-1.2 is used to invoke this formatting technique.

- For formatting hashes for digital signatures (Digital Signature Generate and Digital Signature Verify):
  - RSASSA-PKCS1-v1_5, the newer name for the block-type 1 format. Keyword PKCS-1.1 is used to invoke this formatting technique.
  - The PKCS #1 specification no longer describes the use of block-type 0. Keyword PKCS-1.0 is used to invoke this formatting technique. Use of block-type 0 is discouraged.

Using the terminology from older versions of the PKCS #1 standard, block types 0 and 1 are used to format a hash and block type 2 is used to format a DES key. The blocks consist of the following ("||" means concatenation):

- X'00' || BT || PS || X'00' || D

where:

- BT is the block type, X'00', X'01', or X'02'.
- PS is the padding of as many bytes as required to make the block the same length as the modulus of the RSA key. Padding of X'00' is used for block type 0, X'FF' for block type 1, and random and non-X'00' for block type 2. The length of PS must be a minimum of eight bytes.
- D is the key, or the concatenation of the BER-encoded hash identifier and the hash value.

You can create the ASN.1 BER encoding of an MD5, SHA-1, or SHA-256 value by prepending these strings to the 16-byte or 20-byte hash values, respectively:

- **MD5**
  
  X'3020300C 06082A86 4886F70D 02050500 0410'

- **SHA-1**
  
  X'30213009 06052B0E 03021A05 000414'

- **SHA-256**
  
  X'3031300D 06096086 48016503 04020105 000420'

---


5. The PKA 92 method and the method incorporated into the SET standard are other examples of the Optimal Asymmetric Encryption Padding (OAEP) technique. The OAEP technique is attributed to Bellare and Rogaway.
Appendix G. Access control points and verbs

This appendix gives details about the Access Control Points (ACPs) used by the verbs in this document. ACPs are also referred to as commands.

**Important:** By default, you should disable commands. Do not enable an ACP unless you know why you are enabling it.

For instructions on how to enable and disable these ACPs using the TKE workstation, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User's Guide*.

For systems that do not use the optional TKE workstation, most ACPs (current and new) are enabled in the DEFAULT role with the appropriate licensed internal code on the CEX*C.

Note that each domain in the CEX*C (with hardware enforced access permissions) starts out with its own DEFAULT role with the default ACP values as shown. However, it is possible to use the TKE to change ACP values in the DEFAULT role or to define other roles. The role to which a user is assigned determines the ACPs available to that user. Full coverage of TKE use for configuration is outside the scope of this document. For details, see *z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User’s Guide*.

Table 206 on page 724 lists the CCA ACPs. The name of each ACP is given as it appears on the panels of the TKE user interface. Note that the group names are also given to aid locating the ACPs. The table includes the following columns:

- **ACP number**: The hexadecimal offset, or ACP code, for the command. Offsets between X'0000' and X'FFFF' that are not listed in this table are reserved.
- **Name of ACP from TKE interface**: The name of the ACP as it appears on the TKE interface
- **Verb name**: The names of the verbs that require that ACP to be enabled; for example, the Encipher (CSNBENC) verb fails without permission to use the Encipher ACP.
- **Entry point**: The entry-point name of the verb.
- **Initial setting**: Whether the ACP is ON or OFF by default.
- **Usage**: Usage recommendations for the ACP. The abbreviations in this column are explained at the end of the table.

See the **Restrictions**, **Required commands**, or **Usage notes** sections at the end of each verb description for access control information.
<table>
<thead>
<tr>
<th>ACP number (hex)</th>
<th>Name of ACP from TKE interface</th>
<th>Verb name</th>
<th>Entry point</th>
<th>Initial setting</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 0001</td>
<td>GROUP: ISPF Services</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0018'</td>
<td>Load First DES Master Key Part</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0019'</td>
<td>Combine DES Master Key Parts</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'001A'</td>
<td>Set DES Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0032'</td>
<td>Clear New DES Master Key Register</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'0053'</td>
<td>Load First RSA Master Key Part</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0054'</td>
<td>Combine RSA Master Key Parts</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0057'</td>
<td>Set RSA Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0060'</td>
<td>Clear New RSA Master Key Register</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0124'</td>
<td>Clear New AES Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'0125'</td>
<td>Load First AES Master Key Part</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'0126'</td>
<td>Combine AES Master Key Parts</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'0128'</td>
<td>Set AES Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'031F'</td>
<td>Clear New ECC Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0320'</td>
<td>Load First ECC Master Key Part</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0321'</td>
<td>Combine ECC Master Key Parts</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0322'</td>
<td>Set ECC Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0326'</td>
<td>Generate ECC keys in the clear</td>
<td>PKA Key Generate</td>
<td>CSNDPKG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>0002 0002</td>
<td>GROUP: API Cryptographic Services</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'000E'</td>
<td>Encipher - DES</td>
<td>Encipher</td>
<td>CSNBENC</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'000F'</td>
<td>Decipher - DES</td>
<td>Decipher</td>
<td>CSNBDEC</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0010'</td>
<td>MAC Generate</td>
<td>MAC Generate</td>
<td>CSNBMGN</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0011'</td>
<td>MAC Verify</td>
<td>MAC Verify</td>
<td>CSNBMVFR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0012'</td>
<td>Key Import</td>
<td>Key Import</td>
<td>CSNBMVIM</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0013'</td>
<td>Key Export</td>
<td>Key Export</td>
<td>CSNBKEX</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'001B'</td>
<td>Key Part Import - first key part</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'001C'</td>
<td>Key Part Import - middle and last</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
</tbody>
</table>

**Note:** This group name refers to ISPF, a z/OS feature. Although ISPF is not relevant to Linux on IBM System z, it is listed here as shown on the TKE panels to avoid confusion.
Table 206. Access Control Points and corresponding CCA verbs (continued)

<table>
<thead>
<tr>
<th>ACP number (hex)</th>
<th>Name of ACP from TKE interface</th>
<th>Verb name</th>
<th>Entry point</th>
<th>Initial setting</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'001D'</td>
<td>Key Test and Key Test2</td>
<td>Key Test</td>
<td>CSNBKYT</td>
<td>ON R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Key Test2</td>
<td>CSNBKYT2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Key Test Extended</td>
<td>CSNBKYTX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Key Storage Initialization</td>
<td>CSNBKSI</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AES Key Record Create</td>
<td>CSNBAKRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AES Key Record Delete</td>
<td>CSNBAKRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AES Key Record List</td>
<td>CSNBAKRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AES Key Record Read</td>
<td>CSNBAKRR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AES Key Record Write</td>
<td>CSNBAKRW</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DES Key Record Create</td>
<td>CSNBDKRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DES Key Record Delete</td>
<td>CSNBDKRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DES Key Record List</td>
<td>CSNBDKRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DES Key Record Read</td>
<td>CSNBDKRR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DES Key Record Write</td>
<td>CSNBDKRW</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PKA Key Record Create</td>
<td>CSNDKRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PKA Key Record Delete</td>
<td>CSNDKRD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PKA Key Record List</td>
<td>CSNDKRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PKA Key Record Read</td>
<td>CSNDKRR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PKA Key Record Write</td>
<td>CSNDKRW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'001E'</td>
<td>Reencipher CKDS</td>
<td>Key Token Change</td>
<td>CSNBKTC</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> The TKE name for this ACP refers to z/OS key storage (CKDS). However z/OS key storage is not impacted. This ACP refers to a service for Linux, see verb for details.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'001F'</td>
<td>Key Translate</td>
<td>Key Translate</td>
<td>CSNBKTR</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0021'</td>
<td>Key Test2 - AES, ENC-ZERO</td>
<td>Key Test21</td>
<td>CSNBKYT2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0040'</td>
<td>Diversified Key Generate - CLR8-ENC</td>
<td>Diversified Key Generate2</td>
<td>CSNBDKG</td>
<td>ON O, SEL</td>
<td></td>
</tr>
<tr>
<td>X'0042'</td>
<td>Diversified Key Generate - TDES-DEC</td>
<td>Diversified Key Generate2</td>
<td>CSNBDKG</td>
<td>ON O, SEL</td>
<td></td>
</tr>
<tr>
<td>X'0043'</td>
<td>Diversified Key Generate - SESS-XOR</td>
<td>Diversified Key Generate2</td>
<td>CSNBDKG</td>
<td>ON O, SEL</td>
<td></td>
</tr>
<tr>
<td>X'0044'</td>
<td>Diversified Key Generate - Single length or same halves</td>
<td>Diversified Key Generate2</td>
<td>CSNBDKG</td>
<td>ON SC, SEL</td>
<td></td>
</tr>
<tr>
<td>X'0045'</td>
<td>Diversified Key Generate - TDES-XOR</td>
<td>Diversified Key Generate2</td>
<td>CSNBDKG</td>
<td>ON O, SEL</td>
<td></td>
</tr>
<tr>
<td>X'0046'</td>
<td>Diversified Key Generate - TDESEMV2/TDESEMV4</td>
<td>Diversified Key Generate2</td>
<td>CSNBDKG</td>
<td>ON O, SEL</td>
<td></td>
</tr>
<tr>
<td>X'008A'</td>
<td>MDC Generate</td>
<td>MDC Generate</td>
<td>CSNBMMDG</td>
<td>OFF R</td>
<td></td>
</tr>
<tr>
<td>X'008C'</td>
<td>Key Generate - OPIM_OPEX_IMEX, etc.</td>
<td>Key Generate2</td>
<td>CSNBKGN</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'008E'</td>
<td>Key Generate - OP_IM_EX</td>
<td>Key Generate2</td>
<td>CSNBKGN</td>
<td>ON R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Number Generate</td>
<td>CSNBRNG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0090'</td>
<td>DES Key Token Change</td>
<td>Key Token Change</td>
<td>CSNBKTC</td>
<td>ON R</td>
<td></td>
</tr>
<tr>
<td>X'00A0'</td>
<td>Clear PIN Generate - 3624</td>
<td>Clear PIN Generate</td>
<td>CSNPBGN</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'00A1'</td>
<td>Clear PIN Generate - GBP</td>
<td>Clear PIN Generate</td>
<td>CSNPBGN</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'00A2'</td>
<td>Clear PIN Generate - VISA PVV</td>
<td>Clear PIN Generate</td>
<td>CSNPBGN</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'00A3'</td>
<td>Clear PIN Generate - Interbank</td>
<td>Clear PIN Generate</td>
<td>CSNPBGN</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'00A4'</td>
<td>Clear PIN Generate Alternate - 3624 Offset</td>
<td>Clear PIN Generate Alternate</td>
<td>CSNBCPA</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00A5'</td>
<td>Encrypted PIN Verify - 3624</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00A6'</td>
<td>Encrypted PIN Verify - GBP</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00A7'</td>
<td>Encrypted PIN Verify - VISA PVV</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00A8'</td>
<td>Encrypted PIN Verify - Interbank</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00A9'</td>
<td>Clear PIN Encrypt</td>
<td>Clear PIN Encrypt</td>
<td>CSNBCPE</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B0'</td>
<td>Encrypted PIN Generate - 3624</td>
<td>Encrypted PIN Generate</td>
<td>CSNBEPG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B1'</td>
<td>Encrypted PIN Generate - GBP</td>
<td>Encrypted PIN Generate</td>
<td>CSNBEPG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B2'</td>
<td>Encrypted PIN Generate - Interbank</td>
<td>Encrypted PIN Generate</td>
<td>CSNBEPG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B3'</td>
<td>Encrypted PIN Translate - Translate</td>
<td>Encrypted PIN Translate</td>
<td>CSNBPTR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B4'</td>
<td>Encrypted PIN Translate - Reformat</td>
<td>Encrypted PIN Translate</td>
<td>CSNBPTR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B6'</td>
<td>Clear PIN Generate Alternate - VISA PVV</td>
<td>Clear PIN Generate Alternate</td>
<td>CSNBCPA</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B7'</td>
<td>PIN Change/Unblock - change EMV PIN with OPINENC</td>
<td>PIN Change/Unblock</td>
<td>CSNBPCU</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00B8'</td>
<td>PIN Change/Unblock - change EMV PIN with IPINENC</td>
<td>PIN Change/Unblock</td>
<td>CSNBPCU</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00C3'</td>
<td>Clear Key Import/Multiple Clear Key Import - DES</td>
<td>Clear Key Import</td>
<td>CSNBCKI</td>
<td>ON</td>
<td>SC</td>
</tr>
<tr>
<td>X'00C4'</td>
<td>Secure Key Import - DES_OP</td>
<td>Secure Key Import</td>
<td>CSNBCKM</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00CD'</td>
<td>Prohibit Export</td>
<td>Prohibit Export</td>
<td>CSNBPEX</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00D5'</td>
<td>Control Vector Translate</td>
<td>Control Vector Translate</td>
<td>CSNBCVT</td>
<td>ON</td>
<td>SC</td>
</tr>
<tr>
<td>X'00D6'</td>
<td>Key Generate - OPIM_OPEX_IMEX, etc. extended</td>
<td>Key Generate</td>
<td>CSNBKGN</td>
<td>ON</td>
<td>SC, SUP</td>
</tr>
<tr>
<td>X'00DA'</td>
<td>Cryptographic Variable Encipher</td>
<td>Cryptographic Variable Encipher</td>
<td>CSNBCVE</td>
<td>ON</td>
<td>NRP, O, SUP</td>
</tr>
<tr>
<td>X'00D7'</td>
<td>Key Generate - SINGLE-R</td>
<td>Key Generate</td>
<td>CSNBKGN</td>
<td>ON</td>
<td>NR, SC</td>
</tr>
<tr>
<td>X'00DC'</td>
<td>Secure Key Import - DES_IM</td>
<td>Secure Key Import</td>
<td>CSNBPCU</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00DF'</td>
<td>VISA CVV Generate</td>
<td>VISA CVV Generate</td>
<td>CSNBCSG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00E0'</td>
<td>VISA CVV Verify</td>
<td>VISA CVV Verify</td>
<td>CSNBCSV</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00E1'</td>
<td>UKPT - PIN Verify, PIN Translate</td>
<td>Encrypted PIN Translate</td>
<td>CSNBPTR</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00E2'</td>
<td>HMAC Generate - SHA-1</td>
<td>HMAC Generate</td>
<td>CSNBHMG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00E3'</td>
<td>HMAC Generate - SHA-224</td>
<td>HMAC Generate</td>
<td>CSNBHMG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00E4'</td>
<td>HMAC Generate - SHA-256</td>
<td>HMAC Generate</td>
<td>CSNBHMG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00E5'</td>
<td>HMAC Generate - SHA-384</td>
<td>HMAC Generate</td>
<td>CSNBHMG</td>
<td>ON</td>
<td>O</td>
</tr>
</tbody>
</table>
Table 206. Access Control Points and corresponding CCA verbs (continued)

<table>
<thead>
<tr>
<th>ACP number (hex)</th>
<th>Name of ACP from TKE interface</th>
<th>Verb name</th>
<th>Entry point</th>
<th>Initial setting</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'00E8'</td>
<td>HMAC Generate - SHA-512</td>
<td>HMAC Generate</td>
<td>CSNBHMG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00E9'</td>
<td>Restrict Key Attribute - Export Control</td>
<td>Restrict Key Attribute</td>
<td>CSNBRKA</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00EA'</td>
<td>Key Generate2 - OP_EX_IM</td>
<td>Key Generate2</td>
<td>CSNBKGN2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00EB'</td>
<td>Key Generate2 - OPOP_OPI1_OPEX_etc.</td>
<td>Key Generate2</td>
<td>CSNBKGN2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F0'</td>
<td>Symmetric Key Token Change2</td>
<td>Key Token Change2</td>
<td>CSNBKTC2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F1'</td>
<td>Symmetric Key Token Change2 - RTCMK</td>
<td>Key Token Change2</td>
<td>CSNBKTC2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F2'</td>
<td>Secure Key Import2 - HMAC_OP</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F4'</td>
<td>Symmetric Key Import2 - HMAC_PKCSAOAEP</td>
<td>Symmetric Key Import2</td>
<td>CSNDSYI2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F5'</td>
<td>Symmetric Key Export - HMAC_PKCSAOAEP</td>
<td>Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F7'</td>
<td>HMAC Verify - SHA-1</td>
<td>HMAC Verify</td>
<td>CSNBHMOV</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F8'</td>
<td>HMAC Verify - SHA-224</td>
<td>HMAC Verify</td>
<td>CSNBHMOV</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00F9'</td>
<td>HMAC Verify - SHA-256</td>
<td>HMAC Verify</td>
<td>CSNBHMOV</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00FA'</td>
<td>HMAC Verify - SHA-384</td>
<td>HMAC Verify</td>
<td>CSNBHMOV</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00FB'</td>
<td>HMAC Verify - SHA-512</td>
<td>HMAC Verify</td>
<td>CSNBHMOV</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00FC'</td>
<td>Symmetric Key Export - AES, PKOAEP2</td>
<td>Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'00FD'</td>
<td>Import AES Key (PKOAEP2)</td>
<td>Symmetric Key Import2</td>
<td>CSNDSYI2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0100'</td>
<td>Digital Signature Generate</td>
<td>Digital Signature Generate</td>
<td>CSNDDSG</td>
<td>ON, SC</td>
<td></td>
</tr>
<tr>
<td>X'0101'</td>
<td>Digital Signature Verify</td>
<td>Digital Signature Verify</td>
<td>CSNDDSv</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0102'</td>
<td>PKA Key Token Change RTCMK</td>
<td>PKA Key Token Change</td>
<td>CSNDKTC</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0103'</td>
<td>PKA Key Generate</td>
<td>PKA Key Generate</td>
<td>CSNDPKG</td>
<td>ON, SUP</td>
<td></td>
</tr>
<tr>
<td>X'0104'</td>
<td>PKA Key Import</td>
<td>PKA Key Import</td>
<td>CSNDPKI</td>
<td>ON, SUP</td>
<td></td>
</tr>
<tr>
<td>X'0105'</td>
<td>Symmetric Key Export - DES_PKCS-1.2</td>
<td>Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>ON SC</td>
<td></td>
</tr>
<tr>
<td>X'0106'</td>
<td>Symmetric Key Import - DES_PKCS-1.2</td>
<td>Symmetric Key Import</td>
<td>CSNDSYI</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0109'</td>
<td>Data Key Import</td>
<td>Data Key Import</td>
<td>CSNBDKM</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'010A'</td>
<td>Data Key Export</td>
<td>Data Key Export</td>
<td>CSNBDKX</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'010B'</td>
<td>SET Block Compose</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'010C'</td>
<td>SET Block Decompose</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'010D'</td>
<td>Symmetric Key Generate - DES_PKA92</td>
<td>Symmetric Key Generate</td>
<td>CSNDSYG</td>
<td>ON SC</td>
<td></td>
</tr>
</tbody>
</table>
## Table 206. Access Control Points and corresponding CCA verbs (continued)

<table>
<thead>
<tr>
<th>ACP number (hex)</th>
<th>Name of ACP from TKE interface</th>
<th>Verb name</th>
<th>Entry point</th>
<th>Initial setting</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0116'</td>
<td>Access Control Manager - Read role</td>
<td>Note: This ACP is included for reference only. The service impacted is available only for IBM System x, IBM System p, or using the TKE interface.</td>
<td>ON O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'011E'</td>
<td>PKA Encrypt</td>
<td>PKA Encrypt</td>
<td>CSNDPKE</td>
<td>ON O, SEL</td>
<td></td>
</tr>
<tr>
<td>X'011F'</td>
<td>PKA Decrypt</td>
<td>PKA Decrypt</td>
<td>CSNDPKD</td>
<td>ON SC, SEL</td>
<td></td>
</tr>
<tr>
<td>X'0121'</td>
<td>SET Block Decompose - PIN Extension IPINENC</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0122'</td>
<td>SET Block Decompose - PIN Extension OPINENC</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0129'</td>
<td>Multiple Clear Key Import/Multiple Secure Key Import - AES</td>
<td>Multiple Clear Key Import</td>
<td>CSNBCKM</td>
<td>ON SC</td>
<td></td>
</tr>
<tr>
<td>X'012A'</td>
<td>Symmetric Algorithm Encipher - secure AES keys</td>
<td>Symmetric Algorithm Encipher</td>
<td>CSNBSAE</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'012B'</td>
<td>Symmetric Algorithm Decipher - secure AES keys</td>
<td>Symmetric Algorithm Decipher</td>
<td>CSNBSAD</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'012C'</td>
<td>Symmetric Key Generate - AES_PKCSOAEP_PKS-1.2</td>
<td>Symmetric Key Generate</td>
<td>CSNDSYG</td>
<td>ON SC</td>
<td></td>
</tr>
<tr>
<td>X'012D'</td>
<td>Symmetric Key Generate - AES_ZERO-PAD</td>
<td>Symmetric Key Generate</td>
<td>CSNDSYG</td>
<td>ON SC</td>
<td></td>
</tr>
<tr>
<td>X'012E'</td>
<td>Symmetric Key Import - AES_PKCSOAEP_PKCS-1.2</td>
<td>Symmetric Key Import</td>
<td>CSNDSYI</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'012F'</td>
<td>Symmetric Key Import - AES_ZERO-PAD</td>
<td>Symmetric Key Import</td>
<td>CSNDSYI</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0130'</td>
<td>Symmetric Key Export - AES_PKCSOAEP_PKCS-1.2</td>
<td>Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>ON SC</td>
<td></td>
</tr>
<tr>
<td>X'0131'</td>
<td>Symmetric Key Export - AES_ZERO-PAD</td>
<td>Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>ON SC</td>
<td></td>
</tr>
<tr>
<td>X'0139'</td>
<td>Symmetric token wrapping - internal enhanced method</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'013A'</td>
<td>Symmetric token wrapping - internal original method</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'013B'</td>
<td>Symmetric token wrapping - external enhanced method</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'013C'</td>
<td>Symmetric token wrapping - external original method</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'013D'</td>
<td>Diversified Key Generate - Allow wrapping override keywords</td>
<td>Diversified Key Generate</td>
<td>CSNBDKG</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'013E'</td>
<td>Symmetric Key Generate - Allow wrapping override keywords</td>
<td>Symmetric Key Generate</td>
<td>CSNDSYG</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'0140'</td>
<td>Key Part Import - Allow wrapping override keywords</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0141'</td>
<td>Multiple Clear Key Import - Allow wrapping override keywords</td>
<td>Multiple Clear Key Import</td>
<td>CSNBCKM</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0144'</td>
<td>Symmetric Key Import - Allow wrapping override keywords</td>
<td>Symmetric Key Import</td>
<td>CSNDSYI</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0146'</td>
<td>CKDS Conversion2 - Allow wrapping override keywords</td>
<td>Key Token Change</td>
<td>CSNBKTC</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0146'</td>
<td>CKDS Conversion2 - Convert from enhanced to original</td>
<td>Key Translate2</td>
<td>CSNBKTR2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0149'</td>
<td>Key Translate2</td>
<td>Key Translate2</td>
<td>CSNBKTR2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'014A'</td>
<td>Key Translate2 - Allow wrapping override keywords</td>
<td>Key Translate2</td>
<td>CSNBKTR2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'014B'</td>
<td>Key Translate2 - Allow use of REFORMAT</td>
<td>Key Translate2</td>
<td>CSNBKTR2</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'014C'</td>
<td>CKDS Conversion2 - Allow use of REFORMAT</td>
<td>Key Token Change</td>
<td>CSNBKTC</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'014D'</td>
<td>TR31 Export - Permit Version A TR-31 Key Blocks</td>
<td>Key Export to TR31</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'014E'</td>
<td>TR31 Export - Permit Version B TR-31 Key Blocks</td>
<td>Key Export to TR31</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'014F'</td>
<td>TR31 Export - Permit Version C TR-31 Key Blocks</td>
<td>Key Export to TR31</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0150'</td>
<td>TR31 Import - Permit Version A TR-31 Key Blocks</td>
<td>TR31 Key Import</td>
<td>CSNBT31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0151'</td>
<td>TR31 Import - Permit Version B TR-31 Key Blocks</td>
<td>TR31 Key Import</td>
<td>CSNBT31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0152'</td>
<td>TR31 Import - Permit Version C TR-31 Key Blocks</td>
<td>TR31 Key Import</td>
<td>CSNBT31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0153'</td>
<td>TR31 Import - Permit Override of Default Wrapping Method</td>
<td>TR31 Key Import</td>
<td>CSNBT31I</td>
<td>ON</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0154'</td>
<td>Restrict Key Attribute - Permit Setting the TR-31 Export Bit</td>
<td>Restrict Key Attribute</td>
<td>CSNBRKA</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------</td>
<td>-------------------------------</td>
<td>-------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'0155'</td>
<td>CVV Key Combine</td>
<td>CVV Key Combine</td>
<td>CSNBCKC</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0156'</td>
<td>CVV Key Combine - Allow Wrapping Override Keywords</td>
<td>CVV Key Combine(^1)</td>
<td>CSNBCKC</td>
<td>ON, DC</td>
<td>SC</td>
</tr>
<tr>
<td>X'0157'</td>
<td>CVV Key Combine - Permit Mixed Key Types</td>
<td>CVV Key Combine(^1)</td>
<td>CSNBCKC</td>
<td>ON, O, DC</td>
<td>SC</td>
</tr>
<tr>
<td>X'0158'</td>
<td>TR31 Export - Permit Any CCA Key if INCL-CV Is Specified</td>
<td>Key Export to TR31(^1)</td>
<td>CSNB31X</td>
<td>ON</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'015A'</td>
<td>TR31 Import - Permit C0 to MAC/MACVER-CVVKEY-A</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'015B'</td>
<td>TR31 Import - Permit C0 to MAC/MACVER-AMEX-CSC</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'015C'</td>
<td>TR31 Import - Permit K0:E to EXPORTER/OKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'015D'</td>
<td>TR31 Import - Permit K0:D to IMPORTER/IKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'015E'</td>
<td>TR31 Import - Permit K0:B to EXPORTER/OKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'015F'</td>
<td>TR31 Import - Permit K0:B to IMPORTER/IKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0160'</td>
<td>TR31 Import - Permit K1:E to EXPORTER/OKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0161'</td>
<td>TR31 Import - Permit K1:D to IMPORTER/IKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0162'</td>
<td>TR31 Import - Permit K1:B to EXPORTER/OKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0163'</td>
<td>TR31 Import - Permit K1:B to IMPORTER/IKEYXLAT</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0164'</td>
<td>TR31 Import - Permit M0/M1/M3 to MAC/MACVER-ANY-MAC</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0165'</td>
<td>TR31 Import - Permit P0:E to OPINENC</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0166'</td>
<td>TR31 Import - Permit P0:D to IPINENC</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0167'</td>
<td>TR31 Import - Permit V0 to PINGEN:NO-SPEC</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0168'</td>
<td>TR31 Import - Permit V0 to PINVER:NO-SPEC</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0169'</td>
<td>TR31 Import - Permit V1 to PINGEN:IBM-PIN/IBM-PINO</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'016A'</td>
<td>TR31 Import - Permit V1 to PINVER:IBM-PIN/IBM-PINO</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'016B'</td>
<td>TR31 Import - Permit V2 to PINGEN:VISA-PVV</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'016C'</td>
<td>TR31 Import - Permit V2 to PINVER:VISA-PVV</td>
<td>TR31 Key Import(^1)</td>
<td>CSNB31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'016D'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMAC</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'016E'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL0+DMV</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'016F'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMAC</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0170'</td>
<td>TR31 Import - Permit E0 to DKYGENKY:DKYL1+DMV</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0171'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL0+DMPIN</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0172'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL0+DDATA</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0173'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL1+DMPIN</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0174'</td>
<td>TR31 Import - Permit E1 to DKYGENKY:DKYL1+DDATA</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0175'</td>
<td>TR31 Import - Permit E2 to DKYGENKY:DKYL0+DMAC</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0176'</td>
<td>TR31 Import - Permit E2 to DKYGENKY:DKYL1+DMAC</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0177'</td>
<td>TR31 Import - Permit E3 to ENCIPHER</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0178'</td>
<td>TR31 Import - Permit E4 to DKYGENKY:DKYL0+DDATA</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0179'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DMAC</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'017A'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DDATA</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'017B'</td>
<td>TR31 Import - Permit E5 to DKYGENKY:DKYL0+DEXP</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'017C'</td>
<td>TR31 Import - Permit V0/V1/V2:N to PINGEN/PINVER</td>
<td>TR31 Key Import¹</td>
<td>CSNBT31I</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0180'</td>
<td>TR31 Export - Permit KEYGENKY:UKPT to B0</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0181'</td>
<td>TR31 Export - Permit MAC/MACVER:AMEX-CSC to C0:G/C/V</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0182'</td>
<td>TR31 Export - Permit MAC/MACVER:CVV-KEYA to C0:G/C/V</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0183'</td>
<td>TR31 Export - Permit MAC/MACVER:ANY-MAC to C0:G/C/V</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0184'</td>
<td>TR31 Export - Permit DATA to C0:G/C</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'0185'</td>
<td>TR31 Export - Permit ENCIPHER/DECIPHER/CIPHER to D0:E/D/B</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0186'</td>
<td>TR31 Export - Permit DATA to D0:B</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0187'</td>
<td>TR31 Export - Permit EXPORTER/OKEYXLAT to K0:E</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0188'</td>
<td>TR31 Export - Permit IMPORTER/IKEYXLAT to K0:D</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0189'</td>
<td>TR31 Export - Permit EXPORTER/OKEYXLAT to K1:E</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'018A'</td>
<td>TR31 Export - Permit IMPORTER/IKEYXLAT to K1:D</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'018B'</td>
<td>TR31 Export - Permit MAC/DATA/DATAM to M0:G/C</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'018C'</td>
<td>TR31 Export - Permit MACVER/DATAMV to M0:V</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'018D'</td>
<td>TR31 Export - Permit MAC/DATA/DATAM to M1:G/C</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'018E'</td>
<td>TR31 Export - Permit MACVER/DATAMV to M1:V</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'018F'</td>
<td>TR31 Export - Permit MAC/DATA/DATAM to M3:G/C</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0190'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC to V0</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0191'</td>
<td>TR31 Export - Permit PINENC to P0:E</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0192'</td>
<td>TR31 Export - Permit IPINENC to P0:D</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0193'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0194'</td>
<td>TR31 Export - Permit PINENC/VISA-PVV to V2</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0195'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/IBM-PIN/IBM-PINO to V1</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0196'</td>
<td>TR31 Export - Permit PINENC/IBM-PIN/IBM-PINO to V1</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0197'</td>
<td>TR31 Export - Permit PINVER:NO-SPEC/VISA-PVV to V2</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0198'</td>
<td>TR31 Export - Permit PINENC/VISA-PVV to V2</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0199'</td>
<td>TR31 Export - Permit DKYGENK/IBM-PIN/IBM-PINO to V1</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'019A'</td>
<td>TR31 Export - Permit DKYGENK/IBM-PIN/IBM-PINO to V1</td>
<td>Key Export to TR31¹</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'019B</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E0</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'019C</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E0</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'019D</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMV to E0</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'019E</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DALL to E0</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'019F</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E1</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A0</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMPIN to E1</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A1</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E1</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A2</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DDATA to E1</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A3</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMPIN to E1</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A4</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DALL to E1</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A5</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E2</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A6</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E2</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A7</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DMAC to E2</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A8</td>
<td>TR31 Export - Permit DKYGENKY:DKYL1+DALL to E2</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01A9</td>
<td>TR31 Export - Permit DATA/MAC/CIPHER/ENCIPHER to E3</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01AA</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E4</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'01AB</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E4</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'01AC</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DEXP to E5</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01AD</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DMAC to E5</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01AE</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DDATA to E5</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'01AF</td>
<td>TR31 Export - Permit DKYGENKY:DKYL0+DALL to E5</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'01B0</td>
<td>TR31 Export - Permit PINGEN/PINVER to V0/V1/V2:N</td>
<td>Key Export to TR31&lt;sup&gt;1&lt;/sup&gt;</td>
<td>CSNBT31X</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0203</td>
<td>Retained Key Delete</td>
<td>Retained Key Delete</td>
<td>CSNDRKD</td>
<td>ON</td>
<td>O, SEL</td>
</tr>
</tbody>
</table>

Appendix G. Access control points and verbs 733
<table>
<thead>
<tr>
<th>ACP number (hex)</th>
<th>Name of ACP from TKE interface</th>
<th>Verb name</th>
<th>Entry point</th>
<th>Initial setting</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0204'</td>
<td>PKA Key Generate - Clone</td>
<td>PKA Key Generate¹</td>
<td>CSNDPKG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0205'</td>
<td>PKA Key Generate - Clear</td>
<td>PKA Key Generate¹</td>
<td>CSNDPKG</td>
<td>ON, SUP</td>
<td></td>
</tr>
<tr>
<td>X'0230'</td>
<td>Retained Key List</td>
<td>Retained Key List</td>
<td>CSNDRKL</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0235'</td>
<td>Symmetric Key Import - DES_PKA92 KEK</td>
<td>Symmetric Key Import¹</td>
<td>CSNDSYI</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'023C'</td>
<td>Symmetric Key Generate - DES_ZERO-PAD</td>
<td>Symmetric Key Generate¹</td>
<td>CSNDSYG</td>
<td>ON, SC</td>
<td></td>
</tr>
<tr>
<td>X'023D'</td>
<td>Symmetric Key Import - DES_ZERO-PAD</td>
<td>Symmetric Key Import¹</td>
<td>CSNDSYI</td>
<td>ON, SC</td>
<td></td>
</tr>
<tr>
<td>X'023E'</td>
<td>Symmetric Key Export - DES_ZERO-PAD</td>
<td>Symmetric Key Export¹</td>
<td>CSNDSYX</td>
<td>ON, SC</td>
<td></td>
</tr>
<tr>
<td>X'023F'</td>
<td>Symmetric Key Generate - DES_PKCS-1.2</td>
<td>Symmetric Key Generate¹</td>
<td>CSNDSYG</td>
<td>ON, SC</td>
<td></td>
</tr>
<tr>
<td>X'0240'</td>
<td>Authorize UDX</td>
<td>Note: This ACP is included for reference only. The service impacted is available only for IBM System x, IBM System p, or using the TKE interface.</td>
<td></td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0241'</td>
<td>PKA Key Token Change RTNMK</td>
<td>PKA Key Token Change</td>
<td>CSNDKTC</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0261'</td>
<td>TKE Authorization for domain 0</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0262'</td>
<td>TKE Authorization for domain 1</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0263'</td>
<td>TKE Authorization for domain 2</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0264'</td>
<td>TKE Authorization for domain 3</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0265'</td>
<td>TKE Authorization for domain 4</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0266'</td>
<td>TKE Authorization for domain 5</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0267'</td>
<td>TKE Authorization for domain 6</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0268'</td>
<td>TKE Authorization for domain 7</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>X'0269'</td>
<td>TKE Authorization for domain 8</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'026A'</td>
<td>TKE Authorization for domain 9</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X'026B'</td>
<td>TKE Authorization for domain 10</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X'026C'</td>
<td>TKE Authorization for domain 11</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X'026D'</td>
<td>TKE Authorization for domain 12</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X'026E'</td>
<td>TKE Authorization for domain 13</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X'026F'</td>
<td>TKE Authorization for domain 14</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X'0270'</td>
<td>TKE Authorization for domain 15</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X'0273'</td>
<td>Secure Messaging for Keys</td>
<td>Secure Messaging for Keys</td>
<td>CSNBSKY</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0274'</td>
<td>Secure Messaging for PINs</td>
<td>Secure Messaging for PINs</td>
<td>CSNBSPN</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0275'</td>
<td>DATAM Key Management Control</td>
<td>Diversified Key Generate</td>
<td>CSNBDKG</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data Key Import</td>
<td>CSNBDKM</td>
<td>O, SC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data Key Export</td>
<td>CSNBDKX</td>
<td>O, SC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Key Export</td>
<td>CSNBKEX</td>
<td>O, SC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Key Generate</td>
<td>CSNBKGN</td>
<td>O, SC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Key Import</td>
<td>CSNBKIM</td>
<td>O, SC</td>
<td></td>
</tr>
<tr>
<td>X'0276'</td>
<td>Key Export - Unrestricted</td>
<td>Key Export</td>
<td>CSNBSKEX</td>
<td>ON</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0277'</td>
<td>Data Key Export - Unrestricted</td>
<td>Data Key Export</td>
<td>CSNBDKX</td>
<td>ON</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0278'</td>
<td>Key Part Import - ADD-PART</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0279'</td>
<td>Key Part Import - COMPLETE</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>ON</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'027A'</td>
<td>Key Part Import - Unrestricted</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>ON</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'027B'</td>
<td>Key Import - Unrestricted</td>
<td>Key Import</td>
<td>CSNBKIM</td>
<td>ON</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'027C'</td>
<td>Data Key Import - Unrestricted</td>
<td>Data Key Import</td>
<td>CSNBDKM</td>
<td>ON</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'027D'</td>
<td>PKA Key Generate - Permit Regeneration Data</td>
<td>PKA Key Generate</td>
<td>CSNDPKG</td>
<td>ON</td>
<td>O, NRP, SC</td>
</tr>
<tr>
<td>X'027E'</td>
<td>PKA Key Generate - Permit Regeneration Data Retain</td>
<td>PKA Key Generate</td>
<td>CSNDPKG</td>
<td>ON</td>
<td>O, NRP, SC</td>
</tr>
<tr>
<td>X'0290'</td>
<td>Diversified Key Generate - DKYGENKY - DALL</td>
<td>Diversified Key Generate</td>
<td>CSNBDKG</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIN Change/Unblock</td>
<td>CSNBPCU</td>
<td>SC, SEL</td>
<td></td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'0291'</td>
<td>Transaction Validation - Generate</td>
<td>Transaction Validation</td>
<td>CSNBTRV</td>
<td>ON, O, SEL</td>
<td></td>
</tr>
<tr>
<td>X'0292'</td>
<td>Transaction Validation - Verify CSC-3</td>
<td>Transaction Validation</td>
<td>CSNBTRV</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0293'</td>
<td>Transaction Validation - Verify CSC-4</td>
<td>Transaction Validation</td>
<td>CSNBTRV</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0294'</td>
<td>Transaction Validation - Verify CSC-5</td>
<td>Transaction Validation</td>
<td>CSNBTRV</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0295'</td>
<td>Symmetric Key Encipher/Decipher - Encrypted DES keys</td>
<td>Enables CPACF key translation for DES keys.</td>
<td>N/A</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0296'</td>
<td>Symmetric Key Encipher/Decipher - Encrypted AES keys</td>
<td>Enables CPACF key translation for AES keys.</td>
<td>N/A</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0297'</td>
<td>Key Part Import2 - Load first key part, require 3 key parts</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0298'</td>
<td>Key Part Import2 - Load first key part, require 2 key parts</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0299'</td>
<td>Key Part Import2 - Load first key part, require 1 key parts</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'029A'</td>
<td>Key Part Import2 - Add second of 3 or more key parts</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'029B'</td>
<td>Key Part Import2 - Add last required key part</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'029C'</td>
<td>Key Part Import2 - Add optional key part</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'029D'</td>
<td>Key Part Import2 - Complete key</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, SEL</td>
<td></td>
</tr>
<tr>
<td>X'029E'</td>
<td>Key Part Import2 – RETRKPR</td>
<td>Key Part Import2</td>
<td>CSNBKPI2</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0300'</td>
<td>NOCV KEK usage for export-related functions</td>
<td>Data Key Export Key Export Key Generate Remote Key Export</td>
<td>CSNBDKKX CSNBKEX CSNBKGN CSNDRKX</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0301'</td>
<td>Prohibit Export Extended</td>
<td>Prohibit Export Extended</td>
<td>CSNBPEXX</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'0303'</td>
<td>PCF CKDS conversion utility</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>ON, O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0309'</td>
<td>Key Part Import - RETRKPR</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'030A'</td>
<td>NOCV KEK usage for import-related functions</td>
<td>Data Key Import Key Import Key Generate Remote Key Export</td>
<td>CSNBDKM CSNBKIM CSNBKGN CSNDRKX</td>
<td>ON, O</td>
<td></td>
</tr>
<tr>
<td>X'030C'</td>
<td>DSG ZERO-PAD unrestricted hash length</td>
<td>Digital Signature Generate</td>
<td>CSNDDSG</td>
<td>OFF, O, SC</td>
<td></td>
</tr>
<tr>
<td>X'030F'</td>
<td>Trusted Block Create - Create a Trusted Key Block in Inactive form</td>
<td>Trusted Block Create</td>
<td>CSNDTBC</td>
<td>ON, O, SUP</td>
<td></td>
</tr>
<tr>
<td>X'0310'</td>
<td>Trusted Block Create - Activate an Inactive Trusted Key Block</td>
<td>Trusted Block Create</td>
<td>CSNDTBC</td>
<td>ON, O, SUP</td>
<td></td>
</tr>
<tr>
<td>X'0311'</td>
<td>PKA Key Import - Import an External Trusted Key Block to internal form</td>
<td>PKA Key Import</td>
<td>CSNDPKI</td>
<td>ON, O, SEL</td>
<td></td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>X'0312'</td>
<td>Remote Key Export - Generate or export a key for use by a non-CCA node</td>
<td>Remote Key Export</td>
<td>CSNDRKX</td>
<td>ON</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0313'</td>
<td>PTR Enhanced PIN Security</td>
<td>Clear PIN Generate</td>
<td>CSNBCPA</td>
<td>OFF</td>
<td>O, SC, SEL</td>
</tr>
<tr>
<td>X'0318'</td>
<td>PKA Key Translate - from CCA RSA to SC Visa Format</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0319'</td>
<td>PKA Key Translate - from CCA RSA to SC ME Format</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'031A'</td>
<td>PKA Key Translate - from CCA RSA to SC CRT Format</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'031B'</td>
<td>PKA Key Translate - from source EXP KEK to target EXP KEK</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'031C'</td>
<td>PKA Key Translate - from source IMP KEK to target EXP KEK</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'031D'</td>
<td>PKA Key Translate - from source IMP KEK to target IMP KEK</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>ON</td>
<td>O</td>
</tr>
<tr>
<td>X'0327'</td>
<td>Symmetric Key Export - AESKW</td>
<td>Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>ON</td>
<td>O, R</td>
</tr>
<tr>
<td>X'0328'</td>
<td>Variable-length Symmetric Token - disallow weak wrap</td>
<td>EC Diffie-Hellman(^1) Key Generate(^2) PKA Key Generate(^1) Symmetric Key Export(^1)</td>
<td>CSNDEDH CSNGBKGN2 CSNDPKG CSNDSYX</td>
<td>OFF</td>
<td>O, R</td>
</tr>
<tr>
<td>X'0329'</td>
<td>Symmetric Key Import2 - AESKW</td>
<td>Symmetric Key Import2(^1)</td>
<td>CSNDSYI2</td>
<td>ON</td>
<td>O, R</td>
</tr>
<tr>
<td>X'032A'</td>
<td>Key Translate2 - Disallow AES ver 5 to ver 4 conversion</td>
<td>Key Translate2(^1)</td>
<td>CSNBKTR2</td>
<td>OFF</td>
<td>O, R</td>
</tr>
<tr>
<td>X'032B'</td>
<td>Symmetric Key Import2 - disallow weak import</td>
<td>Symmetric Key Import2(^1)</td>
<td>CSNDSYI2</td>
<td>OFF</td>
<td>O, R</td>
</tr>
<tr>
<td>X'032C'</td>
<td>Variable-length Symmetric Token - warn when weak wrap</td>
<td>EC Diffie-Hellman(^1) Key Generate(^2) Symmetric Key Export(^1)</td>
<td>CSNDEDH CSNGBKGN2 CSNDSYX CSNDSYI2J</td>
<td>OFF</td>
<td>O, R</td>
</tr>
<tr>
<td>X'0350'</td>
<td>ANSI X9.8 PIN - Enforce PIN block restrictions</td>
<td>Clear PIN Generate Alternate Encrypted PIN Translate Secure Messaging for PINs</td>
<td>CSNBCPA CSNBPTR CSNBSPPN</td>
<td>OFF</td>
<td>O, R</td>
</tr>
<tr>
<td>X'0351'</td>
<td>ANSI X9.8 PIN - Allow modification of PAN_01_0350</td>
<td>Encrypted PIN Translate Secure Messaging for PINs</td>
<td>CSNBPTR CSNBSPPN</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0352'</td>
<td>ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350</td>
<td>Encrypted PIN Translate Secure Messaging for PINs</td>
<td>CSNBPTR CSNBSPPN</td>
<td>OFF</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0353'</td>
<td>Allow load of Decimalization Table</td>
<td></td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
</tbody>
</table>

Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.
<table>
<thead>
<tr>
<th>ACP number (hex)</th>
<th>Name of ACP from TKE interface</th>
<th>Verb name</th>
<th>Entry point</th>
<th>Initial setting</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0354'</td>
<td>Allow delete of Decimalization Table</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0355'</td>
<td>Allow activation of Decimalization Table</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'0356'</td>
<td>ANSI X9.8 PIN - Use stored decimalization table only</td>
<td>Clear PIN Generate1</td>
<td>CSNBPGN CSNBCPA CSNBEPCG CSNBPVVR</td>
<td>OFF O, R</td>
<td></td>
</tr>
<tr>
<td>X'0360'</td>
<td>ECC Diffie-Hellmann Callable Services</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0361'</td>
<td>EC Diffie-Hellman - Allow PASSTHRU</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0362'</td>
<td>ECC Diffie-Hellmann - Allow Configuration Override with Keyword in EDH</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0363'</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 192</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0364'</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 224</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0365'</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 256</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0366'</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 384</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0367'</td>
<td>ECC Diffie-Hellmann - Allow Prime Curve 521</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0368'</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 160</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'0369'</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 192</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'036A'</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 224</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'036B'</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 256</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'036C'</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 320</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'036D'</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 384</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'036E'</td>
<td>ECC Diffie-Hellmann - Allow Brainpool Curve 512</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>ON O</td>
<td></td>
</tr>
<tr>
<td>X'036F'</td>
<td>Prevent Weaker Keys from Being Used to Generate Stronger Keys</td>
<td>EC Diffie-Hellman1</td>
<td>CSNDEDH</td>
<td>OFF O</td>
<td></td>
</tr>
<tr>
<td>X'0370'</td>
<td>Allow loading the first key part via CSNBKTPIT for a key that has minimum of 3 key parts to complete</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td>OFF O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACP number (hex)</td>
<td>Name of ACP from TKE interface</td>
<td>Verb name</td>
<td>Entry point</td>
<td>Initial setting</td>
<td>Usage</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td><code>X'0371'</code></td>
<td>Allow loading the first key part via CSNBKPIT for a key that has minimum of 2 key parts to complete</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td><code>X'0372'</code></td>
<td>Allow loading the first key part via CSNBKPIT for a key that has minimum of 1 key part to complete</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td><code>X'0373'</code></td>
<td>Allow loading the second and later key part via CSNBKPIT for a key that has requires more key parts to complete</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td><code>X'0374'</code></td>
<td>Allow loading the last key part via CSNBKPIT for a key</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td><code>X'0375'</code></td>
<td>Allow loading an optional key part for a key via CSNBKPIT</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td><code>X'0376'</code></td>
<td>Allow completing a key that has had all key parts loaded via CSNBKPIT</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
<tr>
<td><code>X'0377'</code></td>
<td>Allow clearing a key part register as managed by CSNBKPIT</td>
<td>Note: This ACP is included for TKE reference only, the service impacted is available only (for IBM System z) on z/OS.</td>
<td></td>
<td>OFF</td>
<td>O</td>
</tr>
</tbody>
</table>

The following codes are used in this table:

ID Initial default.
O Usage of this command is optional; enable it as required for authorized usage.
R Enabling this command is recommended.
NR Enabling this command is not recommended.
NRP Enabling this command is not recommended for production.
SC Usage of this command requires special consideration.
SEL Usage of this command is normally restricted to one or more selected roles.
SUP This command is normally restricted to one or more supervisory roles.

1 This verb performs more than one function, as determined by the keyword in the rule_array parameter of the verb call. Not all functions of the verb require the command in this row.
2 This verb does not always require the command in this row. Use as determined by the control vector for the key and the action being performed.

**TKE Version 6.0 and higher**

The TKE workstation allows you to enable or disable verb access control points.

For systems that do not use the optional TKE workstation, most access control points (current and new) are enabled in the DEFAULT Role with the appropriate licensed internal code on the CEX*C. For more information about the TKE workstation, see z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User’s Guide.
You must have a TKE V6.0 or higher workstation in order to see supported CEX*C features. They are not seen when using earlier TKE workstations. TKE Version 4.0 through TKE Version 6.0 can access CEX2C features.

Use of particular cryptographic or key management verb functions with the CEX*C are controlled through access control points. You can see the default settings of an access control point in Table 206 on page 724 as “Initial setting”.

Note:
1. Access control points DKGNYN-DALL and DSG ZERO-PAD unrestricted hash length are always disabled in the DEFAULT role for all customers (TKE and non-TKE). A TKE workstation is required to enable these access control points.
2. When you modify the setting of an access control point, please be sure to use a procedure according to your organization’s security policy. TKE workstation versions earlier than V6.0 do not show the current setting of the access control points. TKE workstation versions 6.0 and higher show the current setting, but neither show the default settings nor a change history of the listed access control points. If you do not remember the change history, note that using the **Zeroize** function of the card or of the domain in order to reset all access control point settings to their default values discards all keys.

For CCA Release 4.2.0, there are new access control points for these verbs:
- Clear PIN Generate (CSNBPGN) - ANSI X9.8 PIN - Use stored decimmalization table only (offset X'0356')
- Clear PIN Generate Alternate (CSNBCPA) - ANSI X9.8 PIN - Use stored decimmalization table only (offset X'0356')
- CVV Key Combine (CSNBCKC)
  - CVV Key Combine (offset X'0155')
  - CVV Key Combine - Allow Wrapping Override Keywords (offset X'0156')
  - CVV Key Combine - Permit Mixed Key Types command (offset X'0157')
- EC Diffie-Hellman (CSNDEDH)
  - Disallow Weak Key Wrap (offset X'0328')
  - Warn When Wrapping Weak Keys (offset X'032C')
  - ECC Diffie-Hellman Callable Services (offset X'0360')
  - EC Diffie-Hellman - Allow PASSTHRU (offset X'0361')
  - ECC Diffie-Hellman - Allow Configuration Override with Keyword in EDH command (offset X'0362')
  - ECC Diffie-Hellman - Allow Prime Curve 192 (offset X'0363')
  - ECC Diffie-Hellman - Allow Prime Curve 224 (offset X'0364')
  - ECC Diffie-Hellman - Allow Prime Curve 256 (offset X'0365')
  - ECC Diffie-Hellman - Allow Prime Curve 384 (offset X'0366')
  - ECC Diffie-Hellman - Allow Prime Curve 521 (offset X'0367')
  - ECC Diffie-Hellman - Allow Brainpool Curve 160 (offset X'0368')
  - ECC Diffie-Hellman - Allow Brainpool Curve 192 (offset X'0369')
  - ECC Diffie-Hellman - Allow Brainpool Curve 224 (offset X'036A')
  - ECC Diffie-Hellman - Allow Brainpool Curve 256 (offset X'036B')
  - ECC Diffie-Hellman - Allow Brainpool Curve 320 (offset X'036C')
  - ECC Diffie-Hellman - Allow Brainpool Curve 384 (offset X'036D')
ECC Diffie-Hellmann - Allow Brainpool Curve 512 (offset X'036E')
- Prevent Weaker Keys from Being Used to Generate Stronger Keys (offset X'036F')

- Encrypted PIN Generate (CSNBEPG) - ANSI X9.8 PIN - Use stored decimalization table only (offset X'0356')
- Encrypted PIN Verify (CSNPBVR) - ANSI X9.8 PIN - Use stored decimalization table only (offset X'0356')
- Key Export to TR31 (CSNBT31X)
  - All access control points whose name starts with 'TR31 Export'. See Table 206 on page 724
- Key Generate2 (CSNBKGN2)
  - Disallow Weak Key Wrap (offset X'0328')
  - Warn When Wrapping Weak Keys (offset X'032C')
- Key Part Import2 (CSNBKPI2) - Key Part Import2 – RETRKPR (offset X'029E')
- Key Test2 (CSNBKYT2) - Compute ENC-ZERO Verification Pattern for AES command (offset X'0021')
- Key Translate2 (CSNBKTR2) - Key Translate2 - Disallow AES version 5 to version 4 conversion (offset X'032A')
- PKA Key Generate (CSNDPKG)
  - PKA Clone Key Generate (offset X'0204')
  - Disallow Weak Key Wrap (offset X'0328')
  - Warn When Wrapping Weak Keys (offset X'032C')
- Symmetric Key Export (CSNDSYX)
  - Symmetric Key Export - HMAC_PKCSOAEP (offset X'00F5')
  - Symmetric Key Export - AES, PKOAEP2 (offset X'00FC')
  - Symmetric Key Export - DES, PKCS-1.2 (offset X'0105')
- Symmetric Key Generate (CSNDSYG)
  - Symmetric Key Generate - DES, PKA92 (offset X'010D')
  - Symmetric Key Generate - AES, PKCSOAEP, PKCS-1.2 (offset X'012C')
  - Symmetric Key Generate - AES, ZERO-PAD (offset X'0131')
  - Symmetric Key Generate - DES, ZERO-PAD (offset X'023E')
- Symmetric Key Import2 (CSNDSYI2)
  - Import HMAC Key (PKOAEP2) (offset X'00F4')
  - Import AES Key (PKOAEP2 (offset X'00FD')
  - Import AES Key (AESKW) (offset X'0329')
  - Disallow Weak Import command (offset X'032B')
  - Warn When Wrapping Weak Keys command (offset X'032C')
- TR31 Key Import (CSNBT31I)
All access control points whose name starts with 'TR31 Import’. See Table 206 on page 724.

For CCA Release 4.1.0, there are new access control points for these verbs:

- **Clear PIN Generate Alternate (CSNCBPA)** - Enforce ANSI X9.8 PIN Rules (offset X'0350')
- **Encrypted PIN Translate (CSNBPTR)**
  - ANSI X9.8 PIN - Enforce PIN block restrictions (offset X'0350')
  - ANSI X9.8 PIN - Allow modification of PAN_01_0350 (offset X'0351')
  - ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350 (offset X'0352')
- **Secure Messaging for PINs (CSNBSNP)**
  - ANSI X9.8 PIN - Enforce PIN block restrictions (offset X'0350')
  - ANSI X9.8 PIN - Allow modification of PAN_01_0350 (offset X'0351')
  - ANSI X9.8 PIN - Allow only ANSI PIN blocks_01_0350 (offset X'0352')
Appendix H. Sample verb call routines

This appendix contains sample verb call routines for both C and Java.

**IMPORTANT**
The user must load the Symmetric Master Key before the verb calls will complete successfully, otherwise return code 12 reason code 764 will be returned.

To illustrate the practical application of CCA verb calls, this appendix describes the sample routines included with the RPM. A sample in C, and one in Java is included.

The sample routines generate a Message Authentication Code (MAC) on a text string, and then verifies the MAC. To accomplish this, the routine:

- Calls the Key Generate (CSNBKGN or CSNBKGNJ) verb to create a MAC/MACVER key pair.
- Calls the MAC Generate (CSNBMGN or CSNBMGNJ) verb to generate a MAC on a text string with the MAC key.
- Calls the MAC Verify (CSNBMVR or CSNBMVRJ) verb to verify the text string MAC with the MACVER key.

As you review the sample routines shown in [Figure 34](#) and [Figure 35 on page 748](#), refer to the chapters in this book for descriptions of the called verbs and their parameters. These verbs are listed in Table 207.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Entry point name for C and Java versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Generate</td>
<td>CSNBKGN or CSNBKGNJ</td>
</tr>
<tr>
<td>MAC Generate</td>
<td>CSNBMGN or CSNBMGNJ</td>
</tr>
<tr>
<td>MAC Verify</td>
<td>CSNBMVR or CSNBMVRJ</td>
</tr>
</tbody>
</table>

**Sample program in C**

This sample code, which consists of a C program (mac.c) and a makefile (makefile.Inx), can be found in the /opt/IBM/CCA/samples directory.

For reference, a copy of the sample routine is shown in [Figure 34](#).

**Figure 34. Syntax, sample routine in C**

```c
/* ***********************************************
/* Module Name: mac.c
/* DESCRIPTIVE NAME: Cryptographic Coprocessor Support Program
/* C language source code example
/* Licensed Materials - Property of IBM
```
This example program:

1) Calls the Key_Generate verb (CSNBKGN) to create a MAC (message authentication code) key token and a MACVER key token.

2) Calls the MAC_Generate verb (CSNBMGN) using the MAC key token from step 1 to generate a MAC on the supplied text string (INPUT_TEXT).

3) Calls the MAC_Verify verb (CSNBMVR) to verify the MAC for the same text string, using the MACVER key token created in step 1.

#include <stdio.h>
#include <string.h>
#ifdef _AIX
#include <csufincl.h>
#endif
#elif __WINDOWS__
#include "csunincl.h"
#else
#include "csulincl.h" /* else linux */
#endif

/* Defines */
#define KEY_FORM "OPOP"
#define KEY_LENGTH "SINGLE "
#define KEY_TYPE_1 "MAC "
#define KEY_TYPE_2 "MACVER "
#define INPUT_TEXT "abcdefhgijklmn0987654321"
#define MAC_PROCESSING_RULE "X9.9-1 "
#define SEGMENT_FLAG "ONLY "
#define MAC_LENGTH "HEX-9 "
#define MAC_BUFFER_LENGTH 10

void main()
{
    static long   return_code;
    static long   reason_code;
    static unsigned char key_form[4];
    static unsigned char key_length[8];
    static unsigned char mac_key_type[8];
    static unsigned char macver_key_type[8];
    static unsigned char kek_key_id_1[64];
    static unsigned char kek_key_id_2[64];
    static unsigned char mac_key_id[64];
    static unsigned char macver_key_id[64];
    static long    text_length;
    static unsigned char text[26];
    static long    rule_array_count;
    static unsigned char rule_array[3][8]; /* Max 3 rule array elements */
    static unsigned char chaining_vector[18];
    static unsigned char mac_value[MAC_BUFFER_LENGTH];

    /* Print a banner */
    printf("Cryptographic Coprocessor Support Program example program.\n");
    /* Set up initial values for Key_Generate call */
    return_code = 0;
    reason_code = 0;
    memcpy (key_form, KEY_FORM, 4); /* OPOP key pair */
    memcpy (key_length, KEY_LENGTH, 8); /* Single-length keys */
    memcpy (mac_key_type, KEY_TYPE_1, 8); /* 1st token, MAC key type */
    memcpy (macver_key_type, KEY_TYPE_2, 8); /* 2nd token, MACVER key type */
    memset (kek_key_id_1, 0x00, sizeof(kek_key_id_1)); /* 1st KEK not used */
    memset (kek_key_id_2, 0x00, sizeof(kek_key_id_2)); /* 2nd KEK not used */
    memset (mac_key_id, 0x00, sizeof(mac_key_id)); /* Init 1st key token */
    memset (macver_key_id, 0x00, sizeof(macver_key_id)); /* Init 2nd key token */

    /* Generate a MAC/MACVER operational key pair */
    CSNGBKGN(return_code,
             NULL, /* exit_data_length */
             NULL, /* exit_data */
             key_form, key_length, mac_key_type, macver_key_type,
             kek_key_id_1, kek_key_id_2, mac_key_id, macver_key_id);

    /* Check the return/reason codes. Terminate if there is an error. */
    if (return_code != 0 || reason_code != 0) {
        printf("Key_Generate failed: "); /* Print failing verb */
        Appendix H. Sample verb call routines 745
printf ("return_code = %ld, ", return_code); /* Print return code */
printf ("reason_code = %ld\n", reason_code); /* Print reason code */
return;
}
else
    printf ("Key_Generate successful.\n"));

/* Set up initial values for MAC_Generate call */
return_code = 0;
reason_code = 0;
text_length = sizeof (INPUT_TEXT) - 1; /* Length of MAC text */
memcpy (text, INPUT_TEXT, text_length); /* Define MAC input text */
rule_array_count = 3; /* 3 rule array elements */
memset (rule_array, ' ', sizeof(rule_array)); /* Clear rule array */
memcpy (rule_array[0], MAC_PROCESSING_RULE, 8); /* 1st rule array element */
memcpy (rule_array[1], SEGMENT_FLAG, 8); /* 2nd rule array element */
memcpy (rule_array[2], MAC_LENGTH, 8); /* 3rd rule array element */
memset (chaining_vector, 0x00, 18); /* Clear chaining vector */
memset (mac_value, 0x00, sizeof(mac_value)); /* Clear MAC value */

/* Generate a MAC based on input text */
CSNBMGN ( &return_code,
    &reason_code,
    NULL, /* exit_data_length */
    NULL, /* exit_data */
    mac_key_id, /* Output from Key_Generate */
    &text_length,
    text,
    &rule_array_count,
    &rule_array[0][0],
    chaining_vector,
    mac_value);

/* Check the return/reason codes. Terminate if there is an error. */
if (return_code != 0 || reason_code != 0) {
    printf ("MAC Generate Failed: "); /* Print failing verb */
    printf ("return_code = %ld, ", return_code); /* Print return code */
    printf ("reason_code = %ld\n", reason_code); /* Print reason code */
    return;
}
else {
    printf ("MAC_Generate successful.\n");
    printf ("MAC_value = %s\n", mac_value); /* Print MAC value (HEX-9) */
}

/* Set up initial values for MAC_Verify call */
return_code = 0;
reason_code = 0;
rule_array_count = 1; /* 1 rule array element */
memset (rule_array, ' ', sizeof(rule_array)); /* Clear rule array */
memcpy (rule_array[0], MAC_LENGTH, 8); /* Rule array element */
    /* (use default Ciphering */
    /* and Segmenting */
    /* Control) */
memset (chaining_vector, 0x00, 18); /* Clear the chaining vector */

/* Verify MAC value */
CSNBMVR ( &return_code,
    &reason_code,
    NULL, /* exit_data_length */
    NULL, /* exit_data */
    macver_key_id, /* Output from Key_Generate */
    &text_length,
    text,
    &rule_array_count,
    &rule_array[0][0],
    chaining_vector,
    mac_value); /* Output from MAC_Generate */
Sample program in Java

Before running this program, review the information about the JNI interface.

You can find this information in “Building Java applications to use with the CCA JNI” on page 26.

This sample code consists of a Java program named mac.java. For reference, a copy of the sample routine is shown in Figure 35 on page 748. Another sample program named RNG.java is included with the distribution at the same location, but is not copied here because it is a very simple JNI reference exercise to call the Random Number Generate verb.

The default distribution location of the sample code is:

**SUSE Linux**

/opt/IBM/CCA/samples

**Red Hat Linux**

/opt/IBM/CCA/samples

Invoke the following command from the directory that contains the sample source code to compile the program:

```
javac -classpath /opt/IBM/CCA/cnm/HIKM.zip: mac.java
```

**Note:**

1. The classpath option points to the HIKM.zip file because the hikmNativeInteger class is in this file.
2. The path shown for the HIKM.zip file is the default distribution location of that file.

When it is compiled, you can run the sample Java program from the directory that contains the compiled output, with these commands.

**For a Red Hat Linux system:**

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64
/export/ibm/java-1386-60/jre/bin/java -classpath /opt/IBM/CCA/cnm/HIKM.zip: mac
```

**For a SUSE Linux system:**

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64
java -classpath /opt/IBM/CCA/cnm/HIKM.zip: mac
```

**Note:**

1. The path shown for the HIKM.zip file is the default distribution location of that file.
2. The *libcsulcca.so* library for Linux also contains the C support for the CCA Java Native Interface (JNI).

Figure 35. Syntax, sample routine in Java

```
/* Module Name: mac.java */
/* DESCRIPTIVE NAME: Cryptographic Coprocessor Support Program */
/* JNI example code */

/* Licensed Materials - Property of IBM */
/* Copyright IBM Corp. 2010 All Rights Reserved */
/* US Government Users Restricted Rights - Use duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp. */

NOTICE TO USERS OF THE SOURCE CODE EXAMPLES

The source code examples provided by IBM are only intended to assist in the development of a working software program. The source code examples do not function as written: additional code is required. In addition, the source code examples may not compile and/or bind successfully as written.

International Business Machines Corporation provides the source code examples, both individually and as one or more groups, "as is" without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the source code examples, both individually and as one or more groups, is with you. Should any part of the source code examples prove defective, you (and not IBM or an authorized dealer) assume the entire cost of all necessary servicing, repair or correction.

IBM does not warrant that the contents of the source code examples, whether individually or as one or more groups, will meet your requirements or that the source code examples are error-free.

IBM may make improvements and/or changes in the source code examples at any time.

Changes may be made periodically to the information in the source code examples; these changes may be reported, for the sample code included herein, in new editions of the examples.

References in the source code examples to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to the IBM licensed program in the source code examples is not intended to state or imply that IBM's licensed program must be used. Any functionally equivalent program may be used.

This example program:

1) Calls the Key_Generate verb (CSNBKGNJ) to create a MAC (message*
import java.io.*;  
public class mac  
{
  static final String KEY_FORM = "OPOP";
  static final String KEY_LENGTH = "SINGLE ";
  static final String KEY_TYPE_1 = "MAC ";
  static final String KEY_TYPE_2 = "MACVER ";
  static final String INPUT_TEXT = "abcdefgijkmnopqrstuvwxyz";
  static final String MAC_PROCESSING_RULE = "X9.9-1 ";
  static final String SEGMENT_FLAG = "ONLY ";
  static final String MAC_LENGTH = "HEX-9 ";
  public static void main (String args[])  
  {
    byte [] ByteExitData = new byte [4];
    byte [] Byte_key_form = new byte [4];
    byte [] Byte_key_length = new byte [8];
    byte [] Byte_mac_key_type = new byte [8];
    byte [] Byte_macver_key_type = new byte [8];
    byte [] Byte_mac_value = new byte [10];
    byte [] Byte_chaining_vector = new byte [18];
    byte [] Byte_rule_array = new byte [24];
    byte [] Byte_text = new byte [26];
    byte [] Byte_kek_key_id_1 = new byte [64];
    byte [] Byte_kek_key_id_2 = new byte [64];
    byte [] Byte_mac_key_id = new byte [64];
    byte [] Byte_macver_key_id = new byte [64];
    try
    {
      //setup to pause on non-zero return/reason code
      //and require enter key to continue
      BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
      hikmNativeInteger IntReturncode = new hikmNativeInteger(0);
      hikmNativeInteger IntReasoncode = new hikmNativeInteger(0);
      hikmNativeInteger IntExitDataLength = new hikmNativeInteger(0);
      /* Print beginning banner */
      System.out.println("Cryptographic Coprocessor Support Program JAVA example program.\n");
      /* Set up initial values for Key_Generate call */
      Byte_key_form = new String(KEY_FORM).getBytes();
      /* OPOP key pair */
      Byte_key_length = new String(KEY_LENGTH).getBytes();
      /* Single-length keys */
      Byte_mac_key_type = new String(KEY_TYPE_1).getBytes();
      /* 1st token, MAC key type */
      Byte_macver_key_type = new String(KEY_TYPE_2).getBytes();
      /* 2nd token, MACVER key type */
      /* Generate a MAC/MACVER operational key pair */
      new HIKM().CSNBKGNJ (IntReturncode,
                         IntReasoncode,
                         IntExitDataLength,
                         ByteExitData,
                         Byte_key_form,
if ( 0 != IntReturncode.getValue() || 0 != IntReasoncode.getValue() ) {
    System.out.println("nKey Generate Failed");
    /* Print failing verb. */
    System.out.println("Return_code = " + IntReturncode.getValue());
    /* Print return code. */
    System.out.println("Reason_code = " + IntReasoncode.getValue());
    /* Print reason code. */
    System.out.println("Press ENTER to continue...");
    /* Print Pause message */
    stdin.readLine();
} else {
    System.out.println("Key_Generate successful.");
}

/* Set up initial values for MAC_Generate call */
IntReturncode = new hikmNativeInteger(0);
IntReasoncode = new hikmNativeInteger(0);
IntExitDataLength = new hikmNativeInteger(0);

hikmNativeInteger Int_rule_array_count = new hikmNativeInteger(3);
hikmNativeInteger Int_text_length = new hikmNativeInteger(24);

byte [] temp_array = new String (INPUT_TEXT).getBytes();
/*Define MAC input text */
byte [] temp_array = new String (MAC_PROCESSING_RULE).getBytes();
/*1st rule array element*/
System.arraycopy(temp_array, 0, Byte_rule_array, 0, temp_array.length);
/*1st rule array element*/
temp_array = new String(SEGMENT_FLAG).getBytes();
/*2nd rule array element*/
System.arraycopy(temp_array, 0, Byte_rule_array, 8, temp_array.length);
/*2nd rule array element*/
temp_array = new String(MAC_LENGTH).getBytes();
/*3rd rule array element*/
System.arraycopy(temp_array, 0, Byte_rule_array, 16, temp_array.length);
/*3rd rule array element*/

/* Generate a MAC based on input text */
new HIKM().CSNBMGNJ (IntReturncode,
IntReasoncode,
IntExitDataLength,
ByteExitData,
Byte_mac_key_id,
Int_text_length,
Byte_text,
Int_rule_array_count,
Byte_rule_array,
Byte_chaining_vector,
Byte_mac_value);

if ( 0 != IntReturncode.getValue() || 0 != IntReasoncode.getValue() ) {
    System.out.println("nMAC Generate Failed");
    /* Print failing verb. */
    System.out.println("Return_code = " + IntReturncode.getValue());
    /* Print return code. */
System.out.println("Reason_code = " + IntReasoncode.getValue());
/* Print reason code. */
System.out.println("Press ENTER to continue...");
/* Print Pause message */
stdin.readLine();
} else {
    System.out.println("MAC_Generate successful.");
    System.out.println("MAC_value = [" + new String(Byte_mac_value) + "]");
}
/* Set up initial values for MAC_Verify call */
IntReturncode = new hikmNativeInteger(0);
IntReasoncode = new hikmNativeInteger(0);
IntExitDataLength = new hikmNativeInteger(0);
Byte_rule_array = new String(MAC_LENGTH).getBytes();
/* Rule array element */
Int_rule_array_count = new hikmNativeInteger(1);
new HIKM().CSNBMVRJ (IntReturncode,
IntReasoncode,
IntExitDataLength,
ByteExitData,
Byte_macver_key_id,
Int_text_length,
Byte_text,
Int_rule_array_count,
Byte_rule_array,
Byte_chaining_vector,
Byte_mac_value);
if (0 != IntReturncode.getValue() || 0 != IntReasoncode.getValue() ) {
    System.out.println("MAC_Verify Failed");
    /* Print failing verb. */
    System.out.println("Return_code = " + IntReturncode.getValue());
    /* Print return code. */
    System.out.println("Reason_code = " + IntReasoncode.getValue());
    /* Print reason code. */
    System.out.println("Press ENTER to continue...");
    /* Print Pause message */
    stdin.readLine();
} else {
    System.out.println("MAC_Verify successful.");
}
} catch (Exception anException) {
    System.out.println(anException);
}
/* Print ending banner */
System.out.println("Cryptographic Coprocessor Support Program JAVA example program finished.
");
//end main
//end mac class
Appendix I. Initial system set up tips

This appendix contains tips to help you set up your system for the first time.

The name of the CCA 4.2.10 RPM is: csulcca-4.2.10-
maintenance_level.s390x.rpm, where maintenance_level is a number representing
the current maintenance level.

In order to use the full set of CCA Release 4.2.0 or later functions, a CCA feature is
required. This feature must have a CCA code level of 4.2.0 or later for RPM
platform target s390x. Support for the CCA feature became available for System
z10™ in March 2010. A limited set of CCA Release 4.2, 4.1.0, and 4.0.0 functions can
be used with a CEX2C feature.

Consult the README.linz file in the /opt/IBM/CCA/doc/ directory for this
information:
- Release-specific information, if there is any
- Pointers to helpful tools

Installing and loading the cryptographic device driver

The cryptographic device driver z90crypt is included in the regular kernel package
shipped with your Linux distribution.

The cryptographic device driver is provided as a single kernel module z90crypt.

Use the lsmod command to find out if this module is already loaded.

If required, use the modprobe command to load the module. When loading the
device driver module, you can use the following optional module parameters:

\texttt{domain=}

specifies a particular cryptographic domain. By default, the device driver
attempts to use the domain with the maximum number of devices. To use all
CCA Release 4.2.0 functions, the domain must include at least one CEX3C or
above feature.

After loading the device driver, use the lszcrypt command with the -b option
to confirm that the correct domain is used. If your distribution does not
include this command, see the version of \textit{Device Drivers, Features, and
Commands} that applies to your distribution about how to use the sysfs interface
to find out the domain. This publication also provides more information about
loading and configuring the cryptographic device driver.

To change the domain, you must unload the z90crypt module (see \textit{Unloading
the cryptographic device driver} on page 754) and reload it.

\texttt{poll_thread=}

enables the polling thread for instances of Linux on z/VM and for Linux
instances that run in LPAR mode on an IBM mainframe system earlier than
System z10.

For Linux instances that run in LPAR mode on a System z10 or later
mainframe, this setting is ignored and AP interrupts are used instead.
For more information about these module parameters, the polling thread, and AP interrupts, see the version of Device Drivers, Features, and Commands that applies to your distribution.

### Unloading the cryptographic device driver

You might need to unload the cryptographic device driver, for example, to change the domain setting.

Unloading the device driver is complicated slightly because the catcher.exe daemon is always running, to be ready to receive TKE requests. To unload the device driver, for example, in preparation for a reload, you must stop the catcher.exe daemon. This can be done with the service management script /etc/init.d/CSUTKEcat using the stop argument, or by using the ps command to find the PID for the daemon, and then using the kill -9 <PID> command to kill it.

After reloading the device driver, you can restart the catcher.exe daemon (restarting TKE access) using the /etc/init.d/CSUTKEcat start command. See “Files in the RPM” on page 758 for more details.

### Confirming your cryptographic devices

Use the lszcrypt command with the -V option to display a list of available cryptographic devices and their online status.

```
lszcrypt -V
```

```
card00: CEX2A online
card01: CEX2A online
card02: CEX3C offline
card03: CEX3A online
```

In the example, device card02 is offline. You can use the chzcrypt command to set a cryptographic device online. For example, to set a cryptographic device card02 online, issue:

```
chzcrypt -e 2
```

In the command, 2 is the decimal representation of the hexadecimal 02 index in the device name, card02.

For more details about the lszcrypt and chzcrypt commands, see the man pages. If your distribution does not include these commands, see the version of Device Drivers, Features, and Commands that applies to your distribution about how to use the sysfs interface to find out this information and to set devices online.

### Checking the adapter settings

Use the lszcrypt command with the -b option to display information about the settings for your cryptographic adapters.

```
lszcrypt -b
```

```
ap_domain=8
ap_interrups are enabled
config_time=30 (seconds)
poll_thread is disabled
poll_timeout=250000 (nanoseconds)
```

As seen in the example, the command output shows:

- The domain used

Note: To change the domain you must unload the z90crypt module and reload it with the appropriate domain= parameter.

- Whether AP interrupts are used
- Whether the polling thread is used and, if so, at which frequency

Note: Polling threads cannot be enabled if AP interrupts are available.

If your distribution does not include the `lszcrypt` command, see the version of Device Drivers, Features, and Commands that applies to your distribution about how use the sysfs interface to retrieve this information.

### Performance tuning

If no AP interrupts are available to your Linux instance, you can use the settings for the polling thread and the high resolution polling timer to tune the performance of your cryptographic adapters.

See the version of Device Drivers, Features, and Commands that applies to your distribution for more information about these settings.

### Running secure key under a z/VM guest

In order to use the CEX*C feature under z/VM versions 6.1, and 5.4, you need to apply specific APAR fixes.

These fixes are described in “Hardware requirements” on page xxi.

Also, to get secure key running under a z/VM guest, a directory control statement (CRYPTO APDED) for a given z/VM guest needs to be used. This requires that the AP’s with this domain are owned by the LPAR. There is no virtualization done by z/VM.

For secure key, z/VM does not virtualize the AP’s. The AP’s need to be dedicated, which is done by the user statement:

```
CRYPTO DOMAIN 12 APDED 5 7
```

This statement dedicates AP’s 5 and 7 for domain 12 to one Linux guest.

Note: Shared crypto adapters, as defined with the z/VM user directory statement CRYPTO APVIRT, cannot be used for secure key cryptographic operations. Because dedicated and shared cryptographic adapters cannot be mixed in a z/VM guest virtual machine, additional crypto adapters for use with clear key cryptography, coprocessors or accelerators, must be defined as dedicated adapters.
Appendix J. CCA installation instructions

This appendix contains CAA Release 4.0.0 and later installation, configuration, and uninstallation instructions.

Before you begin

In this section, specific terms are used to describe CCA RPMs, referring to several different versions.

About this task

CCA 4.x

CCA 4.0.0, CCA 4.1.0, CCA 4.2.0, and subsequent versions.

CCA 3.x

Any CCA version 3 release.

Before you begin the CCA installation, review these points:

- If you are upgrading from a prior installation, be sure to review "Default installation directory."
- Ensure that you are using supported hardware. See "Hardware requirements" on page xxi.
- Ensure that your Linux distribution has the zcrypt device driver support. For details, see "Installing and loading the cryptographic device driver" on page 753.
- If you are going to use the Java Native Interface (JNI), see "Building Java applications to use with the CCA JNI" on page 26 for supported Java levels and installation instructions.
- If you plan on maintaining a dual install environment alongside the legacy xcryptolinz RPM originally released to support the CEX2C, take note of these facts:
  - The CCA 4.x RPMs support CEX2C access, including AES (for up-to-date CEX2C firmware) and all functions formerly available for the CEX2C. It is recommended to use the latest CCA 4.x library for all CEX*C access.
  - The new TKE catcher daemon supports managing CEX2C as well as CEX3C or above. Note that CEX4Cs are only supported by TKE 7.2 or later if the Linux driver reports them as CEX4s. If you are running in toleration mode, and the Linux driver reports them as CEX3Cs, then TKE 6.0 will be able to manage them as CEX3Cs.
  - You are advised to pay special attention to Appendix K, "Coexistence of CEX3C or later and CEX2C features," on page 762 and particular the section "Dual Support: Key storage interactions" on page 770.

Default installation directory

The CCA RPM chooses a default installation directory. In the past a part of the path name was the name of the specific CCA coprocessor being supported, such as "CEX3C". As of CEX4C support (introduced with RPM level 'csulcca-4.2.10-+'), there are three CCA coprocessors supported from the same install tree, and it is no longer appropriate to continue using the CCA coprocessor name in the path. The new default install path is:
A soft-link will be added to ease migration, such that references to
/opt/IBM/CEX3C/ will still work.

If you are upgrading, note that the 4.2.10 rpm will copy your key storage files
from the old default location to the new default location. The old directory will be
kept and renamed to /opt/IBM/CEX3C-old/.

---

**Download and install the RPM**

The CCA RPM contains files, samples, and groups. The latest CCA RPM as of this
publication is a packaged build of CCA 4.2.10 for RPM platform target s390x.

**About this task**

The CCA RPM has a naming convention of csulcca-4.2.10-
*maintenance_level*.s390x.rpm, where *maintenance_level* is a number representing
the current maintenance level.

To download the RPM, complete these steps:

**Procedure**

1. Point your Web browser at this location:
   

2. Locate the box on the left side labeled Cryptocards.

3. Click the PCIe Cryptographic Coprocessor link from the Cryptocards box.

4. The Cryptocards box will be updated with a submenu underneath the PCIe
   Cryptographic Coprocessor name.

5. Find the Software download link on this submenu and click it.

6. The main page will be updated with information on downloading host
   software for various platforms.

7. On this page, find the heading IBM System z servers running Linux.

8. In the paragraph underneath this heading will be the links to download:
   • The README file for the RPM.
   • The RPM itself that installs the host code.

**Files in the RPM**

These files are included in the RPM.

`/etc/profile.d/csulcca.sh`

   Environment variables are created in this file, customers should read for
   up-to-date information. The key storage environment variables are added
   here. See “Environment variables for the key storage file” on page 346.

`/etc/profile.d/csulcca.csh`

   Environment variables are created in this file, customers should read for
   up-to-date information. The key storage environment variables are added
   here. See “Environment variables for the key storage file” on page 346.
Note: /etc/profile.d/csulcca.sh and /etc/profile.d/csulcca.csh are exactly the same in what they do, but have syntax differences. Only one of these two files is used, depending on the configuration of the particular user running an application.

/etc/init.d/CSUTKEcat
System initialization script that automatically starts the catcher.exe daemon when Linux starts. You can also use this script to start or stop catcher.exe from the command line. To start catcher.exe, issue:
/etc/init.d/CSUTKEcat start
To stop catcher.exe, issue:
/etc/init.d/CSUTKEcat stop

/etc/rc.d/rc2.d/S14CSUTKEcat
Link to: /etc/init.d/CSUTKEcat

/etc/rc.d/rc3.d/S14CSUTKEcat
Link to: /etc/init.d/CSUTKEcat

/etc/rc.d/rc5.d/S14CSUTKEcat
Link to: /etc/init.d/CSUTKEcat

/opt/IBM/CCA/bin/TKECM.dat
/opt/IBM/CCA/bin/acpoints.dat

/opt/IBM/CCA/bin/catcher.exe
Daemon executable that is controlled by the /etc/init.d/CSUTKEcat script. This daemon listens on TCP port 50003 for requests from the Trusted Key Entry workstation (TKE) for secure commands to administer any adapters configured as available to the system. See the TKE documentation for usage and capabilities. The daemon depends on libcsulcca.so and on the z90crypt device driver. While the daemon is running, it has an open handle to the z90crypt device driver. Therefore, catcher.exe as a running process affects whether you can load or unload z90crypt. Use /etc/init.d/CSUTKEcat to start catcher.exe or to stop it if you want to unload z90crypt. Remember that TKE secure administration requires a running catcher.exe process.

/opt/IBM/CCA/bin/panel.exe
Utility: run with no arguments for help

/opt/IBM/CCA/bin/ivp.e
Utility: run with no arguments for install verification

/opt/IBM/CCA/bin/profile.perl
/opt/IBM/CCA/cnm/CNM.jar
/opt/IBM/CCA/cnm/CNMMK.jar
/opt/IBM/CCA/doc/README.linz
/opt/IBM/CCA/doc/license.txt
/opt/IBM/CCA/include/csulincl.h
/opt/IBM/CCA/doc/hikmNativeInteger.html
/opt/IBM/CCA/cnm/HIKM.zip
/opt/IBM/CCA/cnm/HIKMMK.zip
/usr/lib64/libcsulcca.so
   Link to: /usr/lib64/libcsulcca.so.4

/usr/lib64/libcsulcca.so.4
   Link to: /usr/lib64/libcsulcca.so.4.2.5

/usr/lib64/libcsulcca.so.4.2.5

/usr/lib64/libcsulccamk.so
   Link to: /usr/lib64/libcsulccamk.so.4.2.5

/usr/lib64/libcsulccamk.so.4
   Link to: /usr/lib64/libcsulccamk.so.4.2.5

/usr/lib64/libcsulccamk.so.4.2.5

/opt/IBM/CCA/keys/README.keys

Samples in the RPM
These samples are included in the RPM.

/opt/IBM/CCA/samples/mac.c
   C code sample

/opt/IBM/CCA/samples/makefile.lnx
   Used to build mac.c

/opt/IBM/CCA/samples/mac.java
   Java code sample

/opt/IBM/CCA/samples/RNG.java
   Java code sample

/opt/IBM/CCA/samples/RNGpk.java
   Java code sample

Groups in the RPM
These groups are created for the purpose of loading master keys.

They are added during RPM installation as updates to /etc/groups. See Table 208 on page 764

• cca_admin
• cca_clrmk
• cca_lfmkp
• cca_cmkp
• cca_setmk

Install and configure the RPM
Use the following steps to install and configure the CCA 4.2.10 RPM.

Procedure
1. Copy the RPM to the host where it will be installed. For example, /root on your host image.
2. Login to the host as root. Change to the directory where the RPM is located by issuing these commands:
   <login to host>
   cd /root/
3. Install the RPM by issuing the following command:
rpm -i <rpm name>

Note:

a. For compatibility reasons a softlink will be created from /opt/IBM/CCA to /opt/IBM/CEX3C.

b. If this is an upgrade, you can use this command: rpm -Uvh.

c. If you are installing the RPM on a SUSE distribution of Linux, it is possible that you might receive the following warning messages because of an unsupported `groupadd` option.

```
groupadd: You are using an undocumented option (-f)!
groupadd: You are using an undocumented option (-f)!
groupadd: You are using an undocumented option (-f)!
groupadd: You are using an undocumented option (-f)!
groupadd: You are using an undocumented option (-f)!
```

No action on your part is needed. The installation proceeds with another call if this happens.

4. Reboot the host by issuing the following command:

```
shutdown -r now
```

This is necessary because of the defaults added to /etc/profile.d/csulcca.sh and /etc/profile.d/csulcca.csh for using CCA must be propagated to all user login sessions.

5. Login to the host as root. Change to the directory where the RPM binaries are installed by issuing the following command:

```
<login as root to host>
cd /opt/IBM/CCA/bin/
```

6. Verify that at least one card is present and active:

a. Ensure that the device driver is loaded by issuing the following command:

```
lsof
```

You should see the module `z90crypt` loaded. If it is not loaded, verify the contents of the kernel modules directory by issuing the following command:

```
ls /lib/modules/<Linux kernel version>/kernel/drivers/s390/crypto/
```

You should see `z90crypt.ko`. If there is just `z90crypt.ko`, then load it with the following:

```
modprobe z90crypt
```

Note:

1) This works if you have only one domain assigned to the LPAR (or z/VM guest using a dedicated CEX*C card). If more than one domain is available, you need the domain parameter in the `modprobe` command to assign a domain (or use the lowest one). If no domain parameter is specified the lowest domain will be used as the default.

2) SUSE Linux uses its own start script, named `rcz90crypt`, to do all the work. Settings can be specified using YaST.

Note: If you do not see any of these kernel modules, or if there are any errors reported from the call to `modprobe`, contact IBM Service.

b. When you are sure that the device driver is loaded, run one of the RPM-installed utilities to verify accessibility by running the following commands:

1)
This command health checks all active cards.

2) `/opt/IBM/CCA/bin/panel.exe -x`

This command shows the serial numbers and master key register states of all active cards running CCA that are visible to this Linux host. The total number of active cards and any errors is also reported.

Note:

1) To be able to use `/opt/IBM/CCA/panel.exe` the user must be either root or a member of the `cca_admin` group (the owner of `/usr/lib64/libcsulccamk.so`).

2) If there is not at least one active card at this point, double check earlier steps and, if necessary, involve IBM service because the rest of the setup is designed around having active cards.

3) Unload of the device driver requires killing the `catcher.exe` program, and then restarting it when the driver is reloaded. See the note in “Installing and loading the cryptographic device driver” on page 753 for specific instructions.

7. Master key load - This procedure is for using the Linux on System z native API or the utility (panel.exe) to load the master keys for the active cards.

   If you want to use TKE instead, refer to a TKE manual, such as the IBM Redbooks® publication *Exploiting S/390 Hardware Cryptography with Trusted Key Entry* for proper use. After completing this step using the TKE procedure, go to Step 8 on page 765.

   a. Setup the groups for the users who will be loading the master keys to the cards. Each part of the load process is owned by a different Linux group created by the RPM install procedure, and verified in the host library implementing the API allowing master key processing. To complete a specific step the user must have membership in the proper group. There are a couple ways to change group membership depending on your Linux distribution. A third option is to create the users specifically for these roles.

   If a user does not have the proper group membership for a particular master key operation, the error X'0008005a' is returned and an error message is printed to the system log.

   Note: To be able to use `/opt/IBM/CCA/panel.exe`, the user must be either root or a member of the `cca_admin` group (the owner of `/usr/lib64/libcsulccamk.so`).

   1) Group membership for Red Hat (and Fedora) based Linux distributions:

      a) Use the `groups` command to see a list of the user's current group membership:

         ```
         groups <user name>
         ---output is
         <user name> : <grouplist>
         <grouplist> is a single-space separated list
         ```

      b) `<grouplist>` must be passed along with the new group to the `usermod` command as a comma-separated list, followed by the `<user name>`. For example, if you wanted to add `cca_lfmkp` membership to user named `admin`, you would use the following commands:
groups admin
---output:
  admin : admin bin daemon sys wheel
usermod -G admin,bin,daemon,sys,wheel,cca_lfmkp admin
---output:
  [none if successful]

**Note:** Ensure the user logs out and logs back in, otherwise the
group membership in the active session will not be updated.

2) Group membership for SUSE-based Linux distributions:
   a) Use the `usermod` command to add membership for a specific group
      for a specific user. For example, if you wanted to add `cca_lfmkp`
      membership to user `admin`, you would use the following
      commands:
      ```bash
      usermod -A cca_lfmkp admin
      ```
      **Note:** Ensure the user logs out and logs back in, otherwise the
      group membership in the active session will not be updated.

3) Create users for each role with correct group memberships (Same
   commands for Red Hat, Fedora, and SUSE):
   a) Create user `cca_user`, which will own default key storage by
      issuing the following commands:
      i. 
      ```bash
      useradd -g cca_admin -d /home/cca_user -m cca_user
      ```
      This command creates the user with primary group `cca_admin`
      and a new home directory.
      ii. 
      ```bash
      passwd cca_user
      ```
      This command sets the new user's password.
   b) Create user `cca_lfmkp` by issuing the following commands:
      i. 
      ```bash
      useradd -g cca_admin -d /home/cca_lfmkp -G
              cca_admin,cca_lfmkp -m cca_lfmkp
      ```
      This command creates the user with primary group `cca_admin`,
      secondary group `cca_lfmkp`, and a new home directory.
      ii. 
      ```bash
      passwd cca_lfmkp
      ```
      This command sets the new user's password.
   c) Create user `cca_cmkp` by issuing the following commands:
      i. 
      ```bash
      useradd -g cca_admin -d /home/cca_cmkp -G
              cca_admin,cca_cmkp -m cca_cmkp
      ```
      This command creates the user with primary group `cca_admin`,
      secondary group `cca_cmkp`, and a new home directory.
      ii. 
      ```bash
      passwd cca_cmkp
      ```
      This command sets the new user's password.
d) Create user **cca_clrmk** by issuing the following commands:
   i.  
      ```
      useradd -g cca_admin -d /home/cca_clrmk -G cca_admin,cca_clrmk -m cca_clrmk
      ```
      This command creates the user with primary group **cca_admin**, secondary group **cca_clrmk**, and a new home directory.
   ii.  
      ```
      passwd cca_clrmk
      ```
      This command sets the new user's password.

e) Create user **cca_setmk** by issuing the following commands:
   i.  
      ```
      useradd -g cca_admin -d /home/cca_setmk -G cca_admin,cca_setmk -m cca_setmk
      ```
      This command creates the user with primary group **cca_admin**, secondary group **cca_setmk**, and a new home directory.
   ii.  
      ```
      passwd cca_setmk
      ```
      This command sets the new user's password.

b. Add group membership privileges to users based on their required function.

<table>
<thead>
<tr>
<th>Group Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cca_admin</td>
<td>All users who will run part of the master key load process must be in this group because the library itself is owned by root.cca_admin, with no permissions for 'world' as a protective measure. Reasons for this separate group also include allowing one owner of /usr/lib64/libcsulccamk.so, and allowing use of panel.exe without allowing any of the master key processing calls.</td>
</tr>
<tr>
<td>cca_lfmkp</td>
<td>The user to LOAD the first key part must be in this group.</td>
</tr>
<tr>
<td>cca_cmkp</td>
<td>The users to LOAD the middle and last key parts must be in this group.</td>
</tr>
<tr>
<td>cca_clrmk</td>
<td>The new master-key register can be CLEARed using the same Master Key Process call in case a mistake was made entering a key part (use the key verification patterns to check for this). To perform the clear, the user must be a member of this group.</td>
</tr>
<tr>
<td>cca_setmk</td>
<td>The user to call SET after the last key part has been successfully loaded must be a member of this Linux group.</td>
</tr>
</tbody>
</table>

c. Load FIRST, MIDDLE (optional), and LAST key parts for the AES, SYM, ASYM, and APKA master keys and then call SET for each master key. This step can be done using the panel.exe utility provided or by writing your own application to call the Master Key Process (CSNBMKP) verb directly. The application must link with the correct library (installed to /usr/lib64/libcsulccamk.so by the RPM), and must be executed at each step by a user with the appropriate group memberships. The utility supports scripted as well as prompt-driven access.

Repeat this step for each configured adapter. See "Changing the master key for two or more adapters that have the same master key, with shared CCA key storage" on page 350.

For details about panel.exe, see "The panel.exe utility" on page 773.
See "Master Key Process (CSNBMKP)" on page 119 about parity requirements for master key parts.

Note: Loading master key parts modifies state information inside the card. For example you cannot load a 'FIRST' master key part twice in a row without clearing the new master-key register in between attempts. The same goes for setting the 'LAST' register. Any number of 'MIDDLE' parts can be loaded - with each call changing the contents of the new master-key register. Similarly a 'SET' operation changes the state of the 'new' register back to 'empty', while updating the 'current' register.

8. **Key storage initialization** - To perform this step, see "Using panel.exe for key storage initialization" on page 775. See also "Dual Support: Key storage interactions" on page 770.

9. **Key storage re-encipher when changing the master key** - To perform this step, see "Using panel.exe for key storage reencipher when changing the master key" on page 777.

10. If you are going to be using Central Processor Assist for Cryptographic Functions (CPACF), it must be configured. See "CPACF support" on page 12.

---

### Uninstall the RPM

Use the following steps to uninstall the CCA Release 4.2.10 RPM.

**Procedure**

1. Uninstall any RPMs that depend on the CCA RPM. If you try to uninstall the CCA RPM and dependent RPMs are still installed, the uninstall RPM command will fail and list the names of dependent RPMs. Therefore, you can skip to Step 2 and come back to this step if Step 2 fails for that reason.

2. Uninstall the CCA RPM.
   a. Login as root. You have to be root to uninstall the RPM.
   b. You have to use the full name. You can find the name by issuing the following command:

   ```
rpm -qa | grep csulcca
   ```
   c. Uninstall the RPM with the following command:

   ```
   rpm -e <rpm name>
   ```

**Note:**

a. Groups are no longer deleted during the uninstall of CCA RPM. If you created any users with one of the groups created by the RPM install as their primary (note that the RPM install does NOT create any users, just groups), you can delete those users/groups yourself after uninstall, or remove such users before the uninstall of the RPM. This will remove any potential security holes.

b. Card master keys (and other state information) are untouched by the host-side uninstall of the RPM.

c. Key storage files are not deleted by the uninstall. All default and nondefault key storage files will be left as is. If you reinstall or install an upgraded package and load any new cards with the same master keys you will still be able to use your old key storage (old cards will still have the old keys, see Step 7b on page 764 of "Install and configure the RPM" on page 760).
Appendix K. Coexistence of CEX3C or later and CEX2C features

This appendix describes trade-offs when configuring CEX3C or later and CEX2C features available to the same Linux instance.

These terms are used to describe CCA RPMs, referring to several different versions:

CCA 4.x
CCA 4.0.0, CCA 4.1.0, CCA 4.2, and subsequent releases.

CCA 3.x
Any CCA version 3 release.

Legacy support

These are legacy support considerations.

- The latest CCA 4.x RPM will supersede and replace earlier RPMs if installed in the default manner. It is NOT recommended to manually alter installation in order to run newer versions of CCA 4.x RPMs in parallel with older versions. This configuration is not supported.

- Running the latest CCA 4.x RPM in parallel with the IBM 3.x RPM and an IBM Crypto Express2 (CEX2C) feature in coprocessor mode configured to the same system image is a supported configuration. See “Concurrent installations.”

- CCA 4.x RPMs include support for interaction with the CEX2C feature in coprocessor mode. See “Dual Support: TKE catcher can run in only one instance” on page 771.

Concurrent installations

These are background considerations for installation of the CEX3C or later and CEX2C RPM alongside the previously released RPM for the CEX2C.

1. The libcsulcca.so and libcsulsapi.so libraries for CCA 4.x and CCA 3.x have many symbols with the same names. An application cannot deterministically link with both libraries. The first library in the link statement is what will be used for all symbols that can be resolved there, after that the second library will be examined. At this point, either the linker will not allow link to continue, by throwing an error on the duplicate symbols, or will produce a hybrid-linked application. Either case will give the user the wrong answer. A new or updated library cannot itself resolve this kind of conflict because:

   - There is no way to have a default set of symbols or card support in an updated host library. The link operation is a fundamental step in building the customer application and outside the control of the library or library installation process.

   - One way to resolve name collisions is to change all of the function names in the new library. However, this would have greatly impacted the customer’s ability to port applications forward, and this option was rejected.

2. The key storage environment variables in the default user profile (/etc/profile.d/csulcca.sh and /etc/profile.d/csulcca.csh) are changed at installation time to point to the /opt/IBM/CCA/keys/ path, where before the
path contained */4764/*. There is one set of environment variables for a profile. The user can overcome this by setting a local profile in their home profile file that sets the environment variables back to the 4764 version. See “Dual Support: Key storage interactions” on page 770.

3. See “Interaction between the 'default card' and use of Protected Key CPACF” on page 17 for a concurrency and CPACF.

**CEX3C or later and CEX2C co-installation toleration**

The CCA 4.x RPMs all support accessing the CEX2C feature as well as the CEX3C or later feature, with the first CEX3C or later feature becoming the 'default' adapter.

This can be changed using environment variables. See “Environment variables that affect CPACF usage” on page 13. Using CCA 4.x is your best option for accessing a CEX2C feature as well as a CEX3C or later feature going forward, even in a CEX2C-only installation.

**Installing the csulcca RPM over an existing xcryptolinz RPM:**

During installation the new csulcca RPM will look for and rearrange the xcryptolinz RPM pieces that conflict with the new RPM. These consist of a few soft links and some profile settings.

Your old key storage will not be accessed, deleted or modified; it will also not be migrated. As long as the appropriate master keys are set in the CEX3C or later to be the same as the equivalent master keys in the CEX2C, the old key storage can be simply used by the new host library. There is a set of environment variables that control where key storage is found. See “Environment variables for the key storage file” on page 346.

The csulcca RPM does not replace the xcryptolinz RPM, the csulcca RPM will live alongside it, in order to ease the transition process.

**Temporary toleration approach to avoid re-linking applications:**

Because the new RPM has a new name for the CCA host library, it is necessary to re-link your application with the new library. There is a quick method for toleration if this is not immediately possible. Create soft links in /usr/lib64/ from the new libraries to the names of the libraries that existed before:

1. **IMPORTANT** Delete or move the old libraries first.
2. Create a soft link of libcsulcca.so.4.2.0, libcsulcca.so.4.1.0 or libcsulcca.so.4.0.0 to libcsulsapi.so, libcsulsecy.so, libds30.so and libcsulcall.
3. Create a soft link of libcsulccamk.so.4.2.0, libcsulccamk.so.4.1.0 or libcsulccamk.so.4.0.0 to libcsulmkapi.so
4. Run the ldconfig command.
Uninstalling the xcryptolinz RPM:

This task is not impacted by the changes the csulcca RPM makes when it is installed.

Re-installing the xcryptolinz RPM with the csulcca RPM installed:

Re-installing the xcryptolinz RPM with the csulcca RPM installed

<table>
<thead>
<tr>
<th>IMPORTANT</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is not a supported operation. The csulcca RPM cannot detect this scenario to try to recover the csulcca package function.</td>
</tr>
</tbody>
</table>

Difficulties noted above for the coexistence scenario make supporting consistent operation of the csulcca RPM while allowing reinstall of the xcryptolinz RPM impossible. The xcryptolinz RPM install will create TKE daemon soft links pointing to the old TKE daemon (which cannot communicate with CEX3C or later adapters) and corrupt the standard profile settings updated for the csulcca install.

If a refresh of the xcryptolinz RPM is truly needed, choose one of these three methods:

1. By uninstalling and installing:
   a. Uninstall the csulcca RPM first, in order to have a clean system image. The key storage file will be left intact.
   b. Install the xcryptolinz RPM.
   c. Reinstall the csulcca RPM so that the updated function is back in place.

2. By using tools and copying files:
   a. Use the rpm2cpio and cpio tools to extract only the files needed from the xcryptolinz RPM.
   b. Copy the needed files into place manually.

3. By using soft links and environment variables:
   a. Create the soft links from the new RPM host libraries to the old library names. See “Temporary toleration approach to avoid re-linking applications” on page 768.
   b. Point the new RPM host library at your old key storage file using the environment variables, which can be done on a per-process basis.

See “Environment variables for the key storage file” on page 346 for information about the environment variables that are used to specify the name of the key storage files.

Fixing the csulcca RPM install after installing xcryptolinzGA on top of it:

It is possible to simply recover the csulcca RPM install state with these two procedures.

It is recommended that you subsequently reboot the system image. Perform these steps (as root):

1. Run this command:
   ```
   /opt/IBM/CCA/bin/profile.perl delete
   ```
   This will remove the environment variables added to /etc/profile by the xcryptolinzGA RPM. If you need those variables set in a particular application
space, set them in startup scripts for the application that needs them. Because these variables are positioned at the end of /etc/profile, they disable the csulcca RPM configuration.

2. Delete startup file links by issuing these commands:
   ```bash
   rm -f /etc/init.d/rc2.d/S16TKEcat
   rm -f /etc/init.d/rc3.d/S16TKEcat
   rm -f /etc/init.d/rc5.d/S16TKEcat
   ```
   These startup files cause the wrong TKE catcher daemon to be loaded, which cannot communicate with CEX3C or later adapters. The new TKE catcher daemon can work with both CEX2C and CEX3C or later adapters, so it is preferred for all systems with co-install of the xcryptolinzGA RPM and the csulcca RPM.

**Caveats:**

Recommendations worth considering.

- Be very careful with setting Master Keys and allowing other accesses. The older libraries will not detect the difference between a CEX2C and a CEX3C or later, and will attempt to access the new cards if allocated by the user.
- It is best to use a concurrent environment as a temporary aid to a porting effort, the result of which is an application that can use the card it desires allocated through the new CEX*C library.

**Dual Support: Key storage interactions**

All the factors together listed in this topic should be considered, and managed carefully for user installations.

- In all the CCA 4.x RPMs, the environment variables described in "Environment variables for the key storage file" on page 346 have the same names. The purpose is to ease application porting efforts.
- The recommended method to accomplish coexistence of the two CCA packages for porting or debug environments is the installation of the CCA 4.x RPMs on top of an existing xcryptolinz RPM installation. This installation will change how the environment variables are defined for any user performing login after the CCA 4.x RPM installation, such that the environment variables will point to the new location.
- Therefore, legacy applications that have not defined their own key storage environments (those using the default profile location) will now be using the new key storage file defined for the CCA 4.x RPMs. These are likely consequences the next time that the legacy application starts:
  - The legacy application will not be able to find its existing keys.
  - The legacy application may corrupt key storage for applications using the new CCA 4.x RPM location for key storage.

**Solution**

The application developer should ensure that legacy applications are started using the definitions for the key storage environment variables that they require.

- The environment variables were placed in file /etc/profile by the xcryptolinz RPM installation (look for the LINZCRYPT section). They were defined as follows:
  - CSUDESLD=/opt/IBM/4764/keys/deslist
  - CSUDESRS=/opt/IBM/4764/keys/des.key
  - CSUPKALD=/opt/IBM/4764/keys/pkalist
- CSUPKADS=/opt/IBM/4764/keys/pka.key
- LD_LIBRARY_PATH=/usr/lib64
- CNM_CLASSPATH=/opt/IBM/CCA/cnm/
- CSUSRDI=$HOME/srdidata

- There are several ways to ensure that your application is started with these environment variables instead of the defaults. One straightforward way is to complete the following steps:
  1. Manually export the definitions using the command export.
  2. Check that they are set correctly using the commands: env or printenv.
  3. Manually start the application that needs the special environment.

You can use AES key storage functions with a CEX2C feature only if you have installed the CCA 4.* RPMs. This is because the environment variables CSUAESLD and CSUAESDS are necessary for AES key storage, but they did not ship with the CCA 3.* RPMs. These environment variables are defined as follows:
- CSUAESLD=/opt/IBM/CCA/keys/aeslist (for 4.2.10)
- CSUAESDS=/opt/IBM/CCA/keys/aes.key

**Dual Support: TKE catcher can run in only one instance**

The Trusted Key Entry (TKE) catcher daemon is used to interface with the TKE workstation.

This daemon listens on a single port for management communication. This port number has not changed for the CEX3C or later release. Therefore, the new daemon supports TKE management communication to both the CEX2C and CEX3C or later adapters. Special steps are taken in the install/uninstall and daemon management for the CEX3C or later release to ensure that the new daemon is running when it is available.

You must have a TKE V6.0 or higher workstation in order to see supported CEX3Cs or CEX4Cs. Note that CEX4C are reported by TKE7.2 as CEX3Cs. None are seen when using TKE V5 workstations. Also, CEX4Cs are only supported by TKE 7.2 or later if the Linux driver reports them as CEX4s. If you are running in toleration mode, and Linux reports them as CEX3Cs, then TKE 6.0 will be able to manage them as CEX3Cs. For more information about the TKE workstation, see z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User’s Guide.
Appendix L. Utilities

This appendix describes two utilities used in this document.

These two utilities are:

**ivp.e**
This utility is used to verify installation, when run without arguments.

This utility can also be used to tell you the coprocessor type, by calling the Cryptographic Facility Query verb for all available adapters.

**panel.exe**
This utility provides a Linux native mechanism for administering and initializing certain characteristics of active cryptographic coprocessors. It is intended as a basic administration tool for Linux-only IBM System z configurations, where a Trusted Key Entry (TKE) solution is not available.

For mixed z/OS and Linux configurations, it is recommended that administration be accomplished using the z/OS TSO panels as described in the z/OS ICSF Administrator’s Guide. The utility is installed by the Linux for System z Cryptographic Coprocessor install package or RPM to this path in the Linux system:

```
/opt/IBM/CCA/bin/panel.exe
```

The panel.exe utility

The panel.exe utility is installed by the Linux for IBM System z Cryptographic Coprocessor install package or RPM.

It is installed to this path in the Linux system:

```
/opt/IBM/CCA/bin/panel.exe
```

The panel.exe utility numbers cards from card0 to card63, while verbs such as Cryptographic Resource Allocate number cards from CRP01 to CRP64, and therefore card0 corresponds to CRP01, card1 corresponds to CRP02, and so forth.

**panel.exe syntax**

Precise usage information can be obtained by running the panel.exe utility with no arguments on the Linux shell command line.

This is an example output:

```
Panel usage ([[-k,-a <num>, -o, -g][-?,-x,-m,-l,-s,-c,-q,-t,-f,-i,-r,-p,-n] >> [CC] Arg >> arg must precede non-[CC] args
  [CC] -k: Can TKE administer a card?
  [CC] -a <num>: use non-default card
    <num> is the card number [0 - 63]
  [CC] -o: Disable output to stdout:
  [CC] -g <level>: Set the log level:
    <level> can be NONE, TRANSACTIONS, NONZERO,
      ALL, DEBUG, and FUNCTIONS

>> non-[CC] Args >>: (all are mutually exclusive)
```

---BASIC ADMIN---
---MASTER KEY (MK)---

To LOAD a Master Key (MK) PART:
- l (for interactive)
  OR====>
  - l -t [A|S|E|P] -p [F|M|L] KEYPART
  where: -t [A|S|E|P] is which MK: A=ASYM, S=SYM, E=AES P=APKA
  where: -p [F|M|L] is the part: F=FIRST, M=MIDDLE, L=LAST
  where: KEYPART is string in hex 2* size of key
  (recall: 2 text chars = 1 binary Byte)

To SET a Master Key:
- s (for interactive)
  OR====>
  - s -t [A|S|E|P]
  where: -t [A|S|E|P] is which MK: A=ASYM, S=SYM, E=AES P=APKA

To CLEAR a Master Key 'New' Register:
- c (for interactive)
  OR====>
  - c -t [A|S|E|P]
  where: -t [A|S|E|P] is which MK: A=ASYM, S=SYM, E=AES P=APKA

To QUERY a Master Key Verification Pattern:
- q (for interactive)
  OR====>
  - q -t [A|S|E|P] -r [N|C|O]
  where: -r [N|C|O] is which register: N=NEW, C=CURRENT, O=OLD

---KEY STORAGE---

To INIT a KEY STORAGE file:
- t <type> -f <file> -i

To REENCipher KEY STORAGE:
- t <type> -f <file> -r

To LIST a KEY STORAGE:
- t <type> -f <file> -p

where:
  <type> can be AES, DES, PKA
  <file> is the fully qualified name of a key storage file

---RETAINED KEYS---

To LIST RETAINED KEYS (this domain ONLY):
- n

---panel.exe functions---

Different uses for the panel.exe utility.

The panel.exe utility can be used to:
- Determine if a TKE is currently able to administer a specific active coprocessor
- List the labels and key types for all the keys in a designated key storage file.

Note: For security reasons, only a root user (real user id equal to '0') is allowed to use panel.exe to load master key parts or to clear previously loaded master key parts. This is enforced at the shared library level in the implementation of the Master Key Process verb, not in the utility itself. Additionally, only the user who created a set of key storage files or the 'root' user will be able to take actions with respect to those key storage files, based on Linux file system permissions.
List the labels for all of the retained keys (RSA private keys stored in the adapter) in the current domain of the CEX*C.

List the coprocessors currently active in the Linux system and their master key status

Load master key parts to the coprocessor

Set a master key that was loaded to the coprocessor. Note that panel.exe, is not designed to change the master keys for all the cards in a group; this is a more sophisticated operation.

Clear master key parts which were previously loaded to the coprocessor but not yet 'set' or confirmed (used for when a mistake in entering master key parts has been detected)

List serial numbers and master key register states of all active cards running CCA that are visible to this Linux host. The total number of active cards and any errors will also be reported.

Query the master key verification pattern for any master-key register in the current domain

Initialize a local host key storage file. See “Using panel.exe for key storage initialization.”

Re-encipher a local host key storage file (use this when the master key has been changed to ensure currency with key storage). See “Using panel.exe for key storage reencipher when changing the master key” on page 777.

List available CPACF functions, and whether they are supported in the current system image.

The panel.exe utility does **not** support access control point manipulation or more sophisticated administration. Refer to “Trusted Key Entry support” on page 51 for that functionality.

**Using panel.exe for key storage initialization**

Using panel.exe for key storage initialization

Each application using CCA typically creates key objects that are stored in the host, protected by the master key stored inside the card. Perform these steps for key storage initialization.

1. The default locations for the files are setup by the RPM in environment variables added in the new profile files /etc/profile.d/csulcca.sh and /etc/profile.d/csulcca.csh during installation. Key storage is unsupported without a master key loaded, so **Master key load** (Step 7 on page 762) must be completed before this step. The utility panel.exe can be used to initialize both the default key storage and any separate key storage you might want to set up. The full topic is too lengthy for this explanation (see the key storage topics elsewhere in this manual, including the verb “Key Storage Initialization” (CSNBKSI)” on page 116). In brief, an application can specify a particular key storage location. That nondefault key storage can be initialized now (or later) by using panel.exe or with a program using the Key Storage Initialization verb. For details about panel.exe, see “The panel.exe utility” on page 773.

   If you are planning to use both the CEX2C and CEX3C or later in the same environment, see the information about the key storage environment variable in “Concurrent installations” on page 767.

2. The key storage environment variables in the default user profile (/etc/profile.d/csulcca.sh) are changed at installation time to point to the /opt/IBM/CCA/keys/ path, where before the path contained /*/4764/*. There

Appendix L. Utilities 775
is one set of environment variables for a profile. The user can override this by setting a local profile in their home profile file that sets the environment variables back to the 4764 version.

3. Key storage ownership

The default key storage files are actually partially created (but not fully initialized) during the master key load process. This means the ownership and permissions of those files might have to be changed for them to be fully initialized by the user associated with the application that will use the key storage files.

Because of the mutually exclusive nature of the master key admin groups, there can be some harmless access errors reported to the system log during master key load. The example users created previously in Master key load Step 7a3 on page 763 will avoid this and not need to fix key storage ownership because they were all created with the primary group set to 'cca_admin' (the -g argument to useradd). By doing this, the first master key load creates the key storage files with group set to 'cca_admin' and subsequent users all have membership in that group. You still might want to fix the owner of default key storage at the end to be 'root', but the group membership solves the access issue.

Typically 'root' will need to fix the ownership and permissions. We recommend that the owner of key storage be 'root', and that the group be 'cca_admin' ('cca_admin' group is created during the RPM install process). We recommend that the permissions be set to 660, which is rw for owner (root), rw for group (cca_admin), and <none> for 'everyone', for security. Then add the application user to the group 'cca_admin' with the appropriate procedure detailed in Master key load Step 7a on page 762.

RECALL: To be able to use /opt/IBM/CCA/panel.exe the user must be either root OR a member of the 'cca_admin' group (the owner/group of /usr/lib64/libcsulccamk.so). The reasons for the separate 'cca_admin' group are to allow one owner of /usr/lib64/libcsulccamk.so, and to allow use of the executable without allowing any of the master key processing calls.

4. Key storage initialization with panel.exe. This is the default.

a. Ensure permissions to the default location (/opt/IBM/CCA/keys/) allow your user to perform this operation.

b. Initialize key storage (DES is where DES key tokens will be kept, AES is where AES key tokens will be kept, PKA is for all the RSA public/private internal key tokens, and APKA is for APKA key tokens).

   /opt/IBM/CCA/bin/panel.exe -t AES -i
   /opt/IBM/CCA/bin/panel.exe -t DES -i
   /opt/IBM/CCA/bin/panel.exe -t PKA -i

5. Key storage initialization with panel.exe (non-default)

a. Ensure that you are using the account that will use the key storage. If you are not, you will have to fix its ownership and permissions later.

b. Initialize both types of key storage (DES is where DES key tokens will be kept, AES is where AES key tokens will be kept, PKA is for all the RSA public/private internal key tokens). Use a different name for AES, DES, and PKA, because the second initialization would overwrite the first if different names are not used. The file name passed is expected to be the full or relative path and will actually be the core of the filename, because more than one file is created using the stem you provide. To initialize AES, DES and PKA storage, use the following commands:
/opt/IBM/CCA/bin/panel.exe -t AES -f <AES file name> -i
/opt/IBM/CCA/bin/panel.exe -t DES -f <DES file name> -i
/opt/IBM/CCA/bin/panel.exe -t PKA -f <PKA file name> -i

For example, if you entered the following commands:

/opt/IBM/CCA/bin/panel.exe -t AES -f /tmp/a -i
/opt/IBM/CCA/bin/panel.exe -t DES -f /tmp/d -i
/opt/IBM/CCA/bin/panel.exe -t PKA -f /tmp/p -i

These files would be created:
/tmp/a
/tmp/a.NDX
/tmp/d
/tmp/d.NDX
/tmp/p
/tmp/p.NDX

Using panel.exe for key storage reencryption when changing the master key

Because all the key tokens are protected by the master key for the domain, a
preexisting key storage must be re-encrypted when the master key is changed. I

If the example group scheme is used this is very simple because the key storage
files will be owned by the group 'cca_admin' and the user making the reencryption
call will also be in group 'cca_admin'. If this is not the case then, after changing the
master key, the owner of key storage will need to login and drive the reencryption.
This can be done programmatically (using several verbs) or with
/opt/IBM/CCA/panel.exe. Of course, as noted, the user of panel.exe must also be
a member of 'cca_admin' because of ownership of /usr/lib64/libcsulccamk.so.

Perform these steps for key storage reencryption when changing the master key.
1. To re-encrypt default key storage with panel.exe use:

   /opt/IBM/CCA/bin/panel.exe -t AES -r
   /opt/IBM/CCA/bin/panel.exe -t DES -r
   /opt/IBM/CCA/bin/panel.exe -t PKA -r

2. To re-encrypt non-default key storage with panel.exe use:

   /opt/IBM/CCA/bin/panel.exe -t AES -f <AES file name> -r
   /opt/IBM/CCA/bin/panel.exe -t DES -f <DES file name> -r
   /opt/IBM/CCA/bin/panel.exe -t PKA -f <PKA file name> -r
Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Documentation accessibility

The Linux on System z publications are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF file and want to request a Web-based format for this publication, use the Reader Comment Form in the back of this publication, send an email to eservdoc@de.ibm.com, or write to:

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

In the request, be sure to include the publication number and title.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility at

www.ibm.com/able
Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.
This information is for planning purposes only. The information herein is subject to change before the products described become available.

**Programming interface information**

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of the Common Cryptographic Architecture.

**Trademarks**

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at:


Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.
List of abbreviations

A list of abbreviations used in this document.

AES  Advanced Encryption Standard.
ADB  Actual Data Block
AIX® Advanced Interactive Executive operating system
ANS  American National Standards
ANSI American National Standards Institute
API  Application Programming Interface
ASCII American National Standard Code for Information Interchange
ASN  Abstract Syntax Notation
ATC  Application Transaction Counter
ATM  Automated Teller Machine
BC   Block Contents
BDK  Base Derivation Key
BER  ASN.1 Basic Encoding Rules
CA   Certification Authority
CBC  Cipher block chaining.
CCA  Common Cryptographic Architecture.
CCF  Cryptographic Coprocessor Feature.
CEX2A Crypto Express2 Accelerator
CEX2C Crypto Express2 Coprocessor
CEX3A Crypto Express3 Accelerator
CEX3C Crypto Express3 Coprocessor
CKDS Cryptographic Key Data Set.
CKSN Current-Key Serial Number
CLU  Coprocessor Load Utility
CMK  Current Master Key
CMOS Complementary Metal Oxide Semiconductor
CMS  Cryptographic Message Syntax
CNI  Coprocessor Node Initialization
CNM  Cryptographic Node Management (utility)
COBOL Common Business-Oriented Language
CRT Chinese Remainder Theorem.
CPACF Central Processor Assist for Cryptographic Functions
CSC Card Security Code
CV Control Vector
CVC Card verification code used by MasterCard.
CVK Card Verification Key
CVV Card verification value used by VISA.
DEA Data encryption algorithm.
DES Data Encryption Standard.
DMA Direct Memory Access
DOW Day of the Week
DSA Digital Signature Algorithm.
DSS Digital Signature Standard.
DUKPT Derived Unique Key Per Transaction
EDE Encipher-Decipher-Encipher
EEPROM Electrically Erasable, Programmable Read-Only Memory
EBCDIC Extended Binary Coded Decimal Interchange Code
EC Elliptic Curve
ECB Electronic codebook.
ECC Elliptic Curve Cryptography
ECDH Elliptic Curve Diffie-Hellman
ECDSA Elliptic Curve Digital Signature Algorithm
ECI Eurocheque International
EID Environment Identification.
EPP Encrypting PIN PAD
FCV Function Control Vector
FIPS Federal Information Processing Standards
GBP German Bank Pool.
HMAC Keyed Hash MAC
HSM Hardware Security Module
IBM International Business Machines
ICSF Integrated Cryptographic Service Facility.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICV</td>
<td>Initial Chaining Value</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IPL</td>
<td>Initial Program Load</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization.</td>
</tr>
<tr>
<td>ISO/DIS</td>
<td>International Organization for Standardization/Draft International Standard</td>
</tr>
<tr>
<td>ISO/FDIS</td>
<td>International Organization for Standardization/Final Draft International Standard</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>KC</td>
<td>Key Confirmation</td>
</tr>
<tr>
<td>KDF</td>
<td>Key Derivation Function</td>
</tr>
<tr>
<td>KEK</td>
<td>Key-Encrypting Key</td>
</tr>
<tr>
<td>KM</td>
<td>Master Key</td>
</tr>
<tr>
<td>KVP</td>
<td>Key Verification Pattern</td>
</tr>
<tr>
<td>LRC</td>
<td>Longitudinal Redundancy Check</td>
</tr>
<tr>
<td>LSB</td>
<td>Least significant bit</td>
</tr>
<tr>
<td>MB</td>
<td>Megabyte</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>MD5</td>
<td>Message Digest-5 Hash Algorithm</td>
</tr>
<tr>
<td>MDC</td>
<td>Modification Detection Code</td>
</tr>
<tr>
<td>MDK</td>
<td>Master-Derivation Key</td>
</tr>
<tr>
<td>MFK</td>
<td>Master File Key</td>
</tr>
<tr>
<td>MK</td>
<td>Master Key</td>
</tr>
<tr>
<td>MKVP</td>
<td>Master-Key Verification Pattern</td>
</tr>
<tr>
<td>MSB</td>
<td>Most significant bit</td>
</tr>
<tr>
<td>NIST</td>
<td>U.S. National Institute of Science and Technology</td>
</tr>
<tr>
<td>NL</td>
<td>Netherlands</td>
</tr>
<tr>
<td>NMK</td>
<td>New Master Key</td>
</tr>
<tr>
<td>OAEP</td>
<td>Optimal asymmetric encryption padding</td>
</tr>
<tr>
<td>OCSP</td>
<td>Open Certificate Status Protocol</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer</td>
</tr>
<tr>
<td>OID</td>
<td>Object Identifier</td>
</tr>
<tr>
<td>OMK</td>
<td>Old Master Key</td>
</tr>
<tr>
<td>OPK</td>
<td>Object Protection Key</td>
</tr>
<tr>
<td>PAN</td>
<td>Personal Account Number.</td>
</tr>
</tbody>
</table>
PCI  Peripheral Component Interconnect
PCIe  PCI Express
PCI-X  PCI Extended
PCICA  PCI Cryptographic Accelerator.
PCICC  PCI Cryptographic Coprocessor.
PCIXCC  PCI X Cryptographic Coprocessor.
PKA  Public Key Algorithm.
PKCS  Public Key Cryptographic Standards (RSA Data Security, Inc.)
PKDS  Public key data set (PKA cryptographic key data set).
PIN  Personal Identification Number
PKA  Public Key Algorithm
PKCS  Public-Key Cryptography Standards
RMF™  Resource Manager Interface.
RMI  Resource Measurement Facility™.
POST  Power-On Self Test
PRNG  Pseudo Random Number Generator
PROM  Programmable Read-Only Memory
PVV  PIN Validation Value
RA  Registration Authority
RACF  Resource Access Control Facility
RAM  Random Access Memory
RFC  Request for Comments
RHEL  Red Hat Enterprise Linux
RNG  Random Number Generator
ROM  Read-Only Memory
RSA  Rivest, Shamir, and Adleman
SEC 2  Standards for Efficient Cryptography 2
SECG  Standards for Efficient Cryptography Group
SET  Secure Electronic Transaction.
SHA  Secure Hash Algorithm
SLES  SUSE Linux Enterprise Server
SNA  Systems Network Architecture
SSL  Secure Sockets Layer.
TDEA  Triple Data Encryption Algorithm.
TKE  Trusted key entry.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLV</td>
<td>Tag, Length, Value</td>
</tr>
<tr>
<td>TVV</td>
<td>Token-validation value</td>
</tr>
<tr>
<td>UAT</td>
<td>UDX Authority Table.</td>
</tr>
<tr>
<td>UDF</td>
<td>User-defined function.</td>
</tr>
<tr>
<td>UDK</td>
<td>User-derived key.</td>
</tr>
<tr>
<td>UDP</td>
<td>User Developed Program.</td>
</tr>
<tr>
<td>UDX</td>
<td>User Defined Extension.</td>
</tr>
<tr>
<td>UKPT</td>
<td>Unique Key Per Transaction</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>VIS</td>
<td>Visa Integrated Circuit Card Specification</td>
</tr>
<tr>
<td>XOR</td>
<td>Exlusive-OR</td>
</tr>
</tbody>
</table>
Glossary

This glossary includes some terms and definitions from the IBM Dictionary of Computing, New York: McGraw Hill, 1994. This glossary also includes some terms and definitions from:

- The IBM Glossary of Computing Terms. Definitions are identified by the symbol (D) after the definition.
- The IBM TotalStorage Enterprise Storage Server® documentation. Definitions of published parts of this vocabulary are identified by the symbol (E) after the definition.
- The Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

access A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

access control Ensuring that the resources of a computer system can be accessed only by authorized users in authorized ways.

access method A technique for moving data between main storage and input/output devices.

adapter A printed circuit card that modifies the system unit to allow it to operate in a particular way.

address In data communication, the unique code assigned to each device or workstation connected to a network. A character or group of characters that identifies a register, a particular part of storage, or some other data source or data destination. (A) To refer to a device or an item of data by its address. (A) (I)

Advanced Encryption Standard (AES) A data encryption technique that improved upon and officially replaced the Data Encryption Standard (DES). AES is sometimes referred to as Rijndael, which is the algorithm on which the standard is based.

Advanced Interactive Executive (AIX) operating system IBM's implementation of the UNIX6 operating system.

American National Standard Code for Information Interchange (ASCII) The standard code using a coded character set consisting of 7-bit characters (8 bits including parity check) that is used for information exchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

American Institute of Standardization (ANSI) An organization, consisting of producers, consumers, and general interest groups that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. (A)

ANSI key-encrypting key (AKEK) A 64- or 128-bit key used exclusively in ANSI X9.17 key management applications to protect data keys exchanged between systems.

ANSI X9.17 An ANSI standard that specifies algorithms and messages for DES key distribution.

6. UNIX is a trademark of UNIX Systems Laboratories, Incorporated.
ANSI X9.19
An ANSI standard that specifies an optional double-MAC procedure which requires a double-length MAC key.

application program
A program written for or by a user that applies to the user’s work, such as a program that does inventory control or payroll.

A program used to connect and communicate with stations in a network, enabling users to perform application-oriented activities. (D)

application program interface (API)
A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. (D)

Application System/400 system (AS/400)
AS/400 was one of a family of general purpose midrange systems with a single operating system, Operating System/400®, that provides application portability across all models. AS/400 is now referred to as IBM System i.

assembler language
A source language that includes symbolic machine language statements in which there is a one-to-one correspondence between the instruction formats and the data formats of the computer.

asymmetric cryptography
Synonym for public key cryptography. (D)

authentication
A process used to verify the integrity of transmitted data, especially a message (I). In computer security, a process used to verify the user of an information system or protected resources.

authentication pattern
An 8-byte pattern that is calculated from the master key when initializing the cryptographic key data set. The value of the authentication pattern is placed in the header record of the cryptographic key data set.

authorize
To permit or give authority to a user to communicate with or make use of an object, resource, or function.

authorization
The right granted to a user to communicate with or make use of a computer system (I). The process of granting a user either complete or restricted access to an object, resource, or function.

bus
In a processor, a physical facility along which data is transferred.

byte
A binary character operated on as a unit and usually shorter than a computer word. (A) A string that consists of a number of bits, treated as a unit, and representing a character. A group of eight adjacent binary digits that represents one EBCDIC character.

C
Card-Verification Code (CVC)
See Card-Verification Value.

Card-Verification Value (CVV)
A cryptographic method, defined by VISA, for detecting forged magnetic-striped cards. This method cryptographically checks the contents of a magnetic stripe. This process is functionally the same as MasterCard’s Card-Verification Code (CVC) process.

Central Processor Assist for Cryptographic Functions (CPACF)
Implemented on all z890, z990, z9 EC, z9 BC, z10 EC and z10 BC processors to provide SHA-1 secure hashing.

channel
A path along which signals can be sent; for example, a data channel or an output channel. (A)

checksum
The sum of a group of data associated with the group and used for checking purposes. (T)

Chinese Remainder Theorem (CRT)
A mathematical theorem that defines a format for the RSA private key that improves performance.

Cipher Block Chaining (CBC)
A mode of encryption that uses the data encryption algorithm and requires an initial chaining vector. For encipher, it
exclusively ORs the initial block of data with the initial control vector and then enciphers it. This process results in the encryption both of the input block and of the initial control vector that it uses on the next input block as the process repeats. A comparable chaining process works for decipher.

**ciphertext**
Text that results from the encipherment of plaintext. Synonym for enciphered data. (D). See also plaintext.

**clear data**
Data that is not enciphered.

**clear key**
Any type of encryption key not protected by encryption under another key.

**cleartext**
Text that has not been altered by a cryptographic process. Synonym for plaintext. See also ciphertext.

**Common Cryptographic Architecture (CCA)**
The CCA API is the programming interface described in this document.

**Common Cryptographic Architecture: Cryptographic Application Programming Interface**
Defines a set of cryptographic functions, external interfaces, and a set of key management rules that provide a consistent, end-to-end cryptographic architecture across different IBM platforms.

**concatenation**
An operation that joins two characters or strings in the order specified, forming one string whose length is equal to the sum of the lengths of its parts.

**configuration**
The manner in which the hardware and software of an information processing system are organized and interconnected. (I) The physical and logical arrangement of devices and programs that constitutes a data processing system.

**console**
A part of a computer used for communication between the operator or maintenance engineer and the computer. (A)

**control block**
A storage area used by a computer program to hold control information. (I) Synonymous with control area.
The circuitry that performs the control functions such as decoding microinstructions and generating the internal control signals that perform the operations requested. (A)

**control vector (CV)**
In CCA, a 16-byte string that is exclusive-ORed with a master key or a key-encrypting key to create another key that is used to encipher and decipher data or data keys. A control vector determines the type of key and the restrictions on the use of that key.

**coprocessor**
In this document, the IBM 4765 PCIe and IBM 4764 PCI-X Cryptographic Coprocessors, generally also when using the CCA Support Program.

**Crypto Express2 Coprocessor**
An asynchronous cryptographic coprocessor available on the z890, z990, z9 EC, z9 BC, z10 EC and z10 BC.

**Crypto Express3 Coprocessor**
An asynchronous cryptographic coprocessor available on z10 EC and z10 BC.

**cryptographic adapter (4755 or 4758)**
An expansion board that provides a comprehensive set of cryptographic functions for the network security processor and the workstation in the TSS family of products.

**cryptographic coprocessor**
A microprocessor that adds cryptographic processing functions to specific z890, z990, z9 EC, z9 BC, z10 EC and z10 BC processors. The Cryptographic Coprocessor Feature is a tamper-resistant chip built into the processor board.

**cryptographic key data set (CKDS)**
A data set that contains the encrypting keys used by an installation. (D)

**cryptography**
The transformation of data to conceal its meaning.
In computer security, the principles, means, and methods for encrypting plaintext and decrypting ciphertext. (D)

**CUSP (Cryptographic Unit Support Program)**
The IBM cryptographic offering, program product 5740-XY6, using the channel-attached 3848. CUSP is no longer in service.

data
A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. Data includes constants, variables, arrays, and character strings. Any representations such as characters or analog quantities to which meaning is or might be assigned. (A)

data key or data-encrypting key
A key used to encipher, decipher, or authenticate data. Contrast with key-encrypting key.

**Data Encryption Algorithm (DEA)**
A 64-bit block cipher that uses a 64-bit key, of which 56 bits are used to control the cryptographic process and 8 bits are used for parity checking to ensure that the key is transmitted properly.

**Data Encryption Standard (DES)**
The National Institute of Standards and Technology Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data-encryption algorithm.

data translation key
A 64-bit key that protects data transmitted through intermediate systems when the originator and receiver do not share the same key.

data set
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

decipherer
To convert enciphered data in order to restore the original data. (T)

In computer security, to convert ciphertext into plaintext by means of a cipher system.

decrypt
To convert enciphered data into clear data. Synonym for decrypt. Contrast with encipher. (D)

decode
To convert data by reversing the effect of some previous encoding. (A) (I) In the CCA products, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard (DES). Contrast with encode and decipher.

decrypt
To decipher or decode. Synonym for decipher. Contrast with encrypt.

device driver
A program that contains the code needed to attach and use a device.

device ID
In the IBM 4765 and IBM 4764 CCA implementations, a user-defined field in the configuration data that can be used for any purpose the user specifies. For example, it can be used to identify a particular device, by using a unique ID similar to a serial number.

diagnostic
Pertaining to the detection and isolation of errors in programs, and faults in equipment.

directory server
A server that manages key records in key storage by using an Indexed Sequential Access Method.

digital signature
In public key cryptography, information created by using a private key and verified by using a public key. A digital signature provides data integrity and source nonrepudiation.

**Digital Signature Algorithm (DSA)**
A public key algorithm for digital signature generation and verification used with the Digital Signature Standard.

**Digital Signature Standard (DSS)**
A standard describing the use of algorithms for digital signature purposes. One of the algorithms specified is DSA (Digital Signature Algorithm).

domain
That part of a network in which the data processing resources are under common control. (T)
double-length key
A key that is 128 bits long. A key can be either double- or single-length. A single-length key is 64 bits long.

Electronic Code Book (ECB)
A mode of operation used with block cipher cryptographic algorithms in which plaintext or ciphertext is placed in the input to the algorithm and the result is contained in the output of the algorithm.

Elliptic Curve Cryptography (ECC)
A public-key process discovered independently in 1985 by Victor Miller (IBM) and Neal Koblitz (University of Washington). ECC is based on discrete logarithms. The algebraic structure of elliptic curves over finite fields makes it much more difficult to challenge at equivalent RSA key lengths.

encipher
To scramble data or to convert data to a secret code that masks the meaning of the data to unauthorized recipients. Synonym for encrypt. Contrast with decipher. See also encode.

enciphered data
Data whose meaning is concealed from unauthorized users or observers. See also ciphertext.

encode
To convert data by the use of a code in such a manner that reconversion to the original form is possible. (I) In the CCA implementation, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard. Contrast with decode. See also encipher.

encrypt
Synonym for encipher. (I) To convert cleartext into ciphertext. Contrast with decrypt.

erasable programmable read-only memory (EPROM)
A type of memory chip that can retain its contents without electricity. Unlike the programmable read-only memory (PROM), which can be programmed only once, the EPROM can be erased by ultraviolet light and then reprogrammed. (E)

Eurocheque International S.C. (ECI)
A financial institution consortium that has defined three PIN-block formats.

exit routine
In the CCA products, a user-provided routine that acts as an extension to the processing provided with calls to the security API.

exit
To execute an instruction within a portion of a computer program in order to terminate the execution of that portion. Such portions of computer programs include loops, subroutines, modules, and so on. (T)
A user-written routine that receives control from the system during a certain point in processing - for example, after an operator issues the START command.

exportable form
A condition a key is in when enciphered under an exporter key-encrypting key. In this form, a key can be sent outside the system to another system. A key in exportable form cannot be used in a cryptographic function.

EXPORTER key
In the CCA implementation, a type of DES key-encrypting key that can encipher a key at a sending node. Contrast with IMPORTER key.

exporter key-encrypting key
A 128-bit key used to protect keys sent to another system. A type of transport key.

feature
A part of an IBM product that can be ordered separately.

Federal Information Processing Standard (FIPS)
A standard published by the US National Institute of Science and Technology.

file
A named set of records stored or processed as a unit. (T)

financial PIN
A Personal Identification Number used to identify an individual in some financial transactions. To maintain the security of the PIN, processes and data structures have been adopted for creating, communicating, and verifying PINs used in financial transactions. See also Personal Identification Number.
Flash-Erasable Programmable Read-Only Memory (flash EPROM)
A memory that has to be erased before new data can be saved into the memory.

German Bank Pool (GBP)
A German financial institution consortium that defines specific methods of PIN calculation.

hashing
An operation that uses a one-way (irreversible) function on data, usually to reduce the length of the data and to provide a verifiable authentication value (checksum) for the hashed data.

header record
A record containing common, constant, or identifying information for a group of records that follows. (D)

host
In this publication, same as host computer or host processor. The machine in which the coprocessor resides. In a computer network, the computer that usually performs network-control functions and provides end-users with services such as computation and database access. (I)

importable form
A condition a key is in when it is enciphered under an importer key-encrypting key. A key is received from another system in this form. A key in importable form cannot be used in a cryptographic function.

IMPORTER key
In the CCA implementation, a type of DES key-encrypting key that can decipher a key at a receiving mode. Contrast with EXPORTER key.

importer key-encrypting key
A 128-bit key used to protect keys received from another system. A type of transport key.

initialize
In programming languages, to give a value to a data object at the beginning of its lifetime. (I) To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine. (A)

Integrated Cryptographic Service Facility (ICSF)
An IBM licensed program that supports the cryptographic hardware feature for the high-end System/390® processor running in a z/OS environment.

International Organization for Standardization (ISO)
An organization of national standards bodies established to promote the development of standards to facilitate the international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity.

initial chaining vector (ICV)
A 64-bit random or pseudo-random value used in the cipher block chaining mode of encryption with the data encryption algorithm.

input PIN-encrypting key
A 128-bit key used to protect a PIN block sent to another system or to translate a PIN block from one format to another.

installation exit
See exit.

jumper
A wire that joins two unconnected circuits on a printed circuit board.

key
In computer security, a sequence of symbols used with a cryptographic algorithm to encrypt or decrypt data.

key agreement
A key establishment procedure where the resultant secret keying material is a function of information contributed by two participants, so that no party can predetermine the value of the secret keying material independently from the contributions from the other parties.

key-encrypting key (KEK)
A key used for the encryption and decryption of other keys. Contrast with data-encrypting key. Also called a transport key.

key half
In the CCA implementation, one of the two DES keys that make up a double-length key.

key identifier
In the CCA implementation, a 64-byte variable which is either a key label or a key token.
key label
In the CCA implementation, an identifier of a key-record in key storage.

key storage
In the CCA implementation, a data file that contains cryptographic keys which are accessed by key label.

key token
In the CCA implementation, a data structure that can contain a cryptographic key, a control vector, and other information related to the key.

key output data set
A key generator utility program data set containing information about each key that the key generator utility program generates except an importer key for file encryption.

key part
A 32-digit hexadecimal value that you enter to be combined with other values to create a master key or clear key.

key part register
A register in the key storage unit that stores a key part while you enter the key part.

link
The logical connection between nodes including the end-to-end control procedures. The combination of physical media, protocols, and programming that connects devices on a network. In computer programming, the part of a program, in some cases a single instruction or an address, that passes control and parameters between separate portions of the computer program. (A) (I) To interconnect items of data or portions of one or more computer programs. (I) In SNA, the combination of the link connection and link stations joining network nodes.

linkage
The coding that passes control and parameters between two routines.

LPAR mode
The central processor mode that enables the operator to allocate the hardware resources among several logical partitions.

MAC generation key
A 64-bit or 128-bit key used by a message originator to generate a message authentication code sent with the message to the message receiver.

MAC verification key
A 64-bit or 128-bit key used by a message receiver to verify a message authentication code received with a message.

magnetic tape
A tape with a magnetizable layer on which data can be stored. (T)

make file
A composite file that contains either device configuration data or individual user profiles.

master key (MK, KM)
In computer security, the top-level key in a hierarchy of key-encrypting keys.

master key register
A register in the cryptographic coprocessors that stores the master key that is active on the system.

master key variant
A key derived from the master key by use of a control vector. It is used to force separation by type of keys on the system.

MD4 Message Digest 4. A hash algorithm.
MD5 Message Digest 5. A hash algorithm.

Message Authentication Code (MAC)
A number or value derived by processing data with an authentication algorithm, The cryptographic result of block cipher operations on text or data using a Cipher Block Chaining (CBC) mode of operation, A digital signature code.

migrate
To move data from one hierarchy of storage to another. To move to a changed operating environment, usually to a new release or a new version of a system.

Modification Detection Code (MDC)
In cryptography, a number or value that interrelates all bits of a data stream so that, when enciphered, modification of any bit in the data stream results in a new MDC.

multiple encipherment
The method of encrypting a key under a double-length key-encrypting key.
network
A configuration of data-processing devices and software programs connected for information interchange. An arrangement of nodes and connecting branches. (I)

new master key (NMK) register
A register in the key storage unit that stores a master key before you make it active on the system.

NOCV processing
Process by which the key generator utility program or an application program encrypts a key under a transport key itself rather than a transport key variant.

node
In a network, a point at which one-or-more functional units connect channels or data circuits. (I)

nonce
A time-varying value that has at most a negligible chance of repeating, such as a random value that is generated anew for each use, a timestamp, a sequence number, or some combination of these.

nonrepudiation
A method of ensuring that a message was sent by the appropriate individual.

offset
The process of exclusively ORing a counter to a key.

old master key (OMK) register
A register in the key storage unit that stores a master key that you replaced with a new master key.

operational form
The condition of a key when it is encrypted under the master key so that it is active on the system.

output PIN-encrypting key
A 128-bit key used to protect a PIN block received from another system or to translate a PIN block from one format to another.

panel
The complete set of information shown in a single image on a display station screen.

parameter
In the CCA security API, an address pointer passed to a verb to address a variable exchanged between an application program and the verb.

password
In computer security, a string of characters known to the computer system and a user; the user must specify it to gain full or limited access to a system and to the data stored within it.

PCI Cryptographic Coprocessor
The 4758 model 2 standard PCI-bus card supported on the field upgraded IBM S/390 Parallel Enterprise Server - Generation 5, the IBM S/390 Parallel Enterprise Server - Generation 6 and the IBM eServer zSeries.

PCI X Cryptographic Coprocessor
An asynchronous cryptographic coprocessor available on the IBM eServer zSeries 990 and IBM eServer zSeries 800.

Personal Account Number (PAN)
A Personal Account Number identifies an individual and relates that individual to an account at a financial institution. It consists of an issuer identification number, customer account number, and one check digit.

Personal Identification Number (PIN)
The 4-digit to 12-digit number entered at an automatic teller machine to identify and validate the requester of an automatic teller machine service. Personal identification numbers are always enciphered at the device where they are entered, and are manipulated in a secure fashion.

Personal Security card
An ISO-standard “smart card” with a microprocessor that enables it to perform a variety of functions such as identifying and verifying users, and determining which functions each user can perform.

PIN block
A 64-bit block of data in a certain PIN block format. A PIN block contains both a PIN and other data.

PIN generation key
A 128-bit key used to generate PINs or PIN offsets algorithmically.

PIN key
A 128-bit key used in cryptographic
functions to generate, transform, and verify the personal identification numbers.

**PIN offset**
For 3624, the difference between a customer-selected PIN and an institution-assigned PIN. For German Bank Pool, the difference between an institution PIN (generated with an institution PIN key) and a pool PIN (generated with a pool PIN key).

**PIN verification key**
A 128-bit key used to verify PINs algorithmically.

**plaintext**
Data that has not been altered by a cryptographic process. Synonym for cleartext. See also ciphertext.

**Power-On Self Test (POST)**
A series of diagnostic tests run automatically by a device when the power is turned on.

**private key**
In computer security, a key that is known only to the owner and used together with a public-key algorithm to decipher data. The data is enciphered using the related public key. Contrast with private key. See also public-key algorithm.

**procedure call**
In programming languages, a language construct for invoking execution of a procedure. (I) A procedure call usually includes an entry name and possible parameters.

**profile**
Data that describes the significant characteristics of a user, a group of users, or one-or-more computer resources.

**profile ID**
In the CCA implementation, the value used to access a profile within the CCA access-control system.

**protocol**
A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I) In SNA, the meanings of and the sequencing rules for requests and responses used to manage the network, transfer data, and synchronize the states of network components. A specification for the format and relative timing of information exchanged between communicating parties.

**Programmed Cryptographic Facility (PCF)**
An IBM licensed program that provides facilities for enciphering and deciphering data and for creating, maintaining, and managing cryptographic keys. (D) The IBM cryptographic offering, program product 5740-XY5, using software only for encryption and decryption. This product is no longer in service.

**public key**
In computer security, a key that is widely known, and used with a public-key algorithm to encrypt data. The encrypted data can be decrypted only with the related private key. Contrast with private key. See also public-key algorithm.

**Public Key Algorithm (PKA)**
In computer security, an asymmetric cryptographic process that uses a public key to encrypt data and a related private key to decrypt data. Contrast with Data Encryption Algorithm and Data Encryption Standard algorithm. See also Rivest-Shamir-Adleman algorithm.

**public key cryptography**
In computer security, cryptography in which a public key is used for encryption and a private key is used for decryption. Synonymous with asymmetric cryptography.

**Public-Key Cryptography Standards (PKCS)**
Specifications produced by RSA Laboratories in cooperation with secure system developers worldwide, for the purpose of accelerating the deployment of public-key cryptography. First published in 1991.

**Random access memory (RAM)**
A storage device into which data are entered and from which data are retrieved in a nonsequential manner.

**reason code**
A value that provides a specific result as opposed to a general result. Contrast with return code.

**record chaining**
When there are multiple cipher requests
and the output chaining vector (OCV) from the previous encipher request is used as the input chaining vector (ICV) for the next encipher request.

**Read-only memory (ROM)**
Memory in which stored data cannot be modified by the user except under special conditions.

**Resource Access Control Facility (RACF)**
An IBM licensed program that enables access control by identifying and verifying the users to the system, authorizing access to protected resources, logging detected unauthorized attempts to enter the system, and logging detected accesses to protected resources.

**retained key**
A private key that is generated and retained within the secure boundary of the PCI Cryptographic Coprocessor.

**return code**
A code used to influence the execution of succeeding instructions. (A) A value returned to a program to indicate the results of an operation requested by that program. In the CCA implementation, a value that provides a general result as opposed to a specific result. Contrast with reason code.

**Rivest-Shamir-Adleman (RSA) algorithm**
A process for public key cryptography that was developed by R. Rivest, A. Shamir, and L. Adleman.

**RS-232**
A specification that defines the interface between data terminal equipment and data circuit-terminating equipment, using serial binary data interchange.

**RS-232C**
A standard that defines the specific physical, electronic, and functional characteristics of an interface line that uses a 25-pin connector to connect a workstation to a communication device.

**Secure Electronic Transaction**
A standard created by Visa International and MasterCard for safeguarding payment card purchases made over open networks.

**Secure Hash Algorithm (SHA), FIPS 180**
A set of related cryptographic hash functions designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST). The first member of the family, published in 1993, is officially called SHA. However, today, it is often unofficially called SHA-0 to avoid confusion with its successors. Two years later, SHA-1, the first successor to SHA, was published. Four more variants have since been published with increased output ranges and a slightly different design: SHA-224, SHA-256, SHA-384, and SHA-512 (all are sometimes referred to as SHA-2).

**secure key**
A key that is encrypted under a master key. When using a secure key, it is passed to a cryptographic coprocessor where the coprocessor decrypts the key and performs the function. The secure key never appears in the clear outside of the cryptographic coprocessor.

**Secure Sockets Layer**
A security protocol that provides communications privacy over the Internet by allowing client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

**security**
The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

**security server**
In the CCA implementation, the functions provided through calls made to the security API.

**server**
On a Local Area Network, a data station that provides facilities to other data stations; for example, a file server, a print server, a mail server. (A)

**session**
In network architecture, for the purpose of data communication between functional units, all the activities that take place during the establishment, maintenance, and release of the connection. (l) The period of time during which a user of a terminal can communicate with an interactive system (usually, the elapsed time between logon and logoff).
SHA-1 (Secure Hash Algorithm 1, FIPS 180)
A hash algorithm required for use with the Digital Signature Standard.

SHA-2 (Secure Hash Algorithm 2, FIPS 180)
Four additional variants to the SHA family, with increased output ranges and a slightly different design: SHA-224, SHA-256, SHA-384, and SHA-512 (all are sometimes referred to as SHA-2).

SHA-224
One of the SHA-2 algorithms.

SHA-256
One of the SHA-2 algorithms.

SHA-384
One of the SHA-2 algorithms.

SHA-512
One of the SHA-2 algorithms.

single-length key
A key that is 64 bits long. A key can be single- or double-length. A double-length key is 128 bits long.

smart card
A plastic card that has a microchip capable of storing data or process information.

special secure mode
An alternative form of security that allows you to enter clear keys with the key generator utility program or generate clear PINs.

string
A sequence of elements of the same nature, such as characters, considered as a whole. (I)

subsystem
A secondary or subordinate system, usually capable of operating independently of, or asynchronously with, a controlling system. (I)

supervisor state
A state during which a processing unit can execute input/output and other privileged instructions. (D)

system administrator
The person at a computer installation who designs, controls, and manages the use of the computer system.

Systems Network Architecture (SNA)
An architecture that describes logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks. Note: The layered structure of SNA allows the ultimate origins and destinations of information, that is, the end users, to be independent of and unaffected by the specific SNA network services and facilities used for information exchange.

throughput
A measure of the amount of work performed by a computer system over a given period of time; for example, number of jobs per day. (A) (I) A measure of the amount of information transmitted over a network in a given period of time; for example, a network’s data-transfer-rate is usually measured in bits per second.

TLV
A widely used construct, Tag, Length, Value, to render data self-identifying. For example, such constructs are used with EMV smart cards.

token
In a Local Area Network, the symbol of authority passed successively from one data station to another to indicate the station is temporarily in control of the transmission medium. (I) A string of characters treated as a single entity.

Transaction Security System (TSS)
An IBM product offering including both hardware and supporting software that provides access control and basic cryptographic key-management functions in a network environment. In the workstation environment, this includes the 4755 Cryptographic Adapter, the Personal Security Card, the 4754 Security Interface Unit, the Signature Verification feature, the Workstation Security Services Program, and the AIX Security Services Program/6000. In the host environment, this includes the 4753 Network Security Processor and the 4753 Network Security Processor MVS™ Support Program.

transport key
A 128-bit key used to protect keys distributed from one system to another. A transport key can either be an exporter key-encrypting key, an importer key-encrypting key, or an ANSI key-encrypting key.
transport key variant
A key derived from a transport key by use of a control vector. It is used to force separation by type for keys sent between systems.

Unique key per transaction (UKPT)
A cryptographic process that can be used to decipher PIN blocks in a transaction.

user-exit routine
A user-written routine that receives control at predefined user-exit points.

user ID
User identification. A string of characters that uniquely identifies a user to the system.

utility program
A computer program in general support of computer processes. (I)

verb
A function that has an entry-point-name and a fixed-length parameter list. The procedure call for a verb uses the standard syntax of a programming language.

verification pattern
An 8-byte pattern calculated from the key parts you enter when you enter a master key or clear key. You can use the verification pattern to verify that you have entered the key parts correctly and specified a certain type of key.

virtual machine (VM)
A functional simulation of a computer and its associated devices. Each virtual machine is controlled by a suitable operating system. VM controls concurrent execution of multiple virtual machines on one host computer.

VISA
A financial institution consortium that has defined four PIN block formats and a method for PIN verification.

VISA PIN Verification Value (VISA PVV)
An input to the VISA PIN verification process that, in practice, works similarly to a PIN offset.

3621 A model of an IBM Automatic Teller Machine that has a defined PIN block format.

3624 A model of an IBM Automatic Teller Machine that has a defined PIN block format and methods of PIN calculation.
authentication_issuer_master_key_identifier parameter
PIN Change/Unblock verb 447
authentication_issuer_master_key_length parameter
PIN Change/Unblock verb 447
Automated Teller Machine (ATM) 45
AUTOSELECT option 9

B
base CCA services 78
battery indicator 78
battery-backed RAM size 78
Bellare-Rogaway 721
block chaining 286
block_size parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
Byte *
description 25

C
c-variable_encrypting_key_identifier parameter
Cryptographic Variable Encipher verb 141
calculation method
MAC padding method 709
Message Authentication Code (MAC) 709
Modification Detection Code (MDC) 700
X9.19 method 709
callable service 17
CBC processing rule 286, 289, 294, 300, 307
Decipher 288
Encipher 293
Symmetric Algorithm Decipher 298
Symmetric Algorithm Encipher 305
CCA functions 29
overview 29
CCA access control 4
CCA API 3, 8
CCA API build date 78
CCA API version 78
CCA application compile 22
Java 26
link 22
CCA DES-key verification 699
CCA description 4
CCA error log 78
CCA functional overview 4
CCA installation 757
CCA library 8, 17
location 8
CCA management 3
CCA master key 4
establishing 4
CCA nodes and resource control verb 77
CCA nodes and resource control verbs 53
CCA programming 3, 17
CCA RPM configure 760
download 758
files 758
groups 760
samples 760
uninstall 765
CCA sample program 743
Java 747
CCA services base 78
CCA software support 4
CCA system setup 753
CCA verb 3, 54, 75
CCA verb description 4
Central Processor Assist for Cryptographic Functions (CPACF) xviii, xxi, 12, 755
certificate parameter
Remote Key Export verb 508
certificate_length parameter
Remote Key Export verb 508
certificate_parms parameter
Remote Key Export verb 508
certificate_parms_length parameter
Remote Key Export verb 508
CEX2C xv, xix, xxi, 9, 63, 64, 77, 78, 767, 768
CEX2C coprocessors 4
CEX3 77
CEX3C xviii, 14, 78, 397, 768
CEX4C xv, 9, 77
CEXC3C 767
chain_data parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
chain_data_length parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
chzcrypt, command 754
CIPHER 134, 149, 164, 167, 184, 209, 224
cipher block chaining 33
cipher block chaining (CBC) 285
Cipher Block Chaining (CBC) 141
CIPHER key type 36
cipher_text parameter
Decipher verb 289
Encipher verb 294
Cipher-Block Chaining (CBC) mode xvii
ciphering methods 702
ciphertext 141
deciphering 285
ciphertext parameter
Cryptographic Variable Encipher verb 141
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
ciphertext_length parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
CLEAR 479
clear key 14, 16, 204, 285
definition 44
protecting 285
Clear Key Import (CSNBCKI) 129
format 129
JNI version 130
parameters 129
required commands 130
Clear PIN Encrypt (CSNBCPE) 405
format 405
JNI version 407
parameters 405
required commands 407
restrictions 407
Clear PIN Generate (CSNBPGN) 408
format 408
JNI version 411
parameters 408
related information 411
required commands 410
usage notes 410
Clear PIN Generate Alternate (CSNBCPA) 412
extraction rules 686
format 412
JNI version 416
parameters 412
required commands 415
clear_key parameter
Clear Key Import verb 129
Multiple Clear Key Import verb 131
clear_key_bit_length parameter
Key Generate2 verb 176
Key Token Build2 verb 214
clear_key_length parameter
Multiple Clear Key Import verb 131
clear_key_value parameter
Key Token Build2 verb 214
clear_master_key parameter
Key Storage Initialization verb 116
DES CIPHER keys
definition 36
DES CMK 78
DES cryptographic key verb 129
DES cryptography 29
DES encryption
56-bit 78
triple 286
DES encryption algorithm 289
DES encryption algorithm processing rule 294
DES engine 8
DES external key token format 613
DES hardware version 78
DES internal key token format 609
DES key 14
managing 127
questionable 122
translation 14
DES key flow 31
DES Key Record Create (CSNBKRC) 365
format 365
JNI version 366
parameters 365
related information 366
required commands 365
restrictions 365
DES Key Record Delete (CSNBKRD) 367
format 367
JNI version 368
parameters 367
related information 368
required commands 368
restrictions 368
DES Key Record List (CSNBKRL) 369
format 369
JNI version 371
parameters 372
related information 370
required commands 370
DES Key Record Read (CSNBKRR) 372
format 372
JNI version 373
parameters 372
related information 373
required commands 372
restrictions 372
DES Key Record Write (CSNBKRW) 374
format 374
JNI version 375
parameters 374
related information 375
required commands 375
restrictions 374
DES key storage 116, 345
DES key token 141
DES key-storage initialization 116
DES NMK 78
DES OMK 78
DES transport key 36
DES verb 54
device key 4
dexp 134, 209, 224
digital signature 3
using 468
Digital Signature Generate (CSNDDSG) 468
Digital Signature Generate (CSNDDSG) (continued)
format 468
JNI version 471
parameters 468
required commands 471
restrictions 471
digital signature verb 64
Digital Signature Verify (CSNDDSV) 473
format 473
JNI version 476
parameters 473
related information 476
required commands 476
restrictions 475
DIMP 134, 209, 224
directory server 8
Diversified Key Generate (CSNBDKG) 148
format 148
JNI version 152
parameters 149
required commands 151
usage notes 151
DKYGENKY 134, 149, 167, 209, 224
DKYGENKY key type 36
DKYL0 134, 209, 224
DKYL1 134, 209, 224
DKYL2 134, 209, 224
DKYL3 134, 209, 224
DKYL4 134, 209, 224
DKYL5 134, 209, 224
DKYL6 134, 209, 224
DKYL7 134, 209, 224
DMAC 134, 209, 224
DMKEY 134, 209, 224
DMPIN 134, 209, 224
DMV 134, 209, 224
DOUBLE 134, 167, 209, 224, 271
double length key 33, 34
double-length key
multiple decipherment 715
multiple encipherment 714
using 36
DPVR 134, 209, 224
DUKPT-BH 435
DUKPT-JP 435, 441
DUKPT-OP 435
dynamic RAM (DRAM) memory size 78
EC
EC Diffie-Hellman (CSNDEDH) 153
format 155
JNI version 162
parameters 155
required commands 161
restrictions 160
EC level
coprocessor 78
ECB processing rule 300, 307
ECC 484, 497
ECC key 468, 473
ECC key token 637
ECC-PAIR 489
ECC-PUBL 489
ECDSA xvii, 63, 468, 473
ECDSA algorithm 63
ECI-1 440
ECI-2 PIN block format 399, 686
ECI-3 PIN block format 399, 686
ECI-4 440
electronic code book (ECB) 286, 712
Electronic Code Book (ECB)
Symmetric Algorithm Decipher 298
Symmetric Algorithm Encipher 305
Elliptic Curve Cryptography (ECC) xvii, 21, 36, 117
key token 63, 64, 65, 66, 68, 116, 348, 468, 473, 478, 479, 484, 488, 489
Elliptic Curve Digital Signature Algorithm (ECDSA) xvii, 63, 468, 473
EMV 2000 149
EMV MAC smart card standard 710
EMVMAC 323, 327
EMVMACD 323, 327
ENC-ZERO 195, 196, 200, 204
ENCIPHER 134, 164, 167, 184, 209, 224
Encipher (CSNBENC) 293
format 294
JNI version 297
parameters 294
required commands 297
restrictions 297
ENCIPHER key type 36
Encipher processing rule 293
enciphered_key parameter
Symmetric Key Export verb 265
Symmetric Key Import verb 280
enciphered_key_length parameter
Symmetric Key Export verb 265
Symmetric Key Import verb 280
enciphered_text parameter
Secure Messaging for Keys verb 453
Secure Messaging for PINs verb 457
crypt zeros AES-key verification 700
crypt zeros DES-key verification 700
crypt key
definition 44
Encrypted PIN Generate (CSNBEPG) 430
format 430
JNI version 433
parameters 430
required commands 433
restrictions 433
Encrypted PIN Translate (CSNBPTR) 397, 435
extraction rules 687
format 435
JNI version 440
parameters 435
required commands 438
usage notes 440
Encrypted PIN Verify (CSNPBVR) 397, 441
extraction rules 686
format 441
JNI version 445
parameters 441
related information 445
required commands 444
encrypted_PIN_block parameter
Clear PIN Encrypt verb 405
Clear PIN Generate Alternate verb 412
Encrypted PIN Generate verb 430
Encrypted PIN Verify verb 441
encryption algorithm processing rule
AES 300, 307
DES 289, 294
encryption_issuer_master_key_identifier parameter
PIN Change/Unblock verb 447
encryption_issuer_master_key_length parameter
PIN Change/Unblock verb 447
ENH-ONLY 131, 134, 149, 209, 218, 224, 241, 271, 276
entry point 18
entry point name
prefix 18
entry-point names 17
Environment Identifier (EID) 78
environment variable 758
CSU_DEFAULT_ADAPTER 110, 113
CSU_HCPUACLR 13, 335
CSU_HCPUAPRT 13
CSUAESDS 346, 355, 357, 360, 362, 770
CSUAESLD 357, 770
CSUCACHE 9
CSUDESDS 346, 352, 365, 367, 369, 372, 374
CSUPKADS 346, 376, 378, 380, 383, 385
key storage 767, 770
list 770
PATH 25
EPINGEN 134, 209, 224
EPINGENA 134, 209, 224
EPINGENA key subtype 36
EPINVER 134, 209, 224
Europay padding rule 323
EVEN 260, 262
even parity 187, 260, 262
EX 167, 271
EX key form 663
EXEX 134, 167, 209, 224
EXEX key form 665
exit_data parameter 20
exit_data_length parameter 20
expiration_date parameter
CVV Generate verb 417
CVV Verify verb 426
EXPORT 134, 209, 224
exportability parameter
TR31 Key Token Parse verb 579
exportable key
generating 663
exportable key form
definition 30
value 167
EXPORTER 134, 164, 167, 184, 209, 224, 239
exporter key encrypting key
any DES key 164
EXPORTER key type 36
exporter key-encrypting key 144
exporter_key_identifier parameter
Data Key Export verb 144
Key Export verb 164
EXTERNAL 209, 224
external key 204
external key token 21, 31, 67, 647
DES 613
PKA
RSA private 616
verbs 31
extra_data parameter
Remote Key Export verb 508
extra_data_length parameter
Remote Key Export verb 508
extraction rules, PIN 686
F
files
hikmNativeInteger.html 25
key storage 44
financial services verb 393
FIPS-RNT 124
FIRST 187, 191, 315, 319, 323, 327, 333, 342
flash EPROM memory
size 78
form parameter
Random Number Generate verb 260
format control 402
formats
PIN 49
formatting hashes and keys 720
functional overview, CCA 4
G
GBP-PIN 134, 209, 224, 430, 441
GBP-PIN algorithm 441
GBP-PINO 134, 209, 224
GENERATE 196, 200, 204, 214, 462
generated_key_identifier parameter
Diversified Key Generate verb 149
Key Generate verb 167
generated_key_identifier_1 parameter
Diversified Key Generate verb 149
Key Generate verb 167
generated_key_identifier_1_length parameter
Key Generate2 verb 176
generated_key_identifier_1_length parameter
Key Generate2 verb 176
generated_key_identifier_2 parameter
Key Generate verb 167
generated_key_identifier_2_length parameter
Key Generate2 verb 176
generated_key_identifier_2_length parameter
Key Generate2 verb 176
generated_key_token parameter
PKA Key Generate verb 479
generated_key_token_length parameter
PKA Key Generate verb 479
generating_key_identifier parameter
Diversified Key Generate verb 149
German Banking Pool PIN
algorithm 688
GET-UDX 78, 93
H
hardware requirements xxi
Hardware Security Module (HSM) 45
hash algorithm 14
hash formatting 720
hash parameter
Digital Signature Generate verb 468
Digital Signature Verify verb 473
One-Way Hash verb 342
hash pattern 94, 96, 99, 101
hash_length parameter
Digital Signature Generate verb 468
Digital Signature Verify verb 473
One-Way Hash verb 342
Hashed Message Authentication code
(HMAC) xvii
hashing 49
hashing functions 49
hashing verb 54
HCPUACLR 110, 113
HCPUAPRT 110, 113
HEX-8 323, 327
HEX-9 323, 327
HEXDIGIT 412
HEXDIGIT PIN extraction method
keyword 400
hikmNativeInteger
description 25
hikmNativeInteger.html file 25
HMAC 192, 200, 214, 221, 256, 265, 280,
315, 319
HMAC algorithm 34
HMAC Generate (CSNBHMG) 315
format 315
JNI version 318
parameters 315
related information 318
required commands 317
restrictions 317
HMAC key 176
HMAC key token 640
HMAC key type 36
HMAC keys
definition 36
HMAC verb 54
HMAC Verify (CSNBHMG) 319
format 319
JNI version 321
parameters 319
required commands 321
usage notes 321
HMAC Verify (CSNBHMG)
related information 321
HMACVER key type 36
host CPU acceleration 289, 294, 323, 327
how to use this document xxii
I
IBM
contacting xxvii
IBM 3624 408, 441
IBM 4764 Crypto Express2 feature xxi
IBM 4765 Crypto Express3 feature xxi
IBM GBP 441
IBM-PIN 134, 209, 224, 430, 441
input_key_identifier parameter
Key Translate2 verb 241
input_key_length parameter
Key Translate2 verb 241
input_key_token parameter
Key Translate2 verb 241
input_PIN_encrypting_key_identifier parameter
Secure Messaging for PINs verb 457
input_PIN_encrypting_key_identifier parameter
Encrypted PIN Translate verb 435
Encrypted PIN Verify verb 441
input_PIN_profile parameter
Encrypted PIN Translate verb 435
Encrypted PIN Verify verb 441
Secure Messaging for PINs verb 457
intermediate PIN-block (IPB) 684
INTERNAL 209, 214, 224
internal key token 21, 67
AES 609
clear 611
definition 31
DES 609, 612
PKA
RSA private 624, 626, 627, 636
intrusion latch 78
IPB (intermediate PIN-block) 684
IPINENC 134, 164, 167, 184, 209, 224
IPINENC key type 36, 435
IPINENC key identifier parameter
Encrypted PIN Translate verb 435
Encrypted PIN Verify verb 441
Secure Messaging for PINs verb 457
Interbank PIN 54, 396, 408, 441
key
AES master key 36
AES transport 36
asymmetric master key 36
CIPHER 36
clear 14, 16, 44
count vector 30, 36
data key
export 144
importing 129
re-enciphering 144
DECIPHER 36
des exporter key-encrypting 36
des transport 36
double length 33
double-length 664, 665
ENCIPHER 36
encrypted 44, 176
form 30
generating
encrypted 167
HMAC 36
key-encrypting 36
MAC 36
master 14
multiple decipherment/
encipherment 712
NOCV importers and exporters 36
pair 664, 665
parity 129
PIN 36
PIN-encrypting key 435
protected 14, 15, 17
protecting 285
re-encipher 184
re-enciphering 164
separation 29
single-length 663, 664
symmetric master key 36
translated 14, 15, 16
triple length DES 34
type 36
VISA PVV 412
wrapping 15, 34

J
Java
data types 25
test point names 25
tested versions 25
Java Byte code
running 26, 27
Java interaction 3
Java Native Interface (JNI) xviii, 26
Byte code 26, 27

K
KAT 124
KDF in Counter Mode 34
kek_key_identifier parameter
Key Test Extended verb 205
KEK_key_identifier parameter
Control Vector Translate verb 137
Prohibit Export Extended verb 254
KEK_key_identifier_1 parameter
Key Generate verb 167
KEK_key_identifier_2 parameter
Key Generate verb 167
Key encrypting key (continued)
new 14
key encrypting key variant
definition 30
key export 256
Key Export (CSNBKEX) 164
format 164
JNI version 166
parameters 164
required commands 165
restrictions 165
usage notes 166
Key Export to TR31 (CSNBT31X) 524
format 526
JNI version 551
parameters 526
required commands 548
restrictions 548
key form 167, 663
combinations for a key pair 173
combinations with key type 173
definition 30
exportable 30
importable 30
operational 30
value 167
key form bits 673
key formats 609
key formatting 720
key functions 16
Key Generate (CSNBKGN) 167
format 167
JNI version 175
parameters 167
required commands 173
usage notes 173
using 663
Key Generate2 (CSNBKGN2) 176
format 176
JNI version 182
parameters 176
required commands 181
restrictions 181
usage notes 181
key generating key 148
definition 36
key identifier 16, 21
definition 67
PKA 67
key identifier parameter
Clear Key Import verb 129
Key Import (CSNBKIM) 184
format 184
JNI version 186
parameters 184
required commands 186
restrictions 185
usage notes 186
key label 8, 21, 67, 141, 164, 345, 346
key length 103, 105
key management 3, 45
PKA 66
key pair 173
key pair generation 711
key part 119, 120, 204
Key Part Import (CSNBKPI) 187
format 187
Key Part Import (CSNBKPI) (continued)
JNI version 190
parameters 187
required commands 189
restrictions 189
Key Part Import2 (CSNBKPI2) 191
format 191
JNI version 194
parameters 192
required commands 193
restrictions 193
usage notes 193
key part register 103, 105
key part register hash 103, 105
key record 44, 116
caching 9
key rule 196
key storage 8, 9, 119, 164, 187, 239, 345, 350, 767, 770
environment variables 346
Linux on IBM System z 348
key storage file 44, 351, 773
Key Storage Initialization (CSNBKSII) 116
format 116
JNI version 118
parameters 116
required commands 118
restrictions 117
key subtype list 36
specified by rule_array 36
Key Test (CSNBKYT) 195
format 195
JNI version 199
parameters 196
required commands 198
restrictions 198
Key Test Extended (CSNBKYTX) 204
format 204
JNI version 208
parameters 205
required commands 207
restrictions 207
usage notes 208
Key Test2 (CSNBKTYT2) 200
format 200
JNI version 203
parameters 200
required commands 203
restrictions 203
usage notes 203
key token 16, 21, 34, 130, 144, 146, 148, 149, 164, 167, 184, 206, 207, 202, 204, 241, 254, 256, 265, 280, 319, 352, 517, 647
AES 609
definition 31
DES
external 609, 613
internal 609
null 609, 615
DES internal 611, 612
ECC 637
Elliptic Curve Cryptography (ECC) 63, 65, 66, 68, 116, 348, 468, 473, 478, 479, 484, 488, 489
external 21, 31
HMAC 640
internal 21, 31, 68
null 31
operational 21
PKA 65
null 640
RSA 1024-bit modulus-exponent
private external 617
RSA 1024-bit private internal 626, 627, 636
RSA 2048-bit Chinese remainder
theorem private internal 634
RSA 4096-bit Chinese remainder
theorem private external 623
RSA 4096-bit modulus-exponent
private external 618
RSA private 616, 617
RSA private external 616
RSA private internal 624
RSA public 615
variable Modulus-Exponent 636
PKA external 68
verbs 31
Key Token Build (CSNBKTB) 36, 209
format 209
JNI version 213
parameters 209
usage notes 213
Key Token Build2 (CSNBKTB2) 214
format 214
JNI version 217
parameters 214
restrictions 217
Key Token Change (CSNBKTC) 218
format 218
JNI version 220
parameters 218
required commands 220
Key Token Change2 (CSNBKTC2) 221
format 221
JNI version 223
parameters 221
required commands 223
restrictions 222
Key Token Parse (CSNBKTP) 224
format 224
JNI version 227
parameters 224
usage notes 227
Key Token Parse2 (CSNBKTP2) 227
format 230
JNI version 238
parameters 230
required commands 238
usage notes 238
Key Translate (CSNBKTR) 239
format 239
JNI version 240
parameters 239
required commands 240
restrictions 240
Key Translate2 (CSNBKTR2) 241
format 241
JNI version 244
parameters 241
required commands 243
Index 809
Key Translate2 (CSNBKTR2) (continued)
restrictions 243
key translation cache 15
key type 21, 30, 134, 167, 184, 209, 663
list 294
key type 1 664, 665
key type 2 664, 665
key wrapping 176, 241
AES 33
definition 33
DES 33
electronic code book 33
enhanced CBC 34
key_a_identifier parameter
CVV Key Combine verb 422
key_a_identifier_length parameter
CVV Key Combine verb 422
key_b_identifier parameter
CVV Key Combine verb 422
key_b_identifier_length parameter
CVV Key Combine verb 422
key_bit_length parameter
EC Diffie-Hellman verb 155
key_block_length parameter
TR31 Key Token Parse verb 579
key_block_version parameter
TR31 Key Token Parse verb 579
key_check_parameters parameter
Remote Key Export verb 508
key_check_parameters_length parameter
Remote Key Export verb 508
key_check_value parameter
Remote Key Export verb 508
key_check_value_length parameter
Remote Key Export verb 508
key_encrypting_key_identifier parameter
Key Test2 verb 200
Restrict Key Attribute verb 256
Secure Messaging for Keys verb 453
Symmetric Key Generate verb 271
key_encrypting_key_identifier_1 parameter
Key Generate2 verb 176
key_encrypting_key_identifier_1_length parameter
Key Generate2 verb 176
key_encrypting_key_identifier_2 parameter
Key Generate2 verb 176
key_encrypting_key_identifier_2_length parameter
Key Generate2 verb 176
key_encrypting_key_identifier_length parameter
Key Test2 verb 200
Restrict Key Attribute verb 256
key_field_length parameter
Key Export to TR31 verb 526
key_form parameter
Key Generate verb 167
key_generation_data parameter
PIN Change/Unblock verb 447
key_generation_data_length parameter
PIN Change/Unblock verb 447
key_hash_algorithm parameter
Key Token Parse2 verb 230
key_identifiers parameter
Clear Key Import verb 129
Decipher verb 289
Diversified Key Generate verb 149
Encrypt verb 294
HMAC Generate verb 315
HMAC Verify verb 319
Key Part Import verb 187
Key Test Extended verb 205
Key Test verb 196
Key Test2 verb 200
Key Token Change verb 218
Key Token Change2 verb 221
MAC Generate verb 323
MAC Verify verb 327
PKA Key Token Change verb 497
Prohibit Export verb 252
Restrict Key Attribute verb 256
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
key_identifiers_length parameter
HMAC Generate verb 315
HMAC Verify verb 319
Key Test2 verb 200
Key Token Change2 verb 221
PKA Key Token Change verb 497
Restrict Key Attribute verb 256
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
key_label parameter
AES Key Record Create verb 352
AES Key Record Delete verb 355
AES Key Record List verb 357
AES Key Record Read verb 360
AES Key Record Write verb 362
DES Key Record Create verb 365
DES Key Record Delete verb 367
DES Key Record List verb 369
DES Key Record Read verb 372
DES Key Record Write verb 374
PKA Key Record List verb 380
Retained Key Delete verb 387
key_label_mask parameter
Retained Key List verb 390
key_labels parameter
Retained Key List verb 390
key_labels_count parameter
Retained Key List verb 390
key_length parameter
Key Generate verb 167
key_material_state parameter
Key Token Parse2 verb 230
key_name parameter
Key Token Build2 verb 214
Key Token Parse2 verb 230
Symmetric Key Import verb 280
key_name_1 parameter
Key Generate2 verb 176
key_name_1_length parameter
Key Generate2 verb 176
key_name_2 parameter
Key Generate2 verb 176
key_name_2_length parameter
Key Generate2 verb 176
key_name_length parameter
Key Token Build2 verb 214
key_offset parameter
Symmetric Key Import verb 280
key_offset_field_length parameter
Security Messaging for Keys verb 453
key_parms parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
key_parms_length parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
key_part parameter
Key Part Import verb 187
Master Key Process verb 120
key_storage_description parameter
Key Storage Initialization verb 116
key_storage_description_length parameter
Key Storage Initialization verb 116
key_storage_file_name parameter
Key Storage Initialization verb 116
key_storage_file_name_length parameter
Key Storage Initialization verb 116
key_token parameter
AES Key Record Create verb 352
AES Key Record Read verb 360
AES Key Record Write verb 362
DES Key Record Create verb 372
DES Key Record Read verb 374
Key Token Build verb 209
Key Token Parse verb 224
Key Token Parse2 verb 230
PKA Key Token Build verb 489
key_token_length parameter
AES Key Record Create verb 352
AES Key Record Read verb 360
AES Key Record Write verb 362
Key Token Build verb 209
Key Token Parse2 verb 230
PKA Key Token Build verb 489
key_type parameter
Control Vector Generate verb 134
Key Export verb 164
Key Import verb 184
Key Token Build verb 209
Key Token Parse verb 224
Key Token Parse2 verb 230
key_type_1 parameter
Key Generate verb 167
Key Generate2 verb 176
key_type_2 parameter
Key Generate verb 167
Key Generate2 verb 176
key_usage parameter
TR31 Key Token Parse verb 579
key_value parameter
Key Token Build verb 209
Key Token Parse2 verb 224

810 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Index 811

key_value_structure parameter
   PKA Key Token Build verb 489
key_value_structure_length parameter
   PKA Key Token Build verb 489
key_verification_pattern parameter
   Key Token Parse2 verb 230
key_verification_pattern_length parameter
   PKA Key Token Build verb 489
key_verification_pattern_type parameter
   Key Token Parse2 verb 230
key_version_number parameter
   Key Export to TR31 verb 526
   TR31 Key Token Parse verb 579
key_wrapping_method parameter
   Key Token Parse2 verb 230
   KEY-CLR 196, 214, 300, 307
   KEY-CLRD 196
   KEY-ENC 196, 204
   KEY-ENCD 196, 204
   key-encrypting key 36
   description 36
   exporter 144
   key-half processing 680
   KEY-KM 196, 204
   KEY-MGMT 489
   KEY-NKM 196, 204
   KEY-PART 134, 209, 224
   key-storage initialization 116
   key-token verification patterns 698
   key-verification 697
   Keyed-Hash Message Authentication Code (HMAC)
      generating 315
      verifying 319
   KEYGENKY 134, 149, 167, 209, 224
   KEYGENKY key type 36
   KEYIDENT 300, 307
   KEYL16 134, 167, 209, 224, 271
   KEYL24 167, 271
   KEYN32 167, 271
   KEYLN8 134, 167, 209, 224, 271
   keyvalue parameter
      PKA Encrypt verb 248
      keyvalue_length parameter
      PKA Encrypt verb 248
      keyword combinations 36
   KM-ONLY 489
L
label parameter
   PKA Key Record Create verb 376
   PKA Key Record Delete verb 378
   PKA Key Record Read verb 383
   PKA Key Record Write verb 385
   LABEL-DL 355, 367, 378
   LAST 187, 191, 315, 319, 323, 327, 333, 342
   legacy support 767
   Linux
      distributions supported  xix
      mixed configurations 773
   LMTD-KEK key subtype 36
   loading a master key 119
   local_enciphered_key_identifier parameter
      Symmetric Key Generate verb 271
   local_enciphered_key_identifier_length parameter
      Symmetric Key Generate verb 271
   lszcrypt, command 754
M
MAC 36, 48, 134, 149, 164, 167, 184, 209, 214, 224, 663
   length keywords 323, 327
   managing 48
   MAC Generate (CSNBMGN) 323
      format 323
      JNI version 326
      parameters 323
      related information 326
      required commands 326
      restrictions 325
   MAC key type 36
   MAC keys
      definition 36
      mac parameter
      HMAC Generate verb 315
      HMAC Verify verb 319
      MAC Generate verb 323
      MAC Verify verb 327
   MAC Verify (CSNBMVR) 327
      format 327
      JNI version 330
      methods 327
      parameters 327
      related information 330
      required commands 330
      restrictions 330
      usage notes 330
   MACD 166, 184
   MACLEN4 323, 327
   MACLEN6 323, 327
   MACLEN8 323, 327
   MACVER 36, 48, 134, 149, 164, 167, 184, 209, 224
   MACVER key type 36
   MACS
      mask array preparation 678
      mask_array_left parameter
      Control Vector Translate verb 137
      mask_array_right parameter
      Control Vector Translate verb 137
   MASTER 479
      master key 4, 14, 184, 773
      changing 350
      possible effect on internal key tokens 31
      enciphered key 184
      establishing 4
      Master Key coherence 9
      master key loading 119
      master key management 348
      Master Key Process (CSNBMKP) 119
      format 120
      JNI version 123
      parameters 120
   Master Key Process (CSNBMKP)
      (continued)
      Questionable DES keys 122
      required commands 121
      restriction 23
      restrictions 121
      master key register 119
      master key variant
         definition 30
      master key verification 196
      master_key_verification_pattern parameter
      Key Token Parse verb 224
      master-key loading 116
      master-key verification 697
      MasterCard card-verification code (CVC) 50, 393
      MasterCard padding rule 323
      masterkey_verify_parm parameter
      Key Token Build verb 209
      MD5 48, 342
      MDC Generate (CSNBMDG) 332
         format 333
         JNI version 341
         parameters 333
         related information 335
         required commands 335
         restrictions 334
      MDC keyed hash 335
      MDC parameter
      MDC Generate verb 333
      MDC-2 333
      MDC-4 196, 204, 333
      message
         authenticating 313
         message authentication
            definition 48
         Message Authentication Code (MAC) 48
            description 313
            generating 313, 323
            verifying 313, 327
         Message Authentication Code (MAC)
            calculation method 709
         microprocessor chip operating speed 78
         MIDDLE 187, 191, 315, 319, 323, 327, 333, 342
         MIN1PART 192
         MIN2PART 192
         MIN3PART 192
         miniboot firmware version 78
         MIXED 134, 209, 224
         MKVP parameter
         Key Token Parse verb 224
         mode parameter
         TR31 Key Token Parse verb 579
         modes of operation 285
         Modification Detection Code (MDC) 48, 313, 332, 700
         generate 314
         verify 314
         modular-exponentiation engine 8
         Modulus-Exponent format 245, 489, 500, 502, 616, 617, 626, 627, 636
         MRP 248
         multi-coprocessor functions 9
         multiple
            decipherment 712

Index 811
null key token 67, 149, 184
definition 31
format 615, 640
num_opt_blocks parameter
TR31 Key Import verb 556
TR31 Key Token Parse verb 579
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
NUM-DECT 78, 93
number of active coprocessors 78
odd parity 167, 187, 204, 260, 262
OKEK-AES 479
OKEK-DES 479
OKEYXLAT 36, 134, 164, 167, 184, 209, 224, 239
OKEYXLAT key type 36
OMK 348
OMK status 78
OMK status, AES 78
OMK status, ECC 78
One-Way Hash (CSNBOWH) 342
format 342
parameters 342
usage notes 344
ONLY 315, 319, 323, 327, 333, 342
OP 167, 271
OP key form 663
operating speed
microprocessor chip 78
operating system firmware name 78
operating system firmware version 78
operational key 130, 176, 252, 506
distribution 46
generating 663
operational key form
definition 30
value 167
operational key token 21
operational private key 64
OPEX 134, 167, 209, 224
OPEX key form 664
OPIM 134, 167, 209, 224
OPIM key form 664
OPINENC 134, 164, 167, 184, 209, 224
OPINENC key type 36, 435
OPK, object protection key 659
OPOP 167
OPOP key form 664
opt_block_data parameter
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
opt_block_data_length parameter
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
opt_block_id parameter
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
opt_block_ids parameter
TR31 Optional Data Read verb 588
opt_block_lengths parameter
TR31 Optional Data Read verb 588
opt_blocks parameter
Key Export to TR31 verb 526
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
opt_blocks_bfr_length parameter
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
opt_blocks_length parameter
Key Export to TR31 verb 526
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
opt_parameter1 parameter
Restrict Key Attribute verb 256
opt parameter1_length parameter
Restrict Key Attribute verb 256
opt_parameter2 parameter
Restrict Key Attribute verb 256
opt_parameter2_length parameter
Restrict Key Attribute verb 256
optional_data parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
optional_data_length parameter
Symmetric Algorithm Decipher verb 300
Symmetric Algorithm Encipher verb 307
other documentation xxiv
outbound_PIN_encrypting_key_identifier parameter
Encrypted PIN Generate verb 430
output chaining value (OCV) 703
output chaining vector (OCV) description 286
output_chaining_vector parameter
Secure Messaging for Keys verb 453
Secure Messaging for PINs verb 457
output_KEY_identifier parameter
Key Translate2 verb 241
output_KEY_key_identifier parameter
EC Diffie-Hellman verb 155
Key Translate verb 239
output_KEY_key_identifier_length
parameter
EC Diffie-Hellman verb 155
output_KEY_length parameter
Key Translate2 verb 241
output_key_identifier parameter
CVV Key Combine verb 422
EC Diffie-Hellman verb 155
TR31 Key Import verb 556
output_key_identifier_length parameter
CVV Key Combine verb 422
EC Diffie-Hellman verb 155
TR31 Key Import verb 556
output_key_length parameter
Key Translate2 verb 241
output_key_token parameter
Key Translate verb 239
Key Translate2 verb 241
output_PIN_data parameter
Secure Messaging for PINs verb 457
output_PIN_data parameter
PIN Change/Unblock verb 447
output_PIN_data_length parameter
PIN Change/Unblock verb 447
output_PIN_encrypting_key_identifier
parameter
Encrypted PIN Translate verb 435
output_PIN_length parameter
PIN Change/Unblock verb 447
output_PIN_message parameter
PIN Change/Unblock verb 447
output_PIN_message_length parameter
PIN Change/Unblock verb 447
output_PIN_profile parameter
Encrypted PIN Translate verb 435
PIN Change/Unblock verb 447
Secure Messaging for PINs verb 457
overlapped processing restrictions 8
OVERLAY 362, 385

812 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
personal identification number (PIN) (continued)
Clear PIN Generate Alternate
verb 412
Clear PIN Generate verb 408
definition 49
description 393
detailed algorithms 687
eventing 395
eventing algorithm 396, 435
extraction rules 686
formats 49
GBP PIN verification algorithm 692
generating 395, 408
from the translated PIN block 395
German Banking Pool PIN
algorithm 688
Interbank PIN generation
algorithm 695
keys 36
managing 49
PIN offset generation algorithm 689
PVV generation algorithm 693
PVV verification algorithm 694
translating 396
translation of, in networks 394
translating verb 435
using 393
verification verb 441
verifying 395, 441
VISA PIN algorithm 693
PIN 134, 209, 224
PIN block 395
PIN block format
3621 685
3624 685
additional names 440
ANS X9.8 683
detail 683
ECI-2 686
ECI-3 686
ISO-1 684
ISO-2 684
PIN extraction method keywords 400
values 399
VISA-2 685
VISA-3 685
PIN Change/Unblock (CSNBPCU) 446
format 446
JNI version 451
parameters 447
required commands 450
usage notes 451
PIN decimalization table identifier 93
PIN key subtype 36
PIN keys 36
PIN number 683
PIN profile 399
description 395, 441
PIN verification value (PVV) 395, 408
PIN verb 395
PIN_block_in parameter
Encrypted PIN Translant verb 435
PIN_block_out parameter
Encrypted PIN Translant verb 435
PIN_check_length parameter
Clear PIN Generate Alternate
verb 412
Clear PIN Generate verb 408
Encrypted PIN Verify verb 441
PIN_encrypting_key_identifier parameter
Clear PIN Encrypt verb 405
Secure Messaging for PINs verb 457
PIN_encrypting_key_identifier parameter
Clear PIN Generate Alternate
verb 412
PIN_generation_key_identifier parameter
Clear PIN Generate verb 408
Encrypted PIN Generate verb 430
PIN_generation_key_identifier parameter
Clear PIN Generate Alternate
verb 412
PIN_length parameter
Clear PIN Generate verb 408
Encrypted PIN Generate verb 430
PIN_offset parameter
Secure Messaging for PINs verb 457
PIN_offset_field_length parameter
Secure Messaging for PINs verb 457
PIN_profile parameter
Clear PIN Encrypt verb 405
Clear PIN Generate Alternate
verb 412
Encrypted PIN Generate verb 430
PIN_verifying_key_identifier parameter
Encrypted PIN Verify verb 441
PIN_encrypting key 435
PINBLOCK 412
PINBLOCK PIN extraction method
keyword 400
PINGEN 134, 164, 167, 184, 209, 224, 663
PINGEN key 663
PINGEN key type 36
PINGEN04 PIN extraction method
keyword 400
PINGEN12 PIN extraction method
keyword 400
PINGENn 412
PINVER 134, 164, 167, 184, 209, 224
PINVER key type 36
PKA CMK 78
PKA cryptographic key 477
PKA cryptography 63
PKA Decrypt (CSNDPKD) 245
format 245
JNI version 245
parameters 245
required commands 247
restrictions 247
usage notes 247
PKA Encrypt (CSNDPKE) 248
format 248
JNI version 250
parameters 248
required commands 250
restrictions 250
usage notes 250
PKA external key token 67, 68
PKA internal key token 68
PKA key 245, 248
PKA key algorithm 63

P

pad digit 403
format 402
pad_character parameter
Encipher verb 294
PADDIGIT 412
PADDIGIT PIN extraction method
keyword 400
padding method 293
PADEXIST 412
PADEXIST PIN extraction method
keyword 400
PADMDC-2 333
PADMDC-4 333
pair of keys 664, 665
PAN_data_out parameter
PAN_data_in parameter
PAN_data parameter
pair of keys 664, 665
PADMDC-4 333
PADEXIST PIN extraction method
keyword 400
PADMDC-2 333
PADMDC-4 333
pair of keys 664, 665
PAN_data parameter
Clear PIN Encrypt verb 405
Clear PIN Generate Alternate
verb 412
CVV Generate verb 417
CVV Verify verb 426
Encrypted PIN Generate verb 430
Encrypted PIN Verify verb 441
PAN_data_in parameter
Encrypted PIN Translate verb 435
PAN_block_in parameter
Encrypted PIN Translate verb 435
PAN-13 417, 426
PAN-14 417, 426
PAN-15 417, 426
PAN-16 417, 426
PAN-17 417, 426
PAN-18 417, 426
PAN-19 417, 426
panexe xx 11, 16
authorization 773
functions 773
functions not supported 774
utility 9, 36, 77, 350, 758, 760, 773,
774, 775, 777
parity of key 129
EVEN
form parameter 260
ODD
form parameter 260
part number
coprocessor 78
party_info parameter
EC Diffie-Hellman verb 155
party_info_length parameter
EC Diffie-Hellman verb 155
PATH 25
payload parameter
Key Token Parse2 verb 230
payload_length parameter
Key Token Parse2 verb 230
pending change stored in adapter 78
pending change user ID 78
personal account number (PAN) 50
for Encrypted PIN Translate 435
for Encrypted PIN Verify 441
personal identification number (PIN)
3624 PIN generation algorithm 687
3624 PIN verification algorithm 690
algorithm value 412, 441
algorithms 49, 396, 408
block format 396, 435

Index 813
PKA Key Generate (CSNDPKG) 478
   format 479
   JNI version 483
   parameters 479
   required commands 482
   restrictions 482
PKA key identifier 67
PKA Key Import (CSNDPKI) 484
   format 484
   JNI version 487
   parameters 484
   required commands 486
   restrictions 486
   usage notes 486
PKA key label 67
PKA key management 66
PKA key management verb 65
PKA Key Record Create
   (CSNDKRC) 376
   format 376
   JNI version 377
   parameters 376
   related information 377
   required commands 377
PKA Key Record Delete
   (CSNDKRD) 378
   format 378
   JNI version 379
   parameters 378
   related information 379
   required commands 379
PKA Key Record List (CSNDKRL) 380
   format 380
   JNI version 382
   parameters 380
   related information 382
   required commands 382
PKA Key Record Read
   (CSNDKRR) 383
   format 383
   JNI version 384
   parameters 383
   related information 384
   required commands 384
PKA Key Record Write
   (CSNDKRW) 385
   format 385
   JNI version 386
   parameters 385
   related information 386
   required commands 386
PKA key storage 116, 345
PKA key storage file 44
PKA key token 65, 67
   external 68
   record format
      RSA 1024-bit modulus-exponent
      private external 617
      RSA 1024-bit private internal 626, 627, 636
      RSA 2048-bit Chinese remainder
      theorem private internal 634
      RSA 4096-bit Chinese remainder
      theorem private external 623
      RSA 4096-bit modulus-exponent
      private external 618
      RSA private 616, 617
      RSA private external 616
PKA key token (continued)
   record format (continued)
      RSA private internal 624
      RSA public 615
      variable Modulus-Exponent 636
PKA Key Token Build (CSNDPKB) 488
   format 488
   JNI version 495
   parameters 489
PKA Key Token Change
   (CSNDKTC) 497
   format 497
   JNI version 499
   parameters 497
   required commands 499
PKA key token identifier 65
PKA key token sections 65
PKA Key Translate (CSNDPKT) 500
   format 500
   JNI version 503
   parameters 500
   required commands 502
   restrictions 502
   usage notes 503
PKA master key 63
PKA NMIK 78
PKA null key token 67
PKA OMIK 78
PKA Public Key Extract
   (CSNDPKX) 504
   format 504
   JNI version 505
   parameters 504
   usage notes 505
PKA verb 64, 69
   PKA_enciphered_keyvalue parameter
   PKA_encrypt verb 248
   PKA_enciphered_keyvalue_length parameter
      PKA Decrypt verb 245
      PKA Encrypt verb 248
PKA_key_identifier parameter
      PKA Decrypt verb 245
      PKA Encrypt verb 248
PKA_key_identifier_length parameter
      PKA Decrypt verb 245
      PKA Encrypt verb 248
PKA_private_key_identifier parameter
      PKA Digital Signature Generate verb 468
      PKA_private_key_identifier_length parameter
      PKA Digital Signature Generate verb 468
      PKA_public_key_identifier parameter
      PKA Digital Signature Verify verb 473
      PKA_public_key_identifier_length parameter
      PKA Digital Signature Verify verb 473
      PKA92 271, 276
      PKA92 key format and encryption
      process 718
      PKCS #1 formats 721
      PKCS 1.0 468, 473
      PKCS 1.1 468, 473
      PKCS-1.2 245, 248, 265, 271, 276
      PKCS-PAD
      Symmetric Algorithm Decipher 298
PKCS-PAD (continued)
   Symmetric Algorithm Encipher 305
   PKCS-PAD processing rule 300, 307
   PKCSOAEP 265, 271, 276
   PKOAEP2 265, 280
   plaintext
      encipher 141
      enciphering 285
      encrypt 141
   plaintext parameter
      Cryptographic Variable Encipher
      verb 141
   POST firmware version 78
   POST2 version
      coprocessor 78
      power-supply voltage 78
      privacy 48
      private external key token
         RSA 616
      private internal key token
         RSA 624, 626, 627, 636
      private key token
         RSA 616, 617
      private_KEY_key_identifier parameter
         EC Diffie-Hellman verb 155
      private_KEY_key_identifier_length parameter
      EC Diffie-Hellman verb 155
      private_key_identifier parameter
         EC Diffie-Hellman verb 155
      private_key_identifier_length parameter
         EC Diffie-Hellman verb 155
      private_key_name parameter
         PKA Key Token Build verb 489
      private_key_name_length parameter
      PKA Key Token Build verb 489
   problems, reporting xxvii
   procedure call 17
      processing a master key 119
      processing overlap 8
      processing rule
         ANSI X9.23 286, 289, 294
         CBC 286, 289, 294, 300, 307
         CUSP 286, 289, 294
         Decipher 288, 289
         description 286
         ECB 286, 300, 307
         Encipher 293, 294
         GBP-PIN 408
         IBM-PIN 408
         IBM-PINO 408
         INBK-PIN 408
         IPS 286, 289, 294
         PKCS-PAD 286, 300, 307
         recommendations for Encipher 294
         Symmetric Algorithm Decipher 298, 300
         Symmetric Algorithm Encipher 305, 307
         VISA-PVV 408
   profile
      description 4
      Prohibit Export (CSNBPEX) 252
      format 252
      JNI version 252
      parameters 252
      required commands 252
regeneration_data parameter
PKA Key Generate verb 479
regeneration_data_length parameter
PKA Key Generate verb 479
related publications xxiv
remote key distribution 45
Remote Key Export (CSNDRKX) 506
format 508
JNI version 515
parameters 508
required commands 514
restrictions 514
remote key loading
ACP 46
definition 46
new example 47
old example 47
reserved parameter
Control Vector Generate verb 134
Key Test2 verb 200
Key Token Build2 verb 214
Key Token Parse2 verb 230
reserved_1_length parameter
EC Diffie-Hellman verb 155
reserved_2 parameter
Key Token Parse verb 224
PKA Key Token Build verb 489
reserved_2_length parameter
EC Diffie-Hellman verb 155
PKA Key Token Build verb 489
reserved_3 parameter
Key Token Parse verb 224
PKA Key Token Build verb 489
reserved_3_length parameter
EC Diffie-Hellman verb 155
PKA Key Token Build verb 489
reserved_4 parameter
Key Token Parse verb 224
PKA Key Token Build verb 489
reserved_4_length parameter
EC Diffie-Hellman verb 155
PKA Key Token Build verb 489
reserved_5 parameter
Key Token Parse verb 224
PKA Key Token Build verb 489
reserved_5_length parameter
EC Diffie-Hellman verb 155
PKA Key Token Build verb 489
reserved_6 parameter
Key Token Parse verb 224
reserved_length parameter
Key Test2 verb 200
Key Token Build2 verb 214
Key Token Parse2 verb 230
resource_name parameter
Cryptographic Resource Allocate verb 110
Cryptographic Resource Deallocate verb 113
resource_name_length parameter
Cryptographic Resource Allocate verb 110
Cryptographic Resource Deallocate verb 113
Restrict Key Attribute (CSNBRKA) (continued)
parameters 256
related commands 258
restrictions 258
usage notes 258
RETURN 479
retained key 388
Retained Key Delete (CSNDRKD) 387
format 387
JNI version 389
parameters 387
related information 389
required commands 388
Retained Key List (CSNDRKL) 390
format 390
JNI version 392
parameters 390
related information 391
required commands 391
required_keys_count parameter
Restricted Key List verb 390
RETRKPR 187
return code 593
return_code 21
return_code parameter 20
returned_PVV parameter
Clear PIN Generate Alternate verb 412
returned_result parameter
Clear PIN Generate verb 408
revision history xv
RIPEMD-160 48
RKX key token 243
role
DEFAULT 4
description 4
role identifier 78
RPMD-160 342, 468
RSA 468, 473, 484, 497
RSA 1024-bit private internal key
token 626, 627, 636
RSA algorithm 63
RSA hardware version 78
RSA key 245, 248, 271, 276, 280, 468,
473, 478
RSA key generation 711
RSA key token sections 68
RSA key-pair generation 711
RSA private external Chinese remainder
theorem key token 623
RSA private external key token 616
RSA private external modulus-exponent
key token 618
RSA private external Modulus-Exponent
key token 617
RSA private internal Chinese remainder
theorem key token 634
RSA private internal key token 626
RSA private token 616, 617
RSA public token 615
RSA variable Modulus-Exponent
token 636
RSA_enciphered_key parameter
Symmetric Key Generate verb 271
Symmetric Key Import verb 276
rule_array parameter (continued)
Key Token Parse2 verb 230
Key Translate2 verb 241
MAC Generate verb 323
MAC Verify verb 327
Master Key Process verb 120
MDC Generate verb 333
Multiple Clear Key Import verb 131
One-Way Hash verb 342
PIN Change/Unblock verb 447
PKA Decrypt verb 245
PKA Encrypt verb 248
PKA Key Generate verb 479
PKA Key Import verb 484
PKA Key Import verb 484
PKA Key Import verb 484
PKA Key Record Create verb 376
PKA Key Record Delete verb 378
PKA Key Record List verb 380
PKA Key Record Read verb 383
PKA Key Record Write verb 385
PKA Key Token Build verb 489
PKA Key Token Change verb 497
PKA Key Translate verb 500
PKA Public Key Extract verb 504
Random Number Generate Long verb 262
Random Number Tests verb 124
Remote Key Export verb 508
Restrict Key Attribute verb 256
Retained Key Delete verb 387
Retained Key List verb 390
Secure Messaging for Keys verb 453
Secure Messaging for PINs verb 457
Symmetric Algorithm Decrypt verb 300
Symmetric Algorithm Encipher verb 307
Symmetric Key Export verb 265
Symmetric Key Generate verb 271
Symmetric Key Import verb 276, 280
TR31 Key Import verb 556
TR31 Key Token Parse verb 579
TR31 Optional Data Build verb 584
TR31 Optional Data Read verb 588
Transaction Validation verb 462
Trusted Block Create verb 518
rule_array_count parameter (continued)
DES Key Record Delete verb 367
Digital Signature Generate verb 468
Digital Signature Verify verb 473
Diversified Key Generate verb 149
EC Diffie-Hellman verb 155
Encipher verb 294
Encrypted PIN Generate verb 430
Encrypted PIN Translate verb 435
Encrypted PIN Verify verb 441
HMAC Generate verb 315
HMAC Verify verb 319
Key Export to TR31 verb 526
Key Generate2 verb 176
Key Part Import verb 187
Key Storage Initialization verb 116
Key Test Extended verb 205
Key Test verb 196
Key Test2 verb 200
Key Token Build verb 209
Key Token Build2 verb 214
Key Token Change verb 218
Key Token Change2 verb 221
Key Token Parse verb 224
rule_array_count parameter (continued)  
Trusted Block Create verb 518  
rule_id parameter  
Remote Key Export verb 508  

sample verb calls 743, 747  
S CCOMCRT 500  
SCOMME 500  
SCVISA 500  
SECMIG 134, 209, 224  
SECMIG key type 36  
secmsg_key_identifier parameter  
Secure Messaging for Keys verb 453  
Secure Messaging for PINs verb 457  
secure electronic transaction (SET) services 78  
secure messaging 51  
Secure Messaging for Keys (CSNBSKY) 453  
format 453  
JNI version 455  
parameters 453  
required commands 460  
usage notes 461  
Secure Messaging for PINs (CSNBSBPNI) 457  
format 457  
JNI version 461  
parameters 457  
required commands 460  
usage notes 461  
Secure Sockets Layer (SSL) 48  
security API 8, 18  
security API programming 17  
security API return code  
rule_array element 78  
security server 8, 14  
security_server_name parameter  
AES Key Record List verb 357  
DES Key Record List verb 369  
PKA Key Record List verb 380  
seed parameter  
Random Number Generate Long verb 262  
seed_length parameter  
Random Number Generate Long verb 262  
segmenting  
control keywords 323, 327  
selecting a coprocessor resource 110, 113  
SELFENC 457  
sequence_number parameter  
Clear PIN Encrypt verb 405  
Encrypted PIN Generate verb 430  
Encrypted PIN Translate verb 435  
sequences of verbs 51  
serial number  
adapter 78  
coprocessor 78  
service request 8  
service_code parameter  
CVV Generate verb 417  
CVV Verify verb 426  
SESS-XOR 149  
SET command 349  
SHA-1 48, 196, 204, 214, 265, 315, 319, 342, 468  
SHA-1 engine 8  
SHA-224 214, 315, 319, 342  
SHA-256 196, 204, 214, 265, 315, 319, 342, 468  
SHA-384 214, 265, 315, 319, 342, 468  
SHA-512 214, 265, 315, 319, 342, 468  
SHA2VP1 200  
short blocks 293  
SIG-ONLY 489  
signature_bit_length parameter  
Digital Signature Generate verb 468  
signature_field parameter  
Digital Signature Generate verb 468  
Digital Signature Verify verb 473  
signature_field_length parameter  
Digital Signature Generate verb 468  
Digital Signature Verify verb 473  
SIGSEGV error 15  
SINGLE 167, 271  
single-length key multiple decipherment 714  
multiple encipherment 713  
purpose 663, 664  
using 36  
SINGLE-R 167, 271  
size of battery-backed RAM 78  
size of dynamic RAM (DRAM) memory 78  
size of flash EPROM memory 78  
skeleton token 105  
skeleton token length 105  
skeleton_key_identifier parameter  
PKA Key Generate verb 479  
skeleton_key_identifier_length parameter  
PKA Key Generate verb 479  
SMKEY 134, 149, 209, 224  
SMPIN 134, 149, 209, 224  
SNA-SLE 705  
source_key_identifier parameter  
Data Key Export verb 144  
Key Export to TR31 verb 526  
Key Export verb 164  
Key Import verb 184  
PKA Key Import verb 484  
PKA Key Translate verb 500  
PKA Public Key Extract verb 504  
Remote Key Export verb 508  
Symmetric Key Export verb 265  
sym_encrypted_key_identifier parameter  
Key Export to TR31 verb 526  
PKA Key Import verb 484  
PKA Key Translate verb 500  
PKA Public Key Extract verb 504  
Remote Key Export verb 508  
Symmetric Key Export verb 265  
sym_key_token parameter  
Control Vector Translate verb 137  
Data Key Import verb 146  
Prohibit Export Extended verb 254  
sym_transport_key_identifier parameter  
PKA Key Translate verb 500  
sym_transport_key_identifier_length parameter  
PKA Key Translate verb 500  
SSL support 48  
STATAE 78  
statapka 78  
STATCARD 78  
STATCCA 78, 108  
STATCCAE 78  
STATCRD2 78  
STATDECT 93  
STATDIAG 78  
STATEID 78  
STATEXFT 78  
STATICS 78, 94  
adapter serial number 78  
serial number  
adapter 78  
verb_data field 78  
STATICS operational key parts output data format 94  
STATICSB 96  
adapter serial number 78  
serial number  
adapter 78  
verb_data field 78  
STATICSB operational key parts output data format 96  
STATICSE 78, 99  
adapter serial number 78  
serial number  
adapter 78  
verb_data field 78  
STATICX operational key parts output data format 99  
STATICX 78, 101  
STATICX operational key parts output data format 101  
STATKPR 78, 103  
STATKPR operational key parts output data format 103  
STATKPRL 78, 103, 104  
STATMOFN 78  
STATVKG 78  
STATVKG operational key parts output data format 105  
STATVKG 78, 105  
STATVKG operational key parts output data format 105  
sym_encrypted_key_identifier parameter  
Remote Key Export verb 508  
sym_transport_key_identifier_length parameter  
Remote Key Export verb 508  
SYM-MK 36, 63, 119, 120, 122, 196, 204  
Symmetric Algorithm Decipher (CSNBSAD) 298  
format 299  
JNI version 303  
parameters 300  
required commands 303  
restrictions 303  
Symmetric Algorithm Decipher processing rule 298
target_key_token_length parameter
(continued)
PKA Key Translate verb 500
target_keyvalue parameter
PKA Decrypt verb 245
target_keyvalue_length parameter
PKA Decrypt verb 245
target_public_key_token parameter
PKA Public Key Extract verb 504
target_public_key_token_length parameter
PKA Public Key Extract verb 504
target_transport_key_identifier parameter
PKA Key Translate verb 500
target_transport_key_identifier_length parameter
PKA Key Translate verb 500
TDES 706
TDES encryption 78
TDES-CBC 453, 457
TDES-DEC 149
TDES-ENC 149
TDES-MAC 323, 327
TDES-XOR 149, 447
tfdeemv2 149, 447
tfdeemv4 447
temperature 78
termology xxi
text parameter
HMAC Generate verb 315
HMAC Verify verb 319
MAC Generate verb 323
MAC Verify verb 327
MDC Generate verb 333
One-Way Hash verb 342
text_length parameter
Cryptographic Variable Encipher verb 141
Decipher verb 289
Encipher verb 294
HMAC Generate verb 315
HMAC Verify verb 319
MAC Generate verb 323
MAC Verify verb 327
MDC Generate verb 333
One-Way Hash verb 342
Secure Messaging for Keys verb 453
Secure Messaging for PINs verb 457
time of day 78
TIMEDATE 78
TKE access 78
TKE workstation xxi, 4, 63
TKESTATE 78
tlv_data parameter
Key Token Parse verb 230
tlv_data_length parameter
Key Token Parse verb 230
TOKEN 164, 166, 167, 184, 196, 204
token key type 36
token parameter
PKA Key Record Create verb 376
PKA Key Record Read verb 383
PKA Key Record Write verb 385
Token Validation Value (TVV) 611
token_data parameter
Key Token Build2 verb 214
token_data_length parameter
Key Token Build2 verb 214
token_length parameter
PKA Key Record Create verb 376
PKA Key Record Read verb 383
PKA Key Record Write verb 385
TOKEN-DL 355, 367, 378
tokens
wrapping method 78
TPK-ONLY 473
TR-31 200
TR31 Key Import (CSNB31I) 553
examples 555
format 555
JNI version 576
parameters 556
required commands 573
restrictions 573
special notes 554
TR31 Key Token Parse (CSNB31P) 578
format 579
JNI version 581
parameters 579
TR31 Optional Data Build
(CSNBT31O) 583
format 583
JNI version 585
parameters 584
restrictions 585
TR31 Optional Data Read
(CSNBT31R) 587
format 588
JNI version 590
parameters 588
tr31_key parameter
TR31 Key Token Parse verb 579
TR31 Optional Data Read verb 588
tr31_key_block parameter
Key Export to TR31 verb 526
TR31 Key Import verb 556
tr31_key_block_length parameter
Key Export to TR31 verb 526
TR31 Key Import verb 556
tr31_key_length parameter
TR31 Key Token Parse verb 579
TR31 Optional Data Read verb 588
trademarks 782
trailing short blocks 293
Transaction Validation (CSNBTRV) 462
format 462
JNI version 464
parameters 462
required commands 464
transaction_info parameter
Transaction Validation verb 462
transaction_info_length parameter
Transaction Validation verb 462
transaction_key_identifier parameter
Transaction Validation verb 462
transaction_key_identifier_length parameter
Transaction Validation verb 462
translated key 14, 15, 16
transport key 30, 146, 506, 647
transport_key_identifier parameter
PKA Key Generate verb 479
Remote Key Export verb 508
Symmetric Key Import verb 280
Trusted Block Create verb 518
transport_key_identifier_length parameter
Remote Key Export verb 508
Symmetric Key Import verb 280
transport_key_identifier parameter
Symmetric Key Export verb 265
transport_key_identifier_length parameter
Symmetric Key Export verb 265
triple-DES encryption 286
Triple-DES encryption 78
triple-length keys
multiple encipherment 716
multiple decipherment 717
trusted block 47, 517, 647
number representation 649
section format 650
Trusted Block Create (CSNDTBC) 517
format 518
JNI version 520
parameters 518
required commands 520
restrictions 520
trusted block integrity 649
trusted block sections 648
Trusted Key Entry (TKE) 63, 739, 773
overview 51
trusted_block_identifier parameter
Remote Key Export verb 508
Trusted Block Create verb 518
trusted_block_identifier_length parameter
Remote Key Export verb 508
Trusted Block Create verb 518
types of keys 36

U
UKPT 134, 209, 224
format 403
UKPTBOTH 435
UKPTFPIN 435, 441
UKPTOPIN 435
ule_id_length parameter
Remote Key Export verb 508
unwrap_kek_identifier parameter
Key Export to TR31 verb 526
TR31 Key Import verb 556
unwrap_kek_identifier_length parameter
Key Export to TR31 verb 526
TR31 Key Import verb 556
USE-CV 209
USECONF 131, 149, 187, 218, 241, 271, 276
user ID
pending change 78
user_associated_data parameter
Key Token Build2 verb 214
Key Token Parse2 verb 230
user_associated_data_1 parameter
Key Generate2 verb 176
user_associated_data_1_length parameter
Key Generate2 verb 176
user_associated_data_2 parameter
Key Generate2 verb 176
user_associated_data_2_length parameter
Key Generate2 verb 176
user_definable_associated_data_length parameter
PKA Key Token Build verb 489
user_definable_associated_data_length parameter
PKA Key Token Build verb 489
UTC time of day 78
utilities
ivp.e 9, 773
panel.exe 9, 16, 36, 77, 350, 777, 773, 774, 775, 777
PKA Key Token Build 488

V
validation_values parameter
Transaction Validation verb 462
validation_values_length parameter
Transaction Validation verb 462
value_1 parameter
Key Test verb 196
value_2 parameter
Key Test verb 196
variable Modulus-Exponent token
RSA 636
variable types 18
verb
access control 53, 723
AES 54
AES Key Record Create (CSNBAKRC) 352
AES Key Record Delete (CSNBAKRD) 355
AES Key Record List (CSNBAKRL) 357
AES Key Record Read (CSNBAKRR) 360
AES Key Record Write (CSNBAKRW) 362
CCA 54, 75
CCA node 53
CCA nodes and resource control 77
Clear Key Import (CSNBCKI) 129
Clear PIN Encrypt (CSNBCEP) 405
Clear PIN Generate (CSNBPGN) 408
Clear PIN Generate Alternate (CSNBPCEA) 412
common parameters 18, 20
Control Vector Generate (CSNCAVG) 134
Control Vector Translate (CSNCAVT) 137
Cryptographic Facility Query (CSUACFQ) 77, 78
Cryptographic Facility Version (CSUACVF) 108
Cryptographic Resource Allocate (CSUACRA) 110

verb (continued)
Cryptographic Resource Deallocate (CSUACRD) 113
Cryptographic Variable Encipher (CSNBCEV) 141
CVV Generate (CSNBCCG) 417
CVV Key Combine (CSNBCCK) 421
CVV Verify (CSNBCCV) 426
Data Key Export (CSNBDKX) 144
Data Key Import (CSNBDKM) 146
Decipher (CSNBDEC) 288
definition 17, 29
DES 54
DES cryptographic key 129
DES Key Record Create (CSNGBKRC) 365
DES Key Record Delete (CSNGBKRD) 367
DES Key Record List (CSNGBKRL) 369
DES Key Record Read (CSNGBKRR) 372
DES Key Record Write (CSNGBKRW) 374
description 18
digital signature 64
Digital Signature Generate (CSNDMSG) 468
Digital Signature Verify (CSNDSV) 473
Diversified Key Generate (CSNBDKG) 148
EC Diffie-Hellman (CSNDEDH) 153
Encipher (CSNENCG) 293
Encrypt PIN Generate (CSNBEPG) 430
Encrypt PIN Translate (CSNBPTR) 435
Encrypt PIN Verify (CSNBPVR) 441
event point name 18
financial services 393
format 18
hashing 54
HMAC 54
HMAC Generate (CSNBHMG) 315
HMAC Verify (CSNBHMR) 319
input/output (I/O) parameter 18
JNI version 18
Key Export (CSNBKEX) 164
Key Generate (CSNBKGN) 167
Key Generate2 (CSNBKGN2) 176
Key Import (CSNBRIM) 184
Key Part Import (CSNBKPI) 187
Key Part Import2 (CSNBKPI2) 191
key storage 345
Key Storage Initialization (CSNBKSI) 116
Key Test (CSNBKTYT) 195
Key Test Extended (CSNBKTXT) 204
Key Test2 (CSNBKTYT2) 200
Key Token Build (CSNBKTB) 209
Key Token Build2 (CSNBKTB2) 214
Key Token Change (CSNBKTC) 218
Key Token Change2 (CSNBKTC2) 221
Key Token Parse (CSNBKTP) 224

Index 819
verb (continued)
Key Token Parse2 (CSNBKTP2) 227
Key Translate (CSNBKTR) 239
Key Translate2 (CSNBKTR2) 241
MAC Generate (CSNBMMM) 252
MAC Verify (CSNBMVR) 242
Master Key Process (CSNBMKP) 119
MDC Generate (CSNBMDG) 332
Multiple Clear Key Import
(CSNCKM) 130
One-Way Hash (CSNBOWH) 342
parameter list 18
parameters 18
PIN Change/Unblock
(CSNBPCU) 446
PKA 63, 64, 69
PKA Decrypt (CSNDPKD) 245
PKA Encrypt (CSNDPKE) 248
PKA Key Generate (CSNDPKG) 478
PKA Key Import (CSNDPKI) 484
PKA Key management 65
PKA Key Record Create
(CSNDKRC) 376
PKA Key Record Delete
(CSNDKRD) 378
PKA Key Record List
(CSNDKRL) 380
PKA Key Record Read
(CSNDKRR) 383
PKA Key Record Write
(CSNDKRW) 385
PKA Key Token Build
(CSNDPKB) 488
PKA Key Token Change
(CSNDKTC) 497
PKA Key Translate (CSNDPKT) 500
PKA Public Key Extract
(CSNDPKE) 504
prefix 18
Prohibit Export (CSNBPEX) 252
Prohibit Export Extended
(CSNBPAX) 254
Random Number Generate
(CSNBRNG) 260
Random Number Generate Long
(CSNBRNL) 262
Random Number Tests
(CSUARNNT) 124
related information 18
Remote Key Export (CSNDKXX) 506
required commands 18
Restrict Key Attribute
(CSNBRAK) 255
restrictions 18
Retained Key Delete
(CSNDRKD) 387
Retained Key List (CSNRK) 390
Secure Messaging for Keys
(CSNBKSY) 453
Secure Messaging for PINs
(CSNBSPI) 457
sequence 51
Symmetric Algorithm Decrypt
(CSNBSDAD) 298
Symmetric Algorithm Encipher
(CSNBSAE) 305
verb (continued)
Symmetric Key Export
(CSNDSYX) 265
Symmetric Key Generate
(CSNDSYG) 271
Symmetric Key Import
(CSNDSY) 276
Symmetric Key Import2
(CSNDSY2) 280
Transaction Validation
(CSNBTRV) 462
Trusted Block Create
(CSNDBCT) 417
usage notes 18
variable types 18
verb_data field 78
verb_data parameter
Cryptographic Facility Query verb 78
for Cryptographic Facility Query 93
verb_data_length parameter
Cryptographic Facility Query verb 78
verbs
PIN 395
verification pattern 103
verification pattern 105, 119, 195, 204,
350, 697
verification_pattern parameter
Key Test Extended verb 205
Key Test2 verb 200
verification_pattern_length parameter
Key Test2 verb 200
VERIFY 196, 200, 204, 214, 462
version_data parameter
Cryptographic Facility Version
verb 108
version_data_length parameter
Cryptographic Facility Version
verb 108
Visa (EMV) padding rule 323
VISA card-verification value (CVV) 50,
393
VISA PVV 408
VISA PVV key 412
VISA-1 440
VISA-2 PIN block format 399, 685
VISA-3 PIN block format 399, 685
VISA-4 PIN block format 399
VISA-PVV 134, 209, 224
VISA-PVV algorithm 412, 441
VISA-PVV algorithm 412, 441
VISA-PVV4 441
VISA-PVV4 algorithm 441
WRAP-ECB 131, 134, 149, 187, 209, 218,
224, 241, 271, 276
WRAP-ENH 131, 134, 149, 187, 209, 218,
224, 241, 271, 276
WRAPMTHD 78
wrapping key 15
X
X.106 (CBC method) 703
X.19.27P 323, 327
X.9.31 468, 473
X.9.31 hash format 720
X.9.9.1 323, 327
X.9.9.1 keyword 323, 327
XLCATE 134, 209, 224
XLCATE-OK 449
XPORT 479
XPORT-OK 134, 209, 224
XPRT-SYM 214
XPRTRAASY 214
XPRTRUASY 214
Z
z/OS mixed configurations 773
z/VM xxi
z/VM guest 755
zf196 63
zcrypt installing 753
ZERO-PAD 245, 248, 265, 271, 276, 468,
473
Web site xv, xvii
who should use this document xix
wrap_kek_identifier parameter
Key Export to TR31 verb 526
TR31 Key Import verb 556
wrap_kek_identifier_length parameter
Key Export to TR31 verb 526
TR31 Key Import verb 556