Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Note!

Before using this information and the product it supports, be sure to read the general information in the "Notices" on page 557.


This edition applies to the Common Cryptographic Architecture (CCA) API, release 4.0.0, for Linux on IBM System z, and to all subsequent releases and modifications until otherwise indicated in new editions.

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
Business Support GmbH
Department 0790
Pascalstr. 100
70569 Stuttgart
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.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
**Contents**

**Figures** ........................................... xiii

**Tables** ........................................... xv

**About this document** ................................ xix

  - Revision history .................................. xix
  - Second edition, April 2010, CCA Support Program Release 4.0.0 ...... xix
  - Who should use this document ................. xx
  - Distribution-specific information ............ xx
  - Terminology ....................................... xx
  - Hardware requirements ......................... xx
  - How to use this document ...................... xxi
  - Where to find more information ............... xxiii
  - Related publications ............................ xxiii
  - Do you have problems, comments, or suggestions? .......... xxiii

**Part 1. IBM CCA programming** ............................ 1

  **Chapter 1. Introduction to programming for the IBM Common Cryptographic Architecture** ............... 3
  - Available Common Cryptographic Architecture verbs .................. 3
  - Common Cryptographic Architecture functional overview .......... 3
  - How application programs obtain service ............................ 7
  - Overlapped processing .................................. 8
  - CPACF support ....................................... 8
  - Environment variables that affect CPACF usage .................... 9
  - Access control points that affect CPACF protected key operations . 9
  - CPACF operation (protected key) .......................... 10
  - CCA library CPACF preparation at startup ....................... 12
  - Interaction between the 'default card' and use of Protected Key CPACF . 13
  - Security API programming fundamentals ......................... 13
  - Verbs, variables, and parameters .......................... 13
  - Commonly encountered parameters ........................ 15
  - How to compile and link CCA application programs ............... 17
  - Building Java applications to use with the CCA JNI ............ 18

  **Chapter 2. Using DES and AES cryptography and verbs** ............... 21
  - Functions of the DES and AES cryptographic keys .................. 21
    - Key separation ................................ 21
    - Master key variant ................................ 21
    - Transport key variant ............................ 22
    - Key forms ...................................... 22
    - Key token ...................................... 23
    - Control vector .................................. 24
    - Types of keys .................................. 24
  - Multi-coprocessor capabilities ............................. 30
  - Using the CCA node and master key management verbs ............ 31
    - Cryptographic Facility Query verb ................... 31
    - Cryptographic Facility Version verb ................ 31
    - Cryptographic Resource Allocate verb ............... 32
    - Cryptographic Resource Deallocate verb .......... 32
    - Key Storage Initialization verb .................... 32
    - Master Key Process verb .......................... 32

© Copyright IBM Corp. 2007, 2010
<table>
<thead>
<tr>
<th>Verb</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Number Tests verb</td>
<td>32</td>
</tr>
<tr>
<td>Generating and managing DES keys</td>
<td>32</td>
</tr>
<tr>
<td>Clear Key Import verb</td>
<td>32</td>
</tr>
<tr>
<td>Control Vector Generate verb</td>
<td>32</td>
</tr>
<tr>
<td>Control Vector Translate verb</td>
<td>32</td>
</tr>
<tr>
<td>Cryptographic Variable Encipher</td>
<td>32</td>
</tr>
<tr>
<td>Data Key Export verb</td>
<td>33</td>
</tr>
<tr>
<td>Data Key Import verb</td>
<td>33</td>
</tr>
<tr>
<td>Diversified Key Generate verb</td>
<td>33</td>
</tr>
<tr>
<td>Key Export verb</td>
<td>33</td>
</tr>
<tr>
<td>Key Generate verb</td>
<td>33</td>
</tr>
<tr>
<td>Key Import verb</td>
<td>33</td>
</tr>
<tr>
<td>Key Part Import verb</td>
<td>34</td>
</tr>
<tr>
<td>Key Test verb</td>
<td>34</td>
</tr>
<tr>
<td>Key Test Extended verb</td>
<td>34</td>
</tr>
<tr>
<td>Key Token Build verb</td>
<td>34</td>
</tr>
<tr>
<td>Key Token Change verb</td>
<td>34</td>
</tr>
<tr>
<td>Key Token Parse verb</td>
<td>34</td>
</tr>
<tr>
<td>Key Translate verb</td>
<td>34</td>
</tr>
<tr>
<td>Multiple Clear Key Import verb</td>
<td>34</td>
</tr>
<tr>
<td>Prohibit Export verb</td>
<td>34</td>
</tr>
<tr>
<td>Prohibit Export Extended verb</td>
<td>34</td>
</tr>
<tr>
<td>Random Number Generate verb</td>
<td>35</td>
</tr>
<tr>
<td>Random Number Generate Long verb</td>
<td>35</td>
</tr>
<tr>
<td>Verbs for managing CCA key storage files</td>
<td>35</td>
</tr>
<tr>
<td>AES Key Record Create verb</td>
<td>35</td>
</tr>
<tr>
<td>AES Key Record Delete verb</td>
<td>35</td>
</tr>
<tr>
<td>AES Key Record List verb</td>
<td>35</td>
</tr>
<tr>
<td>AES Key Record Read verb</td>
<td>35</td>
</tr>
<tr>
<td>AES Key Record Write verb</td>
<td>35</td>
</tr>
<tr>
<td>DES Key Record Create verb</td>
<td>35</td>
</tr>
<tr>
<td>DES Key Record Delete verb</td>
<td>35</td>
</tr>
<tr>
<td>DES Key Record List verb</td>
<td>36</td>
</tr>
<tr>
<td>DES Key Record Read verb</td>
<td>36</td>
</tr>
<tr>
<td>DES Key Record Write verb</td>
<td>36</td>
</tr>
<tr>
<td>Retained Key Delete verb</td>
<td>36</td>
</tr>
<tr>
<td>Retained Key List verb</td>
<td>36</td>
</tr>
<tr>
<td>Verbs for managing PKA key storage files</td>
<td>36</td>
</tr>
<tr>
<td>PKA Key Record Create verb</td>
<td>36</td>
</tr>
<tr>
<td>PKA Key Record Delete verb</td>
<td>36</td>
</tr>
<tr>
<td>PKA Key Record List verb</td>
<td>36</td>
</tr>
<tr>
<td>PKA Key Record Read verb</td>
<td>36</td>
</tr>
<tr>
<td>PKA Key Record Write verb</td>
<td>37</td>
</tr>
<tr>
<td>PKA Decrypt verb</td>
<td>37</td>
</tr>
<tr>
<td>PKA Encrypt verb</td>
<td>37</td>
</tr>
<tr>
<td>Enciphering and deciphering data</td>
<td>37</td>
</tr>
<tr>
<td>Decipher verb</td>
<td>37</td>
</tr>
<tr>
<td>Encipher verb</td>
<td>37</td>
</tr>
<tr>
<td>Symmetric Algorithm Decipher verb</td>
<td>37</td>
</tr>
<tr>
<td>Symmetric Algorithm Encipher verb</td>
<td>38</td>
</tr>
<tr>
<td>Verbs that support Secure Sockets Layer (SSL)</td>
<td>37</td>
</tr>
<tr>
<td>Message authentication code processing</td>
<td>38</td>
</tr>
<tr>
<td>MAC Generate verb</td>
<td>38</td>
</tr>
<tr>
<td>MAC Verify verb</td>
<td>39</td>
</tr>
<tr>
<td>Hashing functions</td>
<td>39</td>
</tr>
<tr>
<td>Key identifier for PKA key token</td>
<td>58</td>
</tr>
<tr>
<td>Key label</td>
<td>58</td>
</tr>
<tr>
<td>Key token</td>
<td>59</td>
</tr>
</tbody>
</table>

Summary of the PKA verbs | 60 |

Part 2. CCA verbs | 63 |

Chapter 4. Using the CCA nodes and access control verbs | 65 |
CCA nodes and access control verbs | 65 |
- Cryptographic Facility Query (CSUACFQ) | 65 |
- Cryptographic Facility Version (CSUACFV) | 87 |
- Cryptographic Resource Allocate (CSUACRA) | 89 |
- Cryptographic Resource Deallocate (CSUACRD) | 92 |
- Key Storage Initialization (CSNBKSI) | 95 |
- Master Key Process (CSNBMKP) | 98 |
- Random Number Tests (CSUARNT) | 103 |

Chapter 5. Managing DES and AES cryptographic keys | 105 |
DES and AES cryptographic key verbs | 105 |
- Clear Key Import (CSNBCKI) | 105 |
- Control Vector Generate (CSNBCVG) | 108 |
- Control Vector Translate (CSNBCVT) | 111 |
- Cryptographic Variable Encipher (CSNBCVE) | 115 |
- Data Key Export (CSNBDKX) | 118 |
- Data Key Import (CSNBDKM) | 120 |
- Diversified Key Generate (CSNBDKG) | 122 |
- Key Export (CSNBKEX) | 127 |
- Key Generate (CSNBKGN) | 130 |
- Key Import (CSNBKIM) | 138 |
- Key Part Import (CSNBKPI) | 142 |
- Key Test (CSNBKTYT) | 146 |
- Key Test Extended (CSNBKTYTX) | 151 |
- Key Token Build (CSNBKTB) | 157 |
- Key Token Change (CSNBKTC) | 161 |
- Key Token Parse (CSNBKTP) | 164 |
- Key Translate (CSNBKTR) | 169 |
- Multiple Clear Key Import (CSNBCKM) | 171 |
- PKA Decrypt (CSNDPKD) | 174 |
- PKA Encrypt (CSNDPKE) | 178 |
- Prohibit Export (CSNBPEX) | 182 |
- Prohibit Export Extended (CSNBPEXX) | 184 |
- Random Number Generate (CSNBRNG) | 186 |
- Random Number Generate Long (CSNBRNGL) | 188 |

Chapter 6. Protecting data | 191 |
DES modes of operation | 191 |
- Triple DES encryption | 191 |
- Processing rules | 191 |

AES Modes of Operation | 192 |
- Processing rules | 192 |

Verbs to encipher and decipher | 193 |
- Decipher (CSNBDEC) | 193 |
- Encipher (CSNBENC) | 198 |

Symmetric Algorithm Decipher (CSNBSAD) | 203 |
Symmetric Algorithm Encipher (CSNBSAE) | 210 |
Pad digit
Current key serial number
Financial services verbs
Clear PIN Encrypt (CSNBCPE)
Clear PIN Generate (CSNBPFGN)
Clear PIN Generate Alternate (CSNBCPA)
CVV Generate (CSNBCSG)
CVV Verify (CSNCSV)
Encrypted PIN Generate (CSNBEPPG)
Encrypted PIN Translate (CSNBEPT)
Encrypted PIN Verify (CSNBPV)
PIN Change/Unblock (CSNBPCU)
Secure Messaging for Keys (CSNBSKY)
Secure Messaging for PINs (CSNBSPI)
Transaction Validation (CSNBTRV)

Chapter 10. Using digital signatures
Digital signature verbs
Digital Signature Generate (CSNDDSG)
Digital Signature Verify (CSNDDSV)
MDC Generate (CSNBMDG)

Chapter 11. Managing PKA cryptographic keys
Verbs to manage PKA keys
PKA Key Generate (CSNDPKG)
PKA Key Import (CSNDPKI)
PKA Key Token Build (CSNDPKB)
PKA Key Token Change (CSNDKTC)
PKA Key Translate (CSNDKPT)
PKA Public Key Extract (CSNDPKX)
PKA Symmetric Key Export (CSNSSYX)
PKA Symmetric Key Generate (CSNSSYG)
PKA Symmetric Key Import (CSNSSYI)
Remote Key Export (CSNDRKX)
Trusted Block Create (CSNDTBC)

Appendix A. Return codes and reason codes
Return codes
Reason codes
Reason codes that accompany return code 0
Reason codes that accompany return code 4
Reason codes that accompany return code 8
Reason codes that accompany return code 12
Reason codes that accompany return code 16

Appendix B. Key token formats
Format of the internal AES key token
Internal AES key-token flag byte
Token validation value
Format of the clear key token
Format of the DES internal key token
DES external key token
DES null key token
Format of the RSA public key token
Format of RSA private external key tokens
RSA private key token, 1024-bit modulus-exponent external form
RSA private key token, 2048-bit Chinese remainder theorem external form 436
Format of the RSA private internal key token 437
RSA private key token, 1024-bit modulus-exponent internal form for cryptographic coprocessor feature 439
RSA private key token, 1024-bit modulus-exponent internal form for CEX3C 440
RSA private key token, 2048-bit Chinese remainder theorem internal form 441
RSA variable modulus-exponent token formats 443
PKA null key token 443
Trusted blocks 444
Remote key-loading 445
Trusted block sections 445
Trusted block integrity 446
Number representation in trusted blocks 446
Format of trusted block sections 446

Appendix C. Control vectors and changing control vectors with the Control Vector Translate verb 461
Control vector table 461
Control-vector-base bit maps 463
Key Form Bits, 'fff' 466
Specifying a control-vector-base value 466
Changing control vectors with the Control Vector Translate verb 471
Providing the control information for testing the control vectors 471
Mask array preparation 471
Selecting the key-half processing mode 473
When the target key-token CV is null 475
Control vector translate example 475

Appendix D. PIN formats and algorithms 477
PIN notation 477
PIN block formats 477
ANSI X9.8 477
ISO Format 1 478
ISO Format 2 478
ISO Format 3 478
VISA Format 2 479
VISA Format 3 479
IBM 4700 Encrypting PINPAD Format 479
IBM 3624 Format 479
IBM 3621 Format 479
ECI Format 2 479
ECI Format 3 480
PIN extraction rules 480
Encrypted PIN Verify verb 480
Clear PIN Generate Alternate verb 480
Encrypted PIN Translate verb 480
IBM PIN algorithms 481
3624 PIN Generation algorithm 481
German Banking Pool PIN Generation algorithm 482
PIN Offset Generation algorithm 483
3624 PIN Verification algorithm 484
German Banking Pool PIN Verification algorithm 486
VISA PIN algorithms 489
PVV Generation algorithm 489
PVV Verification algorithm 490
Interbank PIN Generation algorithm 490
## Appendix E. Cryptographic algorithms and processes

- Cryptographic key-verification techniques ........................................ 493
- Master-key verification algorithms .................................................. 493
- CCA DES-key verification algorithm .................................................. 494
- Encrypt zeros DES-key verification algorithm .................................... 495
- Modification Detection Code calculation ......................................... 495
- Ciphering methods ........................................................................ 497
  - General data-encryption processes ................................................. 497
  - Triple-DES ciphering algorithms .................................................... 501
- MAC calculation methods .............................................................. 504
  - ANSI X9.9 MAC ..................................................................... 504
  - ANSI X9.19 Optional Procedure 1 MAC ......................................... 505
  - EMV MAC ........................................................................ 505
  - ISO 16609 TDES MAC ............................................................ 505
- RSA key-pair generation ................................................................. 506
- Master-key-splitting algorithm ....................................................... 507
- Multiple decipherment and encipherment ........................................ 507
  - Multiple encipherment of single-length keys .................................... 508
  - Multiple decipherment of single-length keys .................................... 509
  - Multiple encipherment of double-length keys .................................... 510
  - Multiple decipherment of double-length keys .................................... 511
  - Multiple encipherment of triple-length keys .................................... 512
  - Multiple decipherment of triple-length keys .................................... 513
- PKA92 key format and encryption process ....................................... 514
- Formatting hashes and keys in public-key cryptography .................... 516
  - ANSI X9.31 hash format ............................................................ 516
  - PKCS #1 formats .................................................................... 516

## Appendix F. Access control points and verbs

- TKE Version 6.0 and higher ............................................................. 519

## Appendix G. Sample verb call routines

- Sample program in C ..................................................................... 525
- Sample program in Java .................................................................. 530

## Appendix H. Initial system set up tips

- Installing and loading the cryptographic device driver .................... 535
- zcrypt device driver usage ............................................................ 535
  - High resolution polling timer ....................................................... 535
- The sysfs interface ....................................................................... 536
- Running secure key under a z/VM guest ......................................... 537

## Appendix I. CCA installation instructions

- Before you begin ........................................................................... 539
- Download and install the RPM ....................................................... 539
  - Files in the RPM .................................................................... 539
  - Samples in the RPM ................................................................. 541
  - Groups in the RPM ................................................................. 541
  - Install and configure the RPM .................................................... 541
- Uninstall the RPM ....................................................................... 545

## Appendix J. Coexistence

- Legacy support ............................................................................. 547
- Concurrent installations ............................................................... 547
- Dual Support: Key storage interactions ......................................... 549
- Dual Support: TKE catcher can run in only one instance ............... 550
Appendix K. Utilities ........................................ 551
  The panel.exe utility ........................................ 551
    panel.exe syntax ........................................ 551
    panel.exe functions ...................................... 552
    Using panel.exe for key storage initialization .......... 553
    Using panel.exe for key storage re-encipher when changing the master key 555

Notices .............................................................. 557
  Programming interface information .......................... 558
  Trademarks ...................................................... 558

Index ............................................................... 561
Figures

1. CCA security API, access layer, and cryptographic engine ........................................... 4
2. CPACF ............................................................................................................................... 11
3. Control Vector Generate and Key Token Build CV keyword combinations .......................... 29
4. PKA key management ....................................................................................................... 57
5. Control vector base bit map (common bits and key-encrypting keys) ............................... 463
6. Control vector base bit map (data operation keys) ...................................................... 464
7. Control vector base bit map (PIN processing keys and cryptographic variable-encrypting keys) 465
8. Control vector base bit map (key generating keys) ...................................................... 466
9. Control Vector Translate Verb mask_array processing .............................................. 473
10. Control Vector Translate verb ....................................................................................... 474
11. ISO-3 PIN-block format .................................................................................................. 478
12. 3624 PIN generation algorithm ................................................................................... 482
13. GBP PIN generation algorithm .................................................................................... 483
14. PIN-Offset generation algorithm .................................................................................. 484
15. PIN verification algorithm ............................................................................................ 486
16. GBP PIN verification algorithm .................................................................................... 488
17. PVV generation algorithm ............................................................................................. 490
18. Triple-DES data encryption and decryption .................................................................. 498
19. Enciphering using the ANSI X3.106 CBC method ..................................................... 499
20. Deciphering using the CBC method ............................................................................. 499
21. Enciphering using the ANSI X9.23 method ................................................................ 500
22. Deciphering using the ANSI X9.23 method ................................................................ 501
23. Triple-DES CBC encryption process ............................................................................ 502
24. Triple-DES CBC decryption process ............................................................................ 502
25. EDE algorithm ................................................................................................................ 503
26. DED process .................................................................................................................. 504
27. MAC calculation method ............................................................................................. 505
28. Multiple encipherment of single-length keys .............................................................. 509
29. Multiple decipherment of single-length keys .............................................................. 510
30. Multiple encipherment of double-length keys ............................................................. 511
31. Multiple decipherment of double-length keys ............................................................. 512
32. Multiple encipherment of triple-length keys .............................................................. 513
33. Multiple decipherment of triple-length keys .............................................................. 514
34. Syntax, sample routine in C ......................................................................................... 526
35. Syntax, sample routine in Java ..................................................................................... 531
Tables

1. Descriptions of key types ............................................. 27
2. Key subtypes specified by the rule_array keyword ................. 28
3. Combinations of the verbs .............................................. 43
4. Summary of CCA nodes and access control verbs .................... 46
5. Summary of CCA AES, DES, and hashing verbs ...................... 46
6. Summary of key storage verbs ......................................... 50
7. Summary of PKA key token sections .................................. 56
8. Internal and external private RSA key token section identifiers . 59
9. Summary of PKA verbs .................................................. 60
10. Keywords for Cryptographic Facility Query control information .... 67
11. Cryptographic Facility Query information returned in the rule_array .... 68
12. Output data format for STATKPR operational key parts ............. 79
13. Output data format for STATICSA operational key parts ........... 79
14. Output data format for STATICSE operational key parts .......... 82
15. Output data format for STATICSX operational key parts .......... 84
16. Keywords for Master Key Process control information ............... 99
17. Keywords for Random Number Tests control information .......... 104
18. Keywords for Control Vector Translate control information ........ 112
19. Keywords for Diversified Key Generate control information ........ 123
20. Keywords for the Key Generate verb key_form parameter .......... 131
21. Key Generate - key lengths for each key type ...................... 132
22. Keywords for Key Generate, valid key types and key forms for a single key. 136
23. Keywords for Key Generate, valid key types and key forms for a key pair 136
24. Keywords for Key Part Import control information ................. 143
25. Verification pattern input and output ................................ 146
26. Keywords for Key Test control information .......................... 148
27. Keywords for Key Test Extended control information ............... 153
28. Keywords for Key Token Build Control Information ................. 158
29. Keywords for Key Token Change control information ............... 162
30. Keywords for Key Token Parse control information ................. 166
31. Keywords for Multiple Clear Key Import control information ....... 172
32. Keywords for PKA Decrypt control information ..................... 175
33. Keywords for PKA Encrypt control information ..................... 179
34. Keywords for Random Number Generate form parameter .......... 186
35. Keywords for Random Number Generate Long control information .. 189
36. Keywords for Decipher control information .......................... 195
37. Keywords for Encipher control information .......................... 201
38. Keywords for Symmetric Algorithm Decipher control information .. 205
39. Keywords for Symmetric Algorithm Encipher control information .. 212
40. Keywords for MAC Generate control information .................... 221
41. Keywords for MAC Verify control information ...................... 225
42. Keywords for One-Way Hash control information .................. 230
43. Valid symbols for the name token .................................... 234
44. Key labels that are not valid .......................................... 235
45. Keywords for AES Key Record Delete control information ......... 243
46. Keywords for AES Key Record Write control information .......... 250
47. Keywords for DES Key Record Delete control information ........ 255
48. Keywords for PKA Key Record Delete control information ........ 266
49. Keywords for PKA Key Record Write control information .......... 273
50. Format of a PIN profile .................................................. 284
51. Format values of PIN blocks .......................................... 284
52. PIN block format and PIN extraction method keywords ............. 285
53. Format of a pad digit ..................................................... 286
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pad digits for PIN block formats</td>
<td>286</td>
</tr>
<tr>
<td>Format of the Current Key Serial Number Field</td>
<td>287</td>
</tr>
<tr>
<td>Process rules for the Clear PIN Encryption verb</td>
<td>289</td>
</tr>
<tr>
<td>Process rules for the Clear PIN Generate verb</td>
<td>292</td>
</tr>
<tr>
<td>Array elements for the Clear PIN Generate verb</td>
<td>293</td>
</tr>
<tr>
<td>Array elements for Clear PIN Generate</td>
<td>293</td>
</tr>
<tr>
<td>Array elements for Clear PIN Generate Alternate</td>
<td>296</td>
</tr>
<tr>
<td>Keywords for Clear PIN Generate Alternate (first rule_array element)</td>
<td>297</td>
</tr>
<tr>
<td>Array elements for Clear PIN Generate Alternate, data_array (IBM-PINO)</td>
<td>298</td>
</tr>
<tr>
<td>Array elements for Clear PIN Generate Alternate, data_array (VISA-PVV)</td>
<td>298</td>
</tr>
<tr>
<td>Keywords for CVV Generate control information</td>
<td>301</td>
</tr>
<tr>
<td>Keywords for CVV Verify control information</td>
<td>305</td>
</tr>
<tr>
<td>Encrypted PIN Generate - Process rules</td>
<td>309</td>
</tr>
<tr>
<td>Array elements for Encrypted PIN Generate data_array parameter</td>
<td>310</td>
</tr>
<tr>
<td>Keywords for Encrypted PIN Generate control information</td>
<td>310</td>
</tr>
<tr>
<td>Keywords for Encrypted PIN Translate</td>
<td>315</td>
</tr>
<tr>
<td>Additional names for PIN formats</td>
<td>318</td>
</tr>
<tr>
<td>Keywords for Encrypted PIN Verify control information</td>
<td>321</td>
</tr>
<tr>
<td>Array elements for Encrypted PIN Verify data_array parameter</td>
<td>322</td>
</tr>
<tr>
<td>Array elements required by the process rule</td>
<td>322</td>
</tr>
<tr>
<td>Keywords for PIN Change/Unblock control information</td>
<td>325</td>
</tr>
<tr>
<td>Keywords for Secure Messaging for Keys control information</td>
<td>331</td>
</tr>
<tr>
<td>Keywords for Secure Messaging for PINs control information</td>
<td>335</td>
</tr>
<tr>
<td>Keywords for Transaction Validation control information</td>
<td>339</td>
</tr>
<tr>
<td>Values for Transaction Validation validation_values parameter</td>
<td>340</td>
</tr>
<tr>
<td>Keywords for Digital Signature Generate control information</td>
<td>344</td>
</tr>
<tr>
<td>Keywords for Digital Signature Verify control information</td>
<td>349</td>
</tr>
<tr>
<td>Keywords for MDC Generate control information</td>
<td>354</td>
</tr>
<tr>
<td>Keywords for PKA Key Generate control information</td>
<td>365</td>
</tr>
<tr>
<td>Keywords for PKA Key Token Build control information</td>
<td>371</td>
</tr>
<tr>
<td>PKA Key Token Build - Key value structure length maximum values</td>
<td>372</td>
</tr>
<tr>
<td>PKA Key Token Build - Key value structure elements</td>
<td>372</td>
</tr>
<tr>
<td>Keywords for PKA Key Token Change control information</td>
<td>378</td>
</tr>
<tr>
<td>Keywords for PKA Key Translate control information</td>
<td>381</td>
</tr>
<tr>
<td>Keywords for PKA Symmetric Key Export control information</td>
<td>387</td>
</tr>
<tr>
<td>Keywords for PKA Symmetric Key Generate control information</td>
<td>390</td>
</tr>
<tr>
<td>Keywords for PKA Symmetric Key Import control information</td>
<td>395</td>
</tr>
<tr>
<td>Keywords for Remote Key Export certificate_parms parameter</td>
<td>402</td>
</tr>
<tr>
<td>Keywords for Trusted Block Create control information</td>
<td>410</td>
</tr>
<tr>
<td>Return code values</td>
<td>413</td>
</tr>
<tr>
<td>Reason codes for return code 0</td>
<td>414</td>
</tr>
<tr>
<td>Reason codes for return code 4</td>
<td>414</td>
</tr>
<tr>
<td>Reason codes for return code 8</td>
<td>415</td>
</tr>
<tr>
<td>Reason codes for return code 12</td>
<td>425</td>
</tr>
<tr>
<td>Reason codes for return code 16</td>
<td>426</td>
</tr>
<tr>
<td>Internal AES key token format, version X'04'</td>
<td>429</td>
</tr>
<tr>
<td>Internal AES key-token flag byte</td>
<td>430</td>
</tr>
<tr>
<td>Internal clear key token format</td>
<td>430</td>
</tr>
<tr>
<td>Internal key token format</td>
<td>431</td>
</tr>
<tr>
<td>Format of DES external key tokens</td>
<td>432</td>
</tr>
<tr>
<td>Format of null key tokens</td>
<td>433</td>
</tr>
<tr>
<td>RSA Public Key Token</td>
<td>433</td>
</tr>
<tr>
<td>RSA private external key token basic record format</td>
<td>434</td>
</tr>
<tr>
<td>RSA private key token, 1024-bit modulus-exponent external format</td>
<td>435</td>
</tr>
<tr>
<td>RSA private key token, 2048-bit Chinese remainder theorem external format</td>
<td>436</td>
</tr>
<tr>
<td>RSA private internal key token basic record format</td>
<td>438</td>
</tr>
</tbody>
</table>
110. RSA private internal key token, 1024-bit modulus-exponent form for cryptographic coprocessor feature .................................. 439
111. RSA private internal key token, 1024-bit modulus-exponent form for CEX3C ........ 440
112. RSA private internal key token, 2048-bit Chinese remainder theorem internal format ..... 441
113. RSA variable modulus-exponent token formats ........................................... 443
114. Format of PKA null key tokens ........................................................................ 443
115. Trusted block sections and their use ............................................................... 445
116. Trusted block header ....................................................................................... 447
117. Trusted block trusted RSA public-key section (X'11') .......................................... 447
118. Trusted block rule section (X'12') .................................................................... 449
119. Summary of trusted block rule subsection ...................................................... 450
120. Transport key variant subsection (X'0001') of trusted block rule section (X'12') ........ 450
121. Transport key rule reference subsection (X'0002') of trusted block rule section (X'12') . 451
122. Common export key parameters subsection (X'0003') of trusted block rule section (X'12') 452
123. Source key rule reference subsection (X'0004') of trusted block rule section (X'12') . 453
124. Export key CCA token parameters subsection (X'0005') of trusted block rule section (X'12') 454
125. Trusted block key label (name) section (X'13') ................................................. 456
126. Trusted block information section (X'14') ........................................................ 456
127. Summary of trusted block information subsections .......................................... 457
128. Protection information subsection (X'0001') of trusted block information section (X'14') . 457
129. Activation and expiration dates subsection (X'0002') of trusted block information section (X'14') 458
130. Trusted block application-defined data section (X'15') .................................... 458
131. Default control vector values .......................................................................... 461
132. Versions of the MDC calculation method ....................................................... 461
133. MDC calculation procedures .......................................................................... 496
134. PKA96 clear DES key record ......................................................................... 496
135. Access Control Points and corresponding CCA commands ............................ 514
136. Verbs called by the sample routines ............................................................... 519
137. CCA groups .................................................................................................... 525

Tables xvii
About this document

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

This document describes how to use the verbs provided in the Common Cryptographic Architecture (CCA) API for Linux® on IBM® System z®. The CCA functions perform cryptographic operations using the IBM Crypto Express3 feature (CEX3C) in coprocessor mode. The CCA functions also perform some cryptographic operations using the IBM 4764 Crypto Express2 (CEX2C) feature in coprocessor mode. See “Concurrent installations” on page 547 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.

Revision history

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. See http://www.ibm.com/security/cryptocards for the supported environments and product ordering information.

For Linux for 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 Function (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 (CSNBKYTX)
  - MDC Generate (CSNBMDG)
  - 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 (CSNBSAD)
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 V 4.0.0 functions, a distribution of Linux that has support for the CEX3C feature is required. This feature is available with IBM System z10™ model GA3 and higher models.

A limited set of CCA V 4.0.0 functions can be used with a CEX2C feature. In order to make use of this limited set of CCA V 4.0.0 functions, a distribution of Linux that has support for the CEX2C feature is required.

Note that 31-bit support is not provided.

Terminology

These terms are used for the CCA features. For the remainder of this document, the short form (terms in bold) will be used.

- **CEX2C**: An IBM 4764 Crypto Express2 feature, configured in coprocessor mode.
- **CEX3C**: An IBM Crypto Express3 feature, configured in coprocessor mode.
- **CEX*C**: Either the CEX2C, CEX3C, 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 these minimum requirements:

- IBM System z10 model GA3
- One CEX3C, with one CEX3C adapter mapped to a single z/VM® image or single LPAR. The CEX3C must have CCA 4.0.0z or greater firmware loaded.
- If you plan to use a Trusted Key Entry (TKE) workstation, you must have a TKE V6.0 workstation in order to see supported CEX3Cs. They are not seen when using TKE V5 workstations.

This is the maximum supported hardware configuration:

- IBM System z10
- Two CEX3Cs, with four CEX3C adapters mapped to a single z/VM image or single LPAR.
  Additional CEX3C adapter features could function properly, but have not been a requirement or tested to date.

See [“Concurrent installations” on page 547](#) for details about a mixed environment of CEX2C and CEX3C.
To determine if a card is a CEX2C or CEX3C, use one of these methods, available with two utilities included in the CEX3C support program RPM, or your own custom implementation.

- The Cryptographic Facility Query verb (see Determining if a card is a CEX2C or CEX3C on page 65)
- The sysfs interface, the hwtype attribute (see The sysfs interface on page 536)

In order to use the CEX3C feature under z/VM versions 6.1, 5.3 and 5.4, you need to apply these APAR fixes:

<table>
<thead>
<tr>
<th>APAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM64656</td>
<td>Introduces CEX3C support.</td>
</tr>
<tr>
<td>VM64793</td>
<td>Introduces protected key CPACF support.</td>
</tr>
</tbody>
</table>

### How to use this document

For encryption, CCA supports Advanced Encryption Standard (AES), Data Encryption Standard (DES), and public key cryptography (PKA or RSA). These are very different cryptographic systems. Additionally, CCA provides APIs for generating and verifying message authentication codes (MACs), 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” describes the programming considerations for using the CCA DES verbs. It also explains the syntax and parameter definitions used in verbs. Concurrency is also discussed.
- Chapter 2, “Using DES and AES cryptography and verbs” gives an overview of AES and DES 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” introduces Public Key Algorithm (PKA) support and describes programming considerations for using the CCA PKA verbs, such as the PKA key token structure and key management.

Part 2, “CCA verbs,” on page 63 focuses on CCA verbs and includes the following chapters:

- Chapter 4, “Using the CCA nodes and access control verbs” describes using the CCA access control verbs.
- Chapter 8, “Key storage mechanisms” describes the use of key storage, key tokens, and associated verbs.
- Chapter 5, “Managing DES and AES cryptographic keys” 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 6, “Protecting data” describes the verbs for enciphering and deciphering data.
• **Chapter 7, “Verifying data integrity and authenticating messages”** 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 9, “Financial services”** 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 10, “Using digital signatures”** describes the PKA verbs that support using digital signatures to authenticate messages.

• **Chapter 11, “Managing PKA cryptographic keys”** describes the PKA verbs that generate and manage PKA keys.

The appendixes include the following information:

• **Appendix A, “Return codes and reason codes”** explains the return and reason codes returned by the verbs.

• **Appendix B, “Key token formats”** describes the formats for DES internal, external, and null key tokens and for PKA public, private external, and private internal key tokens containing Rivest-Shamir-Adleman (RSA) information. This appendix also describes the PKA null key token.

• **Appendix C, “Control vectors and changing control vectors with the Control Vector Translate verb,” on page 461** contains a table of the default control vector values that are associated with each key type and describes the control information for testing control vectors, mask array preparation, selecting the key-half processing mode, and an example of Control Vector Translate.

• **Appendix D, “PIN formats and algorithms,” on page 477** describes the PIN formats and algorithms.

• **Appendix E, “Cryptographic algorithms and processes,” on page 493** describes various ciphering and key verification algorithms, as well as the formatting of hashes and keys.

• **Appendix F, “Access control points and verbs”** lists which access control points correspond to which verbs.

• **Appendix G, “Sample verb call routines,” on page 525** contains sample verb call routines, both in C and Java, that illustrates the practical application of CCA verb calls.

• **Appendix H, “Initial system set up tips,” on page 535** includes tips to help you set up your system for the first time.

• **Appendix I, “CCA installation instructions,” on page 539** includes RPM installation, configuration, and uninstallation instructions.

• **Appendix J, “Coexistence,” on page 547** includes information about using CEX2C and CEX3C features in the same system, and other restrictions.

• **Appendix K, “Utilities,” on page 551** describes utilities, such as ivp.e and panel.exe.

• **Notices** contains notices, programming interface information, and trademarks.
Where to find more information

Other documents referenced in this document are:

- IBM Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference, SC40-1675

Related publications

- 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:

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. It explains how to use CCA nodes and AES, DES and PKA verbs.
Chapter 1. Introduction to programming for the IBM Common Cryptographic Architecture

This section introduces the IBM CCA application programming interface (API). The section explains basic concepts and describes how you can obtain cryptographic and other services from the CEX3C feature and CCA.

This chapter includes the following topics:
- Available CCA verbs
- CCA functional overview
- Security API programming fundamentals
- How to compile and link CCA application programs
- Legacy support and concurrency

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:
- Encrypt and decrypt information, typically using the DES algorithm in the Cipher Block Chaining (CBC) mode to enable data confidentiality
- Hash data to obtain a digest or process the data to obtain a message authentication code (MAC) that is useful in demonstrating data integrity
- Create and validate digital signatures to demonstrate both data integrity and form the basis for non-repudiation
- Generate, encrypt, translate, and verify finance industry personal identification numbers (PINs) and transaction validation codes with a comprehensive set of finance-industry-specific services
- Manage the various AES, DES and RSA (PKA) keys necessary to perform the above operations
- Control the initialization and operation of CCA
- Interact with the Java Native Interface (JNI). Some of the CCA verbs have a specific version that can be used for JNI work.

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

Figure 1 on page 4 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 programming language.
- 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 CEX3C 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 the CCA access-controls, administering DES and public-key cryptographic keys, 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 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 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

**CEX3C 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, AES key and RSA key-pair generation
- DES, AES, 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

**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 CEX3C 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 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 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 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 G, “Sample verb call routines,” on page 525 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**

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.

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. 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.

**CPACF support**

Central Processor Assist for Cryptographic Function (CPACF) support has these features:

- “Environment variables that affect CPACF usage” on page 9
- “Access control points that affect CPACF protected key operations” on page 9
- “CPACF operation (protected key)” on page 10
- “CCA library CPACF preparation at startup” on page 12
- “Interaction between the 'default card' and use of Protected Key CPACF” on page 13
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 of the CPACF for clear key operations and hashing algorithms is allowed if this variable is set 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 of the CPACF for protected key (translated secure key) operations is allowed if this variable is set 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 (CSNBMV)
- Symmetric Algorithm Decipher (CSNBSAD)
- Symmetric Algorithm Encipher (CSNBSAE)

Access control points that affect CPACF protected key operations

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

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 CEX3C 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 CEX3C. 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 CEX3C 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 CEX3C 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 CEX3C. 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 CEX3C 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 Function (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 11 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 CEX3C for the thread making the call.
Clear key: For keys that are not encrypted under the card master key, so-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 CEX3C 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 CEX3C 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**

1. An eligible CCA verb call (see lists in Access control points that affect CPACF protected key operations on page 9) 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 CEX3C 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.

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 551.

**Using keys with CPACF, clear key**

1. An eligible CCA verb call (see lists in Access control points that affect CPACF protected key operations on page 9) comes into the CCA library.
2. No CEX3C 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 Function (CPACF) by taking the following initialization steps:

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, for protected key operations a CEX3C 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 for use with the CPACF. Note also that for mixed CEX2C and CEX3C configurations, the allocated adapter for the thread must be a CEX3C because the service that translates the keys is not available on the CEX2C, for any CCA firmware version.

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. 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 section explains how each verb is described in the reference material and provides an explanation of the characteristics of the security API.

Each verb has an entry-point name and a fixed-length parameter list. The reference material 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

Each verb has a pseudonym (also called a general-language name) and an entry-point name (known as a computer-language name). The entry-point 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:
Generally, the AES and DES verbs

Public key cryptography verbs, including RSA

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.

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

Format: The format section for each verb lists the entry-point name on the first line in bold type. This is followed by the list of parameters for the verb. You must code all the parameters and in the order listed.

```plaintext
text-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 15 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,
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 4-byte (32-bit), signed, two's-complement binary number.

**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 16.

Restrictions: Any restrictions are noted.

Related information: Any related information is noted.

**Commonly encountered parameters**

Some parameters are common to all verbs, other parameters are used with many
of the verbs. This section describes several groups of these parameters:

- “Parameters common to all verbs”
- “The rule_array and other keyword parameters” on page 16
- “Key tokens, key labels, and key identifiers” on page 16

**Parameters common to all verbs**

The first four parameters (return_code, reason_code, exit_data_length, and
exit_data) are the same for all verbs. A parameter is an address pointer to the
associated variable in application program storage.

**return_code**

The return code specifies the general result of the verb. Appendix A, “Return
codes and reason codes” 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” 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 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 413. 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” for a detailed discussion of return codes and a complete list of all return 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 2 on page 28.

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 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 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' through 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' to 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 Support Program includes the C Language source code and the make file for a sample program. The file and its default distribution location is:

```
/opt/IBM/CEX3C/samples
```

Compile application programs that use CCA, and link the compiled programs to the CCA libraries. The libraries and their default distribution locations are:

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

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

**Note:** `/usr/lib64/libcsulccamk.so` contains the verb Master Key Process (CSNBMKP). 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 install 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 the following 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.

1. Loads a master key part to the new master-key register (AES-MK, SYM-MK or ASYM-MK)
2. Clears the new master-key register (AES-MK, SYM-MK or ASYM-MK)
3. Calls 'SET' on a master-key register that has had the 'LAST' key part loaded in order to activate the new master-key (AES-MK, SYM-MK or ASYM-MK)
Building Java applications to use with the CCA JNI

The CCA Support Program includes a CCA Java Native Interface (JNI). To illustrate how to use the CCA JNI to call CCA verbs, a sample module named `mac.java` is provided. The `mac.java` sample program calls the same CCA verbs as the sample C language program `mac.c`. See “Sample program in Java” on page 530.

The default distribution location of the sample code is:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Default distribution location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novell SUSE Linux</td>
<td>/opt/IBM/CEX3C/samples</td>
</tr>
<tr>
<td>Red Hat Linux</td>
<td>/opt/IBM/CEX3C/samples</td>
</tr>
</tbody>
</table>

These versions of Java are supported for JNI:

- Java 1.6.0 for Red Hat Enterprise Linux (RHEL)
- Java 1.4.2 for SUSE Linux Enterprise Server 10 and 11 (SLES 10 and 11) from Novell

These Java versions are the tested versions, and they were installed from the distribution CD or other authorized source for that distribution, and they were not customized in any way.

So that the CCA can access Java, do one of the following:

1. Add the path to the `java/javac` executable to the user's PATH environment variable, so that they can call the command without preconditions.
2. Create soft-links from the `java/javac` executables from wherever they are located to a directory that is in the user's PATH environment variable by default, such as `/usr/bin/`.

The Java entry points of CCA verbs are very 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 verb Key Generate, and CSNBKGNJ is the Java entry point for this verb. Where each verb is described in detail in Part 2, “CCA verbs,” on page 63, a section for the JNI interface is included.

Data types used in the JNI

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hikmNativeInteger</td>
<td>64-bit native signed integer (type long), matching the C interface</td>
</tr>
<tr>
<td>Byte *</td>
<td>General pointer type to unsigned byte</td>
</tr>
</tbody>
</table>

A file named `hikmNativeInteger.html` provides information about this class, and is located in the same directory as the `mac.java` file.

Building the Java Byte code

Issue the following command from the directory that contains the source code file `<yourfile>.java` to compile the program:

```
javac -classpath /opt/IBM/CEX3C/cmm/HIKM.zip <yourfile>.java
```
Notes:
1. The classpath option points to the HIKM.zip file because the hikmNativeInteger class and JNI verb front end classes are in this file.
2. The path shown for the HIKM.zip file is the default RPM installation location of that file.

For applications that also use the Master Key Process (CSNBMKP) verb: For security, the JNI interface for the Master Key Process (CSNBMKP) verb is also in the restricted access library libcsulccamk.so, and the Java class front end is implemented in a separate zip file. Therefore, to compile a Java Byte code file named <yourfile>.java, issue this command:

javac -classpath /opt/IBM/CEX3C/cnm/HIKM.zip:/opt/IBM/CEX3C/cnm/HIKMMK.zip <yourfile>.java

Notes:
1. The classpath option points to the HIKM.zip file because the hikmNativeInteger class and JNI verb front end classes are in this file.
2. The classpath option also points to the HIKMMK.zip file, where the Master Key Process (CSNBMKP) verb Java class front end for the JNI implementation is found.
3. The path shown for the HIKM.zip and HIKMMK.zip files is the default RPM installation location of that file.

Running the Java Byte code
Issue the following command from the directory that contains the Java Byte code file <yourfile>.class to run the program:

java -classpath /opt/IBM/CEX3C/cnm/HIKM.zip:.<yourfile>.class

Notes:
1. See the Notes in “Building the Java Byte code” on page 18 for notes on the HIKM.zip classpath.
2. Notice that '.' (period) is added to the class path so that Java can find <yourfile>.class in the current directory.

For applications that also use the Master Key Process (CSNBMKP) verb: You must also add the extra classpath option noted above for building to the run step:

java -classpath /opt/IBM/CEX3C/cnm/HIKM.zip:/opt/IBM/CEX3C/cnm/HIKMMK.zip:.<yourfile>.class

Notes:
1. See the Notes in “Building the Java Byte code” on page 18 for notes on the HIKM.zip classpath.
2. Notice that '.' (period) is added to the class path so that Java can find <yourfile>.class in the current directory.
Chapter 2. Using DES and AES cryptography and verbs

The CEX3C 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. It uses cryptography to accomplish 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

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

Functions of the DES and AES cryptographic keys

The CCA API provides functions to create, import, and export DES and AES keys. This section 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 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

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 24.

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**

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 26.

**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 (CSNBKDM) 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 DES and AES cryptographic keys” on page 21. See “Key forms and types used in the Key Generate verb” on page 43 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  →  OPERATIONAL  →  EXPORTABLE

Key token
A key token is a 64-byte field 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 5, “Managing DES and AES cryptographic keys.”
- Key Generate
- Key Import
- Clear Key Import
- Multiple Clear Key Import
- Key Record Read
- Key Token Build
- Data Key Import

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.

For debugging information, see Appendix B, “Key token formats” 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 5, “Managing DES and AES cryptographic keys.”
- Key Generate
- Key Export
- Data Key Export

For debugging information, see Appendix B, “Key token formats” 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.

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

**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 551.

**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 only to encrypt other AES 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 551.

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 551.

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.

CIPHER keys
These consist of CIPHER, ENCRYPTER, and DECRYPTER keys. They are single and double length DES keys for enciphering and deciphering data.

MAC keys
The MAC keys are single-length DES (64-bits - MAC, MACVER, DATAM, and DATAMV) and double-length DES (128 -bits - DATAM, MAC, MACVER, and DATAMV) 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 40 gives an overview of the PIN algorithms you need to know to write your own application programs.
Transport keys (or key-encrypting keys)

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

There are several types of 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, DATA2XLAT, 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).

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, and Key Translate.

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 1 on page 27 describes the key types.
### Table 1. Descriptions of key types

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPHER</td>
<td>Used only to encrypt or decrypt data. CIPHER keys cannot be used in the Encipher (CSNBENC) or Decipher (CSNBDEC) verbs. This is a single-length key. This is a single or double length key and can be used in the Encipher or Decipher verbs.</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>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 CEX3C.</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, ENCIPHER, 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>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></td>
<td>This is a single or double length key and can be used in the Decipher verb.</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>ENCIPHER</td>
<td>Used only to encrypt data. ENCIPHER keys cannot be used in the Decipher (CSNBDEC) verb. This is a single-length key.</td>
</tr>
<tr>
<td></td>
<td>This is a single or double length key and can be used in the Encipher verb.</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>Exporter key-encrypting key. Use this double-length key to convert a key from the operational form into exportable form.</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>Used to decrypt an input key in the Key Translate verb. This is a double-length key.</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>Importer key-encrypting key. Use this double-length key to convert a key from importable form into operational form.</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>MAC generation key. Use this single-length key to generate a message authentication code.</td>
</tr>
<tr>
<td></td>
<td>This is a single or double length key on a CEX3C.</td>
</tr>
<tr>
<td>MACVER</td>
<td>MAC verification key. Use this single-length key to verify a message authentication code.</td>
</tr>
<tr>
<td></td>
<td>This is a single or double length key on CEX3C.</td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>Used to encrypt an output key in the Key Translate verb. This is a double-length key.</td>
</tr>
</tbody>
</table>
Table 1. Descriptions of key types (continued)

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

Table 2 lists key subtypes passed in the rule_array keyword.

Table 2. Key subtypes specified by 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>ANSIX9.9</td>
<td>A MAC key that can be used for the ANSI X9.9 MAC calculation method, either for MAC Generate (CSNBMGN), MAC Verify (CSNBMVR), 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 (CSNBMVR), 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>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>
### 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.

---

**Figure 3. Control Vector Generate and Key Token Build CV keyword combinations**

---

Chapter 2. Using DES and AES cryptography and verbs 29
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.

Multi-coprocessor capabilities

Multi-coprocessor capabilities allow you to employ more than one coprocessor. When more than one coprocessor with CCA is installed, an application program can explicitly select which cryptographic resource (coprocessor) to use, or it can optionally employ the default coprocessor. To explicitly select a coprocessor, use the Cryptographic Resource Allocate verb. This verb allocates a coprocessor loaded with the CCA software. When a coprocessor is allocated, CCA requests are routed to it until it is deallocated.

To deallocate an allocated coprocessor, use the Cryptographic Resource Deallocate verb. When a coprocessor is not allocated (either before an allocation occurs or after the cryptographic resource is deallocated), requests are routed to the default coprocessor. To determine the number of CCA coprocessors installed, use the Cryptographic Facility Query verb with the STATCARD rule_array keyword. The verb returns the number of coprocessors running CCA software, which includes any coprocessors loaded with CCA user defined function (UDX) code.

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

- The Cryptographic Facility Query verb (see “Determining if a card is a CEX2C or CEX3C” on page 65).
- The sysfs interface, the hwtype attribute (see “The sysfs interface” on page 536).

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 application. As each coprocessor is evaluated, the CCA host associates the identifiers CRP01, CRP02, and so forth to the coprocessors with CCA. Coprocessors loaded with a UDX extension to CCA are also assigned a CRPnn identifier.

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.

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.
The commands `ivp.e` and `panel.exe -x` will tell you whether your cards are CEX3C or CEX2C, by calling the Cryptographic Facility Query verb for all available adapters. For details about these commands, see Appendix K, “Utilities,” on page 551.

When CEX2C and CEX3C cards are active in the same system, take note of these points:

- The CCA library will detect CEX2C and CEX3C 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 CEX3C instance found by the device driver.
- A user can specify the proper 'CRPnn' number to allocate and work with any card, however, CEX2C or CEX3C.

**Note:** The scope of the Cryptographic Resource Allocate and the Cryptographic Resource Deallocate verbs is to a thread. A multithreaded 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.

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 Crypto Facility Query verb (see “Cryptographic Facility Query” on page 65) with the STATCARD 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 551).

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

- The Cryptographic Facility Query verb (see “Determining if a card is a CEX2C or CEX3C” on page 65).
- The sysfs interface, the hwtype attribute (see “The sysfs interface” on page 536).

---

**Using the CCA node and master key management verbs**

The following verbs allow you to use the CCA node and master key management verbs.

**Cryptographic Facility Query verb**

This verb is used to retrieve information about the coprocessor and the CCA application program in that coprocessor.

**Cryptographic Facility Version verb**

This verb returns the Security Application Program Interface (SAPI) Version and build date.
Cryptographic Resource Allocate verb

This verb is used to allocate a specific CCA coprocessor for use by the thread or process, depending on the scope of the verb.

Cryptographic Resource Deallocate verb

This verb is used to deallocate a specific CCA coprocessor that is allocated by the thread or process, depending on the scope of the verb.

Key Storage Initialization verb

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.

Master Key Process verb

This verb operates on the three master-key registers (new, current, and old). Use this verb to 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, 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.

Random Number Tests verb

This verb 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.

Generating and managing DES keys

The CCA API provides the following verbs that support key management for DES keys.

Clear Key Import verb

This verb imports a 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.

Control Vector Generate verb

This verb builds a control vector from keywords specified by the key_type and rule_array parameters.

Control Vector Translate verb

This verb changes the control vector used to encipher an external DES key.

Cryptographic Variable Encipher

This verb encrypts plaintext using a CVARENC key to produce ciphertext using the Cipher Block Chaining (CBC) method.
**Data Key Export verb**

This verb 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.

**Data Key Import verb**

This verb 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.

**Diversified Key Generate verb**

This verb generates 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.

**Key Export verb**

This verb 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.

**Key Generate verb**

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.

When you call the Key Generate verb, include parameters specifying information about the key you want generated. Because the form of the key restricts its use, you need to choose the form you want the generated key to have. You can use the `key_form` parameter to specify the form. The possible forms are:

**Operational**

The key is used for cryptographic operations on the local system. 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**

The key is stored with a file or sent to another system. Importable keys are protected by importer key-encrypting keys.

**Exportable**

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 23.

**Key Import verb**

This verb 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.
Key Part Import verb

This verb combines clear key parts of any key type and returns the combined key value in an internal token.

Key Test verb

This verb generates or verifies a secure cryptographic verification pattern for keys. A parameter indicates the action you want to perform. The key to test can be in the clear or encrypted under a master key.

Key Test Extended verb

This verb is essentially the same as "Key Test (CSNBKYT)" on page 146, 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.

Key Token Build verb

This verb is a utility function you can use to build CCA internal or external key tokens for all key types 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 Generate and Key Part Import verbs.

Key Token Change verb

This verb is used to re-encipher 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 Parse verb

This verb is used to disassemble a key toke into separate pieces of information. The verb can disassemble an external key-token or an internal key-token in application storage.

Key Translate verb

This 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.

Multiple Clear Key Import verb

This verb 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.

Prohibit Export verb

This verb modifies the control vector of a CCA key token so that the key cannot be exported. The verb operates only on internal key tokens.

Prohibit Export Extended verb

This verb modifies an external DES key-token so that the key can no longer be exported after it has been imported. The verb operates only on internal key tokens.
Random Number Generate verb
This verb uses hardware to generate a cryptographic-quality random number suitable for use as an encryption key or for other purposes.

Random Number Generate Long verb
This verb generates a cryptographic-quality random number suitable for use as an encryption key or for other purposes, ranging from 1 to 8192 bytes long.

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

AES Key Record Create verb
This verb creates 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.

AES Key Record Delete verb
This verb performs one of the following tasks 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.

AES Key Record List verb
This 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.

AES Key Record Read verb
This verb reads a key-token record from AES key-storage and returns 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.

AES Key Record Write verb
This verb writes a copy of an AES key-token from application storage into AES key-storage. The 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.

DES Key Record Create verb
This verb accepts a key label and creates a null key record in the key storage file. The record contains a key token set to binary zeros and is identified by the key label passed in the function call statement. The key label must be unique.
DES Key Record Delete verb
This verb accepts a key label and deletes the associated key record from key storage file. This verb deletes the entire record, including the key label.

DES Key Record List verb
This verb creates a list of specified information about key records in the DES key storage file.

DES Key Record Read verb
This verb copies a key token from the key storage file to the application storage, where it can be used directly in other cryptographic services.

DES Key Record Write verb
This verb accepts an internal key token and a label and writes the key token to the key storage file record identified by the key label. The key label must be unique and the record must already exist in the file.

Retained Key Delete verb
This verb deletes a PKA key-record currently retained within the cryptographic engine.

Retained Key List verb
This verb lists the key labels of selected PKA key records that have been retained within the cryptographic engine.

Verbs for managing PKA key storage files
The PKA key storage file is a repository for RSA keys. 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.

PKA Key Record Create verb
This verb accepts an RSA private key token in either external or internal format, or an RSA public key token and writes a new record to the PKA key storage file. An application can create a null token in the file by specifying a token length of zero. The key label must be unique.

PKA Key Record Delete verb
This verb deletes a record from the PKA key storage file. An application can specify that the entire record be deleted, or that only the contents of the record be deleted. If only the contents of the record are deleted, the record will still exist in the file but will contain only binary zeros. The key label must be unique.

PKA Key Record List verb
This verb creates a list containing information about specified key records in the PKA Key Storage file.

PKA Key Record Read verb
This verb reads a record from the PKA key storage file and returns the contents of that record to the caller. The key label must be unique.
PKA Key Record Write verb

This verb accepts an RSA private key token in either external or internal format, or an RSA public key token and writes over an existing record in the PKA key storage file. An application can check the file for a null record with the label provided and overwrite this record if it does exist. Alternatively, an application can specify to overwrite a record regardless of the contents of the record.

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.

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.

PKA Decrypt verb

The PKA Decrypt verb uses the RSA private key to unwrap the RSA-encrypted key and parse the key value. This verb then returns the clear key value to the application.

PKA Encrypt verb

The PKA Encrypt verb encrypts a supplied clear key value under an RSA public key. Currently, the supplied key can be formatted using the PKCS 1.2 or ZERO-PAD methods prior to encryption.

Enciphering and deciphering data

The Encipher and Decipher verbs protect data 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.

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

The Encipher and Decipher verbs can work only with a key type of DATA, if they are to make use of CPACF.

Decipher verb

This verb is used to decipher data using the DES cipher block chaining mode.

Encipher verb

This verb is used to encipher data using the DES cipher block chaining mode.

Symmetric Algorithm Decipher verb

Use the Symmetric Algorithm Decipher verb to decipher data using the AES cipher block chaining mode.
CCA supports the following processing rules to decipher data: CBC, ECB, PKCS-PAD. You choose the type of processing rule that the verb should use for block chaining. The AES key used to decipher the data can either be 16, 24, or 32 bytes (128, 192, or 256 bits) in length.

**Symmetric Algorithm Encipher verb**

Use the Symmetric Algorithm Encipher verb to encipher data using the AES cipher block chaining mode.

CCA supports the following processing rules to encipher data: CBC, ECB, PKCS-PAD. You choose the type of processing rule that the verb should use for block chaining. The AES key used to encipher the data can either be 16, 24, or 32 bytes (128, 192, or 256 bits) in length.

---

**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
- 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 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.

MAC Generate verb

When a message is sent, an application program can generate an authentication code for it using the MAC Generate verb. The 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.

CCA allows a MAC to 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.

MAC Verify verb

When the receiver gets the message, an application program calls the MAC Verify verb. The 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 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.

Hashing functions

Hashing functions include one-way hash generation and modification detection code (MDC) processing.

One-Way Hash verb

This verb hashes a supplied message. Supported hashing methods include:

- SHA-1
- MD5
- RIPEMD-160

---

1. The Secure Hash Algorithm (SHA) is also called the Secure Hash Standard (SHS), which Federal Information Processing Standard (FIPS) Publication 180 defines.
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- 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 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.

CVV Generate verb

This verb is used to generate a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2.

2. The VISA CVV and the MasterCard CVC refer to the same value. CVV is used here to mean both CVV and CVC.
CVV Verify verb

This verb is used to verify a VISA Card Verification Value (CVV) or MasterCard Card Verification Code (CVC) as defined for track 2.

Clear PIN Encrypt verb

To format a PIN into a PIN block format and encrypt the results, use the Clear PIN Encrypt verb. 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.

Clear PIN Generate verb

To generate personal identification numbers, call the Clear PIN Generate verb. Using a PIN generation algorithm, data used in the algorithm, and the PIN generation key, the verb generates a clear PIN, a PIN verification value, or an offset.

Clear PIN Generate Alternate verb

To generate a clear VISA PIN validation value from an encrypted PIN block, call the Clear PIN Generate Alternate verb. This verb also supports the IBM-PINO algorithm to produce a 3624 offset from a customer selected encrypted PIN.

Note: The PIN block must be encrypted under either an input PIN-encrypting key (IPINENC) or output PIN-encrypting key (OPINENC).

Encrypted PIN Generate verb

To generate personal identification numbers, call the Encrypted PIN Generate verb. Using a PIN generation algorithm, data used in the algorithm, and the PIN generation key, the verb generates a PIN and using a PIN block format and the PIN encrypting key, formats and encrypts the PIN.

Encrypted PIN Translate verb

To translate a PIN from one PIN-encrypting key to another or from one PIN block format to another or both, call the Encrypted PIN Translate verb. 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.

Encrypted PIN Verify verb

To verify a supplied PIN, call the Encrypted PIN Verify verb. You need to specify the supplied enciphered PIN, the PIN-encrypting key that enciphers it, and other relevant data. You must also specify the PIN verification key and PIN verification algorithm. It compares the two personal identification numbers; if they are the same, it verifies the supplied PIN. See Chapter 9, “Financial services,” on page 281 for additional information.

PIN Change/Unblock verb

To support PIN change algorithms specified in the VISA Integrated Circuit Card Specification, call the PIN Change/Unblock verb.
Transaction Validation verb

To support generation and validation of American Express card security codes, call the Transaction Validation verb.

Secure messaging

The following verbs will assist applications in encrypting secret information such as clear keys and PIN blocks in a secure message. These verbs will execute within the secure boundary of the cryptographic coprocessor.

Secure Messaging for Keys verb

This verb encrypts a text block, including a clear key value decrypted from an internal or external DES token.

Secure Messaging for PINs verb

This verb encrypts a text block, including a clear PIN block recovered from an encrypted PIN block.

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 CEX3C 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 also can set AES master keys on the CEX3C 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 F, “Access control points and verbs,” on page 519 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 are shown in Table 3.

Table 3. Combinations of the verbs

<table>
<thead>
<tr>
<th>Combination A (DATA keys only)</th>
<th>Combination B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Random number generate</td>
<td>1. Random number generate</td>
</tr>
<tr>
<td>2. Clear key import or multiple clear key import</td>
<td>2. Any Service</td>
</tr>
<tr>
<td>3. Data key export or key export (optional step)</td>
<td>3. Data key export for DATA keys, or key export in the general case (optional step)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combination C</th>
<th>Combination D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Key generate (OP form only)</td>
<td>1. Key generate (OPEX form)</td>
</tr>
<tr>
<td>2. Any service</td>
<td>2. Any service</td>
</tr>
<tr>
<td>3. Key export (optional)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combination E</th>
<th>Combination F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Key generate (IM form only)</td>
<td>1. Key generate (IMEX form)</td>
</tr>
<tr>
<td>2. Key import</td>
<td>2. Key import</td>
</tr>
<tr>
<td>3. Any service</td>
<td>3. Any service</td>
</tr>
<tr>
<td>4. Key export (optional)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combination G</th>
<th>Combination H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Key generate</td>
<td>1. Key import</td>
</tr>
<tr>
<td>2. Key record create</td>
<td>2. Key record create</td>
</tr>
<tr>
<td>3. Key record write</td>
<td>3. Key record write</td>
</tr>
<tr>
<td>4. Any service (passing label of the key just generated)</td>
<td>4. Any service (passing label of the key just generated)</td>
</tr>
</tbody>
</table>

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 “Key Generate (CSNBKGN)” on page 130.

Key forms and types used in the Key Generate verb

The Key Generate verb is the most complex of all the CCA verbs. This section provides examples of the key forms and key types used in the Key Generate verb.

Generating an operational key

To generate an operational key, choose one of the following methods:

- **For operational keys**, call the Key Generate verb (CSNBKGN). Table 22 on page 136 and Table 23 on page 136 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 verb (CSNBRNG) and specify the form parameter as ODD. Then pass the generated value to the Clear Key Import verb (CSNBCKI) or the Multiple Clear Key Import verb (CSNBCKM). 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**

<table>
<thead>
<tr>
<th>Key</th>
<th>Key Form</th>
<th>Key Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>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>
</tr>
<tr>
<td>OP</td>
<td>MAC</td>
<td>MAC Generate. Because no MACVER key exists, there is no secure communication of the MAC with another cryptographic partner.</td>
</tr>
<tr>
<td>IM</td>
<td>DATA</td>
<td>Key Import, and then Encipher or Decipher. Then Key Export to communicate ciphertext and key with another cryptographic partner.</td>
</tr>
<tr>
<td>EX</td>
<td>DATA</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Examples of OPIM 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</th>
<th>Key Form</th>
<th>Key Type</th>
<th>Type Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPIM</td>
<td>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>
<td></td>
</tr>
<tr>
<td>OPIM</td>
<td>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</td>
<td></td>
</tr>
</tbody>
</table>
Verify verb, which compares the MAC it generates with the MAC supplied with the message and issues a return code indicating whether they compare.

Examples of OPEX single-length, double-length, and triple-length keys in two forms

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Type Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form 1</td>
<td>2</td>
</tr>
</tbody>
</table>

- **OPEX DATA DATA**
  - Use the OP form in Encipher. Send the EX form and the ciphertext to another cryptographic partner.

- **OPEX MAC MAC**
  - 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.

- **OPEX MAC MACVER**
  - 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.

- **OPEX PINGEN PINVER**
  - 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.

- **OPEX IMPORTER EXPORTER**
  - 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.

- **OPEX EXPORTER 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 installed, you can start key and data exchange.

Examples of IMEX single-length and double-length keys in two forms

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Type Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form 1</td>
<td>2</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 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 verbs 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

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>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEX</td>
<td>DATA</td>
<td>DATA</td>
</tr>
<tr>
<td>EXEX</td>
<td>MAC</td>
<td>MACVER</td>
</tr>
<tr>
<td>EXEX</td>
<td>IMPORTER</td>
<td>EXPORTER</td>
</tr>
<tr>
<td>EXEC</td>
<td>OPINENC</td>
<td>IPINENC</td>
</tr>
</tbody>
</table>

Send the first EX form to a cryptographic partner with the corresponding IMPORTER and send the second EX form to another cryptographic partner with the corresponding IMPORTER. This exchange establishes a key between two partners.

Summary of the CCA nodes and access control verbs

Table 4 lists the CCA nodes and access control verbs described in this document. The table also references the chapter that describes the verb.

Table 4. Summary of CCA nodes and access control verbs

<table>
<thead>
<tr>
<th>Verb</th>
<th>Name</th>
<th>Function</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>
</tr>
<tr>
<td>CSUACFV</td>
<td>Cryptographic Facility Version</td>
<td>Retrieve the Security Application Program Interface (SAPI) Version and build date.</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>
</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>
</tr>
<tr>
<td>CSNBKSI</td>
<td>Key Storage Initialization</td>
<td>Initializes a key-storage file using the current symmetric or asymmetric master-key.</td>
</tr>
<tr>
<td>CSNBMKP</td>
<td>Master Key Process</td>
<td>Operates on the three master-key registers: new, current, and old.</td>
</tr>
<tr>
<td>CSUARNT</td>
<td>Random Number Tests</td>
<td>Invokes the USA NIST FIPS PUB 140-1 specified cryptographic operational tests.</td>
</tr>
</tbody>
</table>

Summary of the AES, DES, and hashing verbs

Table 5 lists the AES, DES, and hashing verbs described in this document. The table also references the chapter that describes the verb.

Table 5. Summary of CCA AES, DES, and hashing verbs

<table>
<thead>
<tr>
<th>Verb</th>
<th>Verb Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 5, “Managing DES and AES cryptographic keys.” on page 105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verb</td>
<td>Verb Name</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<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. CSNBCKI converts the clear key into operational form as a DATA key.</td>
</tr>
<tr>
<td>CNBCVG</td>
<td>Control Vector Generate</td>
<td>Builds a control vector from keywords specified by the key_type and rule_array parameters.</td>
</tr>
<tr>
<td>CNBCVT</td>
<td>Control Vector Translate</td>
<td>Changes the control vector used to encipher an external DES key.</td>
</tr>
<tr>
<td>CNBCVE</td>
<td>Cryptographic Variable Encipher</td>
<td>Encrypts plaintext using a CVARENC key to produce ciphertext using the Cipher Block Chaining (CBC) method.</td>
</tr>
<tr>
<td>CSNBDKKX</td>
<td>Data Key Export</td>
<td>Converts a DATA key from operational form into exportable form.</td>
</tr>
<tr>
<td>CSNBDKM</td>
<td>Data Key Import</td>
<td>Imports an encrypted source DES single- or double-length DATA key and creates or updates a target internal key token with the master key enciphered source key.</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.</td>
</tr>
<tr>
<td>CSNBEKX</td>
<td>Key Export</td>
<td>Converts any key from operational form into exportable form. (However, this verb does not export a key that was marked non-exportable when it was imported.)</td>
</tr>
<tr>
<td>CSNKGN</td>
<td>Key Generate</td>
<td>Generates a 64-bit, 128-bit, or 192-bit odd parity key, or a pair of keys; and returns them in encrypted forms (operational, exportable, or importable). CSNKGN does not produce keys in plaintext.</td>
</tr>
<tr>
<td>CSNBKIM</td>
<td>Key Import</td>
<td>Converts any key from importable form into operational form.</td>
</tr>
<tr>
<td>CSNBKP</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>
</tr>
<tr>
<td>CSNBKTYT</td>
<td>Key Test</td>
<td>Generates or verifies (depending on keywords in the rule_array) a secure verification pattern for keys. It requires the tested key to be in the clear or encrypted under the master key.</td>
</tr>
<tr>
<td>CSNBKYTX</td>
<td>Key Test Extended</td>
<td>Verifies the value of a key or key part in an external or internal key token.</td>
</tr>
<tr>
<td>CSNBKTB</td>
<td>Key Token Build</td>
<td>Builds an internal or external token from the supplied parameters. You can use this verb to build CCA key tokens for all key types CCA supports.</td>
</tr>
<tr>
<td>CSNBKTC</td>
<td>Key Token Change</td>
<td>Re-enciphers a DES key from encryption under the old master key to encryption under the current master key.</td>
</tr>
<tr>
<td>Verb</td>
<td>Verb Name</td>
<td>Function</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CSNBKTP</td>
<td>Key Token Parse</td>
<td>Disassembles a key token into separate pieces of information.</td>
</tr>
<tr>
<td>CSNBKTR</td>
<td>Key Translate</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.</td>
</tr>
<tr>
<td>CSNBCKM</td>
<td>Multiple Clear Key Import</td>
<td>Imports a single, double, or triple-length clear DATA key, enciphers it under the master key, and places the result into an internal key token. CSNBCKM converts the clear key into operational form as a DATA key.</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>
</tr>
<tr>
<td>CSNDPKE</td>
<td>PKA Encrypt</td>
<td>Encrypts a supplied clear key value under an RSA public key.</td>
</tr>
<tr>
<td>CSNBPEX</td>
<td>Prohibit Export</td>
<td>Modifies an operational key so that it cannot be exported.</td>
</tr>
<tr>
<td>CSNBPEXX</td>
<td>Prohibit Export Extended</td>
<td>Modifies an operational key so that it cannot be exported after it has been imported.</td>
</tr>
<tr>
<td>CSNBRNG</td>
<td>Random Number Generate</td>
<td>Generates an 8-byte random number. The output can be specified in three forms of parity: RANDOM, ODD, and EVEN.</td>
</tr>
<tr>
<td>CSNBRNGL</td>
<td>Random Number Generate Long</td>
<td>Generates a random number from 1 to 8192 bytes long. The output can be specified in three forms of parity: RANDOM, ODD, and EVEN.</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>
</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>
</tr>
<tr>
<td>CSNBSAD</td>
<td>Symmetric Algorithm Decipher</td>
<td>Deciphers data using the AES cipher block chaining mode.</td>
</tr>
<tr>
<td>CSNBSAE</td>
<td>Symmetric Algorithm Encipher</td>
<td>Enciphers data using the AES cipher block chaining mode</td>
</tr>
<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>
</tr>
</tbody>
</table>

Chapter 6, “Protecting data,” on page 191

Chapter 7, “Verifying data integrity and authenticating messages,” on page 217
<table>
<thead>
<tr>
<th>Verb</th>
<th>Verb Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<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>
</tr>
<tr>
<td>CSNBOWH</td>
<td>One-Way Hash</td>
<td>Generates a one-way hash on specified text.</td>
</tr>
<tr>
<td>CSNBCPE</td>
<td>Clear PIN Encrypt</td>
<td>Formats a PIN into a PIN block format and encrypts the results.</td>
</tr>
<tr>
<td>CSNBPGN</td>
<td>Clear PIN Generate</td>
<td>Generates a clear personal identification number (PIN), a PIN verification value (PVV), or an offset using one of the following algorithms:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IBM 3624 (IBM-PIN or IBM-PINO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IBM German Bank Pool (GBP-PIN or GBP-PINO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VISA PIN validation value (VISA-PVV)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interbank PIN (INBK-PIN)</td>
</tr>
<tr>
<td>CSNBCPA</td>
<td>Clear PIN Generate Alternate</td>
<td>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.</td>
</tr>
<tr>
<td>CSNBCSG</td>
<td>CVV Generate</td>
<td>Generates a VISA Card Verification Value (CVV) or a MasterCard Card Verification Code (CVC).</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).</td>
</tr>
<tr>
<td>CSNBEPG</td>
<td>Encrypted PIN Generate</td>
<td>Generates and formats a PIN and encrypts the PIN block.</td>
</tr>
<tr>
<td>CSNBPTR</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.</td>
</tr>
<tr>
<td>CSNPBVR</td>
<td>Encrypted PIN Verify</td>
<td>Verifies a supplied PIN using one of the following algorithms:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IBM 3624 (IBM-PIN or IBM-PINO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IBM German Bank Pool (GBP-PIN or GBP-PINO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VISA PIN validation value (VISA-PVV)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interbank PIN (INBK-PIN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UKPT keywords are supported.</td>
</tr>
<tr>
<td>Verb</td>
<td>Verb Name</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</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>
</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>
</tr>
<tr>
<td>CSNBSPN</td>
<td>Secure Messaging for PINs</td>
<td>Encrypts a text block, including a clear PIN block recovered from an encrypted PIN block.</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>
</tr>
</tbody>
</table>

### Summary of key storage verbs

Table 6 lists the key storage access verbs described in this document. The figure also references the chapter that describes the verb.

<table>
<thead>
<tr>
<th>Verb</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNBAKRC</td>
<td>AES Key Record Create</td>
<td>Adds a key record containing a key token set to binary zeros to AES key storage file.</td>
</tr>
<tr>
<td>CSNBAKRD</td>
<td>AES Key Record Delete</td>
<td>Deletes a key record from the AES key storage file.</td>
</tr>
<tr>
<td>CSNBAKRR</td>
<td>AES Key Record Read</td>
<td>Copies an internal key token from the AES key storage file to application storage.</td>
</tr>
<tr>
<td>CSNBAKRW</td>
<td>AES Key Record Write</td>
<td>Writes an internal key token to the AES key storage file. storage file record specified in the key label parameter.</td>
</tr>
<tr>
<td>CSNBAKRL</td>
<td>AES Key Record List</td>
<td>Creates a key-record-list file containing information about specified key records in key-storage.</td>
</tr>
<tr>
<td>CSNBKRC</td>
<td>DES Key Record Create</td>
<td>Adds a key record containing a key token set to binary zeros to DES key storage file.</td>
</tr>
<tr>
<td>CSNBKRD</td>
<td>DES Key Record Delete</td>
<td>Deletes a key record from the DES key storage file.</td>
</tr>
<tr>
<td>CSNBKRR</td>
<td>DES Key Record Read</td>
<td>Copies an internal key token from the DES key storage file to application storage.</td>
</tr>
<tr>
<td>CSNBKRW</td>
<td>DES Key Record Write</td>
<td>Writes an internal key token to the DES key storage file. storage file record specified in the key label parameter.</td>
</tr>
<tr>
<td>CSNBKRL</td>
<td>DES Key Record List</td>
<td>Creates a key-record-list file containing information about specified key records in key-storage.</td>
</tr>
</tbody>
</table>
### Table 6. Summary of key storage verbs (continued)

<table>
<thead>
<tr>
<th>Verb</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNDKRC</td>
<td>PKA Key Record Create</td>
<td>Writes a new record to the PKA key storage file.</td>
</tr>
<tr>
<td>CSNDKRD</td>
<td>PKA Key Record Delete</td>
<td>Delete a record from the PKA key storage file.</td>
</tr>
<tr>
<td>CSNDKRR</td>
<td>PKA Key Record Read</td>
<td>Read a record from the PKA key storage file and return the contents of that record.</td>
</tr>
<tr>
<td>CSNDKRW</td>
<td>PKA Key Record Write</td>
<td>Write over an existing record in the PKA key storage file.</td>
</tr>
<tr>
<td>CSNDKRL</td>
<td>PKA Key Record List</td>
<td>Generate a list of information about specified records in the PKA key storage file.</td>
</tr>
<tr>
<td>CSNDRKL</td>
<td>Retained Key Delete</td>
<td>Deletes a PKA key-record currently retained within the cryptographic engine.</td>
</tr>
<tr>
<td>CSNDRKL</td>
<td>Retained Key List</td>
<td>Obtains a list of the key labels of selected PKA key records that have been retained within the cryptographic engine.</td>
</tr>
</tbody>
</table>
Chapter 3. Introducing PKA cryptography and using PKA verbs

The preceding sections 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).

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

The RSA algorithm

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:

ciphertext = cleartext\(^{\text{PU}}\) (modulo M)

Similarly, the following operation recovers cleartext from ciphertext:

cleartext = ciphertext\(^{\text{PR}}\) (modulo M)

An RSA key consists of an exponent and a modulus. The private exponent must be secret, but the public exponent and modulus need not be secret.

PKA master keys

On the cryptographic processor, RSA private, keys are protected by the Asymmetric-Keys Master Key (ASYM-MK). The ASYM-MK is a triple-length key used to encipher and decipher PKA keys.

In order for PKA verbs to function on the processor, the Asymmetric-Keys Master Key must be installed. The administrator installs the ASYM-MK on the CEX3C by using the clear master key entry panels, the z/OS clear key entry panels, or the 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 PKA master key. A new, random OPK is generated for each private key when it is imported or generated. CCA provides a public key storage file for the storage of application PKA keys. The PKA Key Token Change verb can be executed to change a private PKA token from encryption under the old ASYM-MK to encryption under the current ASYM-MK.
PKA verbs

The CEX3C running on the IBM System z10 model GA3 provides RSA digital signature functions, key management and key generation functions, DES key distribution functions, and data encryption functions, and application programming interfaces to these functions through verbs.

Verbs supporting digital signatures

CCA provides the following verbs that support digital signatures.

Digital Signature Generate verb

This verb generates a digital signature. It supports the following methods:

- ANSI X9.31
- ISO 9796-1
- RSA DSI PKCS 1.0 and 1.1
- Padding on the left with zeros

The input text must have been previously hashed using the One-Way Hash verb or the MDC Generate verb.

Digital Signature Verify verb

This verb verifies a digital signature using a PKA public key. It supports the following methods:

- ANSI X9.31
- ISO 9796-1
- RSA DSI PKCS 1.0 and 1.1
- Padding on the left with zeros

The text that is input to this verb must be previously hashed using the One-Way Hash verb or the MDC Generate verb.

MDC Generate verb

The user must enable the Generate MDC command (offset X'008A') with a Trusted Key Entry (TKE) workstation before using this verb.

This verb creates 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.

Verbs for PKA key management

CCA provides the following verbs for PKA key management.

PKA Key Generate verb

This verb generates an RSA key pair. The private key can be in cleartext, in an internal token, or in an external token.
Input to the PKA Key Generate verb is either a skeleton key token created by the PKA Key Token Build verb or a valid key token.

**PKA Key Import verb**

This verb imports an RSA private key into an internal key token.

The key token to import can be in the clear or encrypted. The PKA key token build utility creates a clear PKA key token. The PKA Key Generate verb generates either a clear or an encrypted PKA key token.

**PKA Key Token Build verb**

This verb is a utility you can use to create an external PKA key token containing an unenciphered private RSA key. You can supply this token as input to the PKA Key Import verb to obtain an operational internal token containing an enciphered private key. You can also use this verb to input a clear unenciphered public RSA key and return the public key in a token format that other PKA verbs can use directly.

Use this verb to build skeleton key tokens for input to the PKA Key Generate verb for generation of RSA keys.

**PKA Key Token Change verb**

This verb is a utility you can use to change 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.

**PKA Key Translate verb**

This verb is a utility you can use to translate 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.

**PKA Public Key Extract verb**

This verb extracts a PKA public key token from a PKA internal (operational) or external (importable) private key token. It performs no cryptographic verification of the PKA private key token.

**PKA Symmetric Key Export verb**

This verb transfers an application-supplied symmetric key (a DATA key) from encryption under the DES host master key to encryption under an application-supplied RSA public key. The application-supplied DATA key must be a DES internal key token or the label of such a token in the CCA key storage file. The PKA Symmetric Key Import verb can import the PKA-encrypted form at the receiving node.

**PKA Symmetric Key Generate verb**

This verb generates a symmetric key (that is, a DATA key) and returns it encrypted using DES and encrypted under an RSA public key token.

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 PKA Symmetric Key Import verb to import the PKA-encrypted form.)
PKA Symmetric Key Import verb

This verb imports a symmetric (DES) DATA key enciphered under an RSA public key. The verb returns the key in operational form enciphered under the DES master key.

Remote Key Export verb

The 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).

Trusted Block Create verb

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 444.

PKA key tokens

PKA key tokens contain RSA private or public keys. Although DES tokens are 64 bytes, PKA tokens are variable length because they contain RSA key values, which are variable in length. Consequently, length parameters precede all PKA token parameters. The maximum allowed size is 2500 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:

- 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 private key info</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>RSA public key info</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Key name (optional)</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Internal info</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 key tokens begin in "Format of the RSA public key token" on page 433.

**PKA key management**

You can generate RSA keys using the CCA PKA Key Generate verb.
- Using the Transaction Security System PKA Key Generate verb, or a comparable product from another vendor.

![Diagram of PKA key management](image)

*Figure 4. PKA key management*

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 370).
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 RSA 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-justified, 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 2500 bytes) field composed of key value and control information. PKA keys can be either public or private RSA 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'). Table 8 is a list of private key section identifiers for internal and external private RSA key tokens:

Table 8. Internal and external private RSA key token section identifiers

<table>
<thead>
<tr>
<th>Key token</th>
<th>Section identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA Private Key Token 1024 Modulus-Exponent External Form</td>
<td>X'02'</td>
</tr>
<tr>
<td>RSA Private Key Token 2048 Chinese Remainder Theorem External Form</td>
<td>X'08'</td>
</tr>
<tr>
<td>RSA Private Key Token 1024 Modulus-Exponent Internal Form (Cryptographic Coprocessor Feature)</td>
<td>X'02'</td>
</tr>
<tr>
<td>RSA Private Key Token 1024 Modulus-Exponent Internal Form (CEX3C)</td>
<td>X'06'</td>
</tr>
<tr>
<td>RSA Private Key Token 2048 Chinese Remainder Theorem Internal Form (CEX3C)</td>
<td>X'08'</td>
</tr>
</tbody>
</table>

See Appendix B, “Key token formats,” on page 429 for descriptions of the PKA key tokens.

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 11, “Managing PKA cryptographic keys.”

- 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.

For debugging information, see Appendix B, “Key token formats” for the format of an internal 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 11, “Managing PKA cryptographic keys.”

- PKA Public Key Extract
- PKA Key Token Build
- PKA Key Generate
For debugging information, see Appendix B, “Key token formats” 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.

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

### Summary of the PKA verbs

Table 9 lists the PKA verbs, described in this book, and their corresponding verb names. (The PKA verb names start with CSND and have corresponding CSF names.) This table also references the chapter that describes the verb.

**Table 9. Summary of PKA verbs**

<table>
<thead>
<tr>
<th>Verb</th>
<th>Verb Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNDDSG</td>
<td>Digital Signature Generate</td>
<td>Generates a digital signature using an RSA private key.</td>
</tr>
<tr>
<td>CSNDDSV</td>
<td>Digital Signature Verify</td>
<td>Verifies a digital signature using an RSA public key.</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>
</tr>
<tr>
<td>CSNDPKG</td>
<td>PKA Key Generate</td>
<td>Generates an RSA key pair.</td>
</tr>
<tr>
<td>CSNDPKI</td>
<td>PKA Key Import</td>
<td>Imports a PKA key token containing either a clear key or an RSA key enciphered under a transport key.</td>
</tr>
<tr>
<td>CSNDPKB</td>
<td>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 CSNDPKB on a clear public RSA key, returns the public key in a token format that other PKA verbs can directly use. CSNDPKB 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>
</tr>
<tr>
<td>CSNDKTC</td>
<td>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>
</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>
</tr>
</tbody>
</table>
Table 9. Summary of PKA verbs (continued)

<table>
<thead>
<tr>
<th>Verb</th>
<th>Verb Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<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>
</tr>
<tr>
<td>CSNDSYX</td>
<td>PKA Symmetric Key Export</td>
<td>Transfer an application-supplied symmetric key (a DATA key) from encryption under the DES master key to encryption under an application-supplied RSA public key.</td>
</tr>
<tr>
<td>CSNDSYG</td>
<td>PKA 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.</td>
</tr>
<tr>
<td>CSNDSYI</td>
<td>PKA Symmetric Key Import</td>
<td>Import a symmetric DATA key enciphered under an RSA public key into operational form enciphered under a DES master key.</td>
</tr>
<tr>
<td>CSNDRKX</td>
<td>Remote Key Export</td>
<td>Secure 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).</td>
</tr>
<tr>
<td>CSNDTBC</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>
</tr>
</tbody>
</table>
This part of the document introduces AES, DES and PKA verbs.
Chapter 4. Using the CCA nodes and access control verbs

This chapter describes the following verbs:

- "Cryptographic Facility Query (CSUACFQ)"
- "Cryptographic Facility Version (CSUACFV)" on page 87
- "Cryptographic Resource Allocate (CSUACRA)" on page 89
- "Cryptographic Resource Deallocate (CSUACRD)" on page 92
- "Key Storage Initialization (CSNBKSI)" on page 95
- "Master Key Process (CSNBMKP)" on page 98
- "Random Number tests (CSUARNT)" on page 103

CCA nodes and access control verbs

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

Determining if a card is a CEX2C or CEX3C

Using Cryptographic Facility Query, the output rule_array for option STATCCA is the most accurate way to determine if you are using a CEX2C or CEX3C:

- If first two characters of the CCA application version field are ‘z’ followed by ‘3’, then this card is a CEX2C adapter without any doubt.

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.

- 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.

- 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 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 whether your cards are CEX3C or CEX2C, by calling the Cryptographic Facility Query verb for all available adapters.

For details about panel.exe, see "The panel.exe utility" on page 551.

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

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

**Format**

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

**Parameters**

**return_code**

Direction: Output  
Type: Integer

The return_code parameter is a pointer to an integer value that expresses the general results of processing. See "Return code and reason code overview" on page 16 for more information about return codes.

**reason_code**

Direction: Output  
Type: Integer

The reason_code parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See "Return code and reason code overview" on page 16 for more information about reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: Integer

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 0.

**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, the value must be 1 or 2 for this verb.
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

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.

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

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAPTER1</td>
<td>This keyword is ignored. It is accepted for backward compatibility.</td>
</tr>
<tr>
<td>STATCCA</td>
<td>Obtains CCA-related status information.</td>
</tr>
<tr>
<td>STATCCAE</td>
<td>Obtains CCA-related extended status information.</td>
</tr>
<tr>
<td>STATCARD</td>
<td>Obtains coprocessor-related basic status information.</td>
</tr>
<tr>
<td>STATDIAG</td>
<td>Obtains diagnostic information.</td>
</tr>
<tr>
<td>STATEID</td>
<td>Obtains the Environment Identifier (EID).</td>
</tr>
<tr>
<td>STATEXPT</td>
<td>Obtains function control vector-related status information.</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>
</tr>
<tr>
<td>STATAES</td>
<td>Obtains status information on AES master-key registers and AES key-length enablement.</td>
</tr>
<tr>
<td>STATMOFN</td>
<td>Obtains master-key shares distribution information.</td>
</tr>
<tr>
<td>QPENDING</td>
<td>TKE uses this <code>rule_array</code> 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.</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.</td>
</tr>
</tbody>
</table>

This keyword applies only when using Linux on IBM System z.
Table 10. Keywords for Cryptographic Facility Query control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET-UDX</strong></td>
<td>Obtains UDX identifiers. See &quot;GET-UDX&quot; on page 78. This keyword applies only when using Linux on IBM System z.</td>
</tr>
<tr>
<td><strong>STATKPRL</strong></td>
<td>Obtains the names of the operational key parts. See &quot;STATKPRL&quot; on page 78. This keyword applies only when using Linux on IBM System z.</td>
</tr>
<tr>
<td><strong>STATKPR</strong></td>
<td>Obtains non-secret information about an operational key part. See &quot;STATKPR&quot; on page 78. This keyword applies only when using Linux on IBM System z.</td>
</tr>
<tr>
<td><strong>TKESTATE</strong></td>
<td>Indicates whether TKE access is enabled or not. This keyword applies only when using Linux on IBM System z.</td>
</tr>
<tr>
<td><strong>STATICSX</strong></td>
<td>Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. See &quot;STATICSX&quot; on page 83. This keyword applies only when using Linux on IBM System z.</td>
</tr>
<tr>
<td><strong>STATICSA</strong></td>
<td>Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. See &quot;STATICSA&quot; on page 79. This keyword applies only when using Linux on IBM System z.</td>
</tr>
<tr>
<td><strong>STATICSE</strong></td>
<td>Obtains the indicated master key hash and verification patterns to be returned for the master keys loaded in the current domain. See &quot;STATICSE&quot; on page 81. This keyword applies only when using Linux on IBM System z.</td>
</tr>
</tbody>
</table>

The format of the output `rule_array` depends on the value of the `rule_array` element, which identifies the information to be returned. Different sets of `rule_array` elements are returned depending on whether the input keyword is **STATCCA**, **STATCCAE**, **STATCARD**, **STATDIAG**, **STATEID**, **STATEXPT**, **STATMOFN**, or **TIMEDATE**.

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).

On output, the `rule_array` elements can have the values shown in Table 11.

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><strong>Output <code>rule_array</code> for option STATCCA</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NMK status</td>
<td>The state of the new master-key register:</td>
</tr>
<tr>
<td></td>
<td><strong>Value</strong></td>
<td><strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>The register is clear.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>The register contains a partially complete key.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>The register contains a key.</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>2</td>
<td>CMK status</td>
<td>The state of the current master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2  The register contains a key.</td>
</tr>
<tr>
<td>3</td>
<td>OMK status</td>
<td>The state of the old master-key register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2  The register contains a key.</td>
</tr>
<tr>
<td>4</td>
<td>CCA application version</td>
<td>A character string that identifies the version of the CCA application program running in the coprocessor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The results of this query come directly from the card itself. If the host device driver is not up to date, it could incorrect identify a CEX3C as a CEX2C. Therefore, looking at this field resolves all questions.</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 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>
</tbody>
</table>

**Output rule_array for option STATCCAE**

<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> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2  The register contains a partially complete key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3  The register contains a key.</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> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2  The register contains a key.</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> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1  The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2  The register contains a key.</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>
</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>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> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 The register contains a partially complete key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 The register contains a key.</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> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 The register contains a key.</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> <strong>Meaning</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 The register is clear.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 The register contains a key.</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>
<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>
</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>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 <strong>binary</strong> 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 STATDIAG**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</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>Meaning</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>

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The error log is empty.</td>
</tr>
<tr>
<td>2</td>
<td>The error log contains abnormal termination data, but is not yet full.</td>
</tr>
<tr>
<td>3</td>
<td>The error log is full and cannot hold any more data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No intrusion has been detected.</td>
</tr>
<tr>
<td>2</td>
<td>An intrusion attempt has been detected.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Only acceptable voltages have been detected.</td>
</tr>
<tr>
<td>2</td>
<td>A voltage has been detected below the low-voltage tamper threshold.</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>
</table>
| 6              | 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:  
*Value Meaning*  
1 Only acceptable voltages have been detected.  
2 A voltage has been detected greater than the high-voltage tamper threshold. |
| 7              | 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:  
*Value Meaning*  
1 The temperature is acceptable.  
2 The temperature has been detected outside of an acceptable limit. |
| 8              | 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:  
*Value Meaning*  
1 No radiation has been detected.  
2 Radiation has been detected. |
| 9, 11, 13, 15, 17 | 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. |
| 10, 12, 14, 16, 18 | 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. |

**Output rule_array for option STATEID**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>EID</td>
</tr>
</tbody>
</table>

The two elements, when concatenated, provide the 16-byte Environment Identifier (EID) value.

**Output rule_array for option STATEXPT**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Base CCA services availability</td>
</tr>
</tbody>
</table>

A numeric character string containing a value to indicate whether base CCA services are available:  
*Value Meaning*  
0 Base CCA services are not available.  
1 Base CCA services are available. 

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>56-bit DES availability</td>
</tr>
</tbody>
</table>

A numeric character string containing a value to indicate whether 56-bit DES encryption is available:  
*Value Meaning*  
0 56-bit DES encryption is not available.  
1 56-bit DES encryption is available.
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>Triple-DES availability</td>
<td>A numeric character string containing a value to indicate whether Triple-DES encryption is available:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>SET services availability</td>
<td>A numeric character string containing a value to indicate whether SET (secure electronic transaction) services are available:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> The SET services are not supported in the Linux on IBM System z environment.</td>
</tr>
<tr>
<td>6</td>
<td>Maximum modulus for symmetric key encryption</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Output rule_array for option TIMEDATE**

| 1 | Date | The current date is returned as a character string of the form YYYYMMDD, where: |
|   |      | YYYY | Represents the year. |
|   |      | MM   | Represents the month (01 - 12). |
|   |      | DD   | Represents the day of the month (01 - 31). |

| 2 | Time | The current UTC time of day is returned as a character string of the form HHMMSS, where: |
|   |      | HH   | Represents the hour (0 - 23). |
|   |      | MM   | Represents the minute (0 - 59). |
|   |      | SS   | Represents second (0 - 59). |

| 3 | Day of the week | The day of the week is returned as a number between 1 (Sunday) and 7 (Saturday). |

**Output rule_array for option QPENDING**

| 1 | Change type (ASCII number) | An ASCII number that indicates the type of pending change stored in the adapter (if there is one) |
|   | Value | Meaning                     |
|   | none  | No pending change           |
|   | 1     | Role load                    |
|   | 2     | Profile load                 |
|   | 3     | Role delete                  |
|   | 4     | Profile delete               |
|   | 5     | Domain zeroize               |
|   | 6     | Enable                       |

| 2 | user ID (string) | A string of eight ASCII characters for the user ID of the user who initiated the pending change. |

**Output rule_array for option GET-UDX**

This keyword 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 78.

**Output rule_array for option STATKPRL**
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></td>
<td>This keyword causes a list of all the names provided for loaded operational key parts. The key parts are only loadable from the TKE using TKE-specific secured callable services. This keyword has verb data returned in the <code>verb_data</code> field. See [Verb data returned for Cryptographic Facility Query rule_array keywords on IBM System z](on page 78).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Output rule_array for option STATKPR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This keyword has INPUT verb_data as well as OUTPUT verb_data. A name for an operational key part is expected to be provided in the <code>verb_data</code> field, with an appropriately set <code>verb_data_length</code>. This name must match exactly a name returned by the STATKPRL keyword to list operational key part names, and have the same length. This keyword has verb data returned in the <code>verb_data</code> field. See [Verb data returned for Cryptographic Facility Query rule_array keywords on IBM System z](on page 78).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Output rule_array for option TKESTATE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TKE access enabled</td>
<td>Indicates whether a TKE can be used to administer this CEX3C. Values are: TKEPERM Allowed, TKEDENY Not allowed.</td>
</tr>
<tr>
<td></td>
<td>Output rule_array for option STATICSF</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Card serial number</td>
<td>Eight ASCII characters for the adapter serial number.</td>
</tr>
</tbody>
</table>
| 2              | DES new master-key register state | An ASCII number showing the state of the DES new master-key register:  
Value | Meaning  
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 | Meaning  
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 | Meaning  
1 | Invalid  
2 | Valid |
| 5              | PKA new master-key register state | An ASCII number showing the state of the PKA new master-key register:  
Value | Meaning  
1 | Empty  
2 | Partially full  
3 | Full |
| 6              | PKA current master-key register state | An ASCII number showing the state of the PKA current master-key register:  
Value | Meaning  
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>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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

**Output rule_array for option STATICSX**

This keyword 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 78.

| 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** | **Meaning** | 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** | **Meaning** | 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** | **Meaning** | 1 | Invalid |
|                |                                            | 2         | Valid        |

| 5              | PKA new master-key register state         | An ASCII number showing the state of the PKA new master-key register:        |
|                |                                            | **Value** | **Meaning** | 1 | Empty |
|                |                                            | 2         | Partially full |
|                |                                            | 3         | Full         |

| 6              | PKA current master-key register state     | An ASCII number showing the state of the PKA current master-key register:    |
|                |                                            | **Value** | **Meaning** | 1 | Invalid |
|                |                                            | 2         | Valid        |

| 7              | PKA old master-key register state         | An ASCII number showing the state of the PKA old master-key register:        |
|                |                                            | **Value** | **Meaning** | 1 | Invalid |
|                |                                            | 2         | Valid        |

**Output rule_array for option STATICSX**

This keyword 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 78.

| 1              | Card serial number                        | Eight ASCII characters for the adapter serial number                         |
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>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></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>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></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>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></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>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></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>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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</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></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>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></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>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></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 STATICSE

This keyword 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 78.
### 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></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>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></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>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></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>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></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>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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</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

The `verb_data` parameter is a pointer to a string variable containing data sent to the coprocessor for this verb or received from the coprocessor as a result of this verb. Its use depends on the options specified by the host application program.

The `verb_data` parameter is not used by this verb.
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 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.

STATKPRL: This keyword causes a list of the names of all the operational key parts loaded by the TKE into the CEX3C to be returned. Each name has a length of 64 bytes. If not enough space has been provided (using the verb_data_length field passed in by the application) to return the available list, a return code of 8 and a reason code of 72 is returned.

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 given in Table 12 on page 79.

Notes:
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 CEX3C communication.
Table 12. Output data format for STATKPR operational key parts

<table>
<thead>
<tr>
<th>Field name</th>
<th>Number of 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>X'00' The register is empty.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'01' The first DES key part was entered for the named key into this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'02' An intermediate DES key part (part after first) has been entered.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'03' The register contains a completed DES key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'11' The first AES key part was entered for the named key into this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'12' An intermediate AES key part (part after first) has been entered.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'13' The register contains a completed AES key.</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,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 bytes, indicating how much of the CV field to use. Note that CV is NOT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>

**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**.

**Notes:**

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 CEX3C communication.

Table 13. Output data format for STATICSA operational key parts

<table>
<thead>
<tr>
<th>Field name</th>
<th>Number of 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>Field name</td>
<td>Number of bytes</td>
<td>Field value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</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>
<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>
</tbody>
</table>
### Table 13. Output data format for STATICSA operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Number of bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SYM_CMK_VP_ID</code></td>
<td>2</td>
<td>X'0F07'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td><code>SYM_CMK_VP</code></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><code>SYM_NMK_VP_LEN</code></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><code>SYM_NMK_VP_ID</code></td>
<td>2</td>
<td>X'0F06'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td><code>SYM_NMK_VP</code></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><code>SYM_NMK_MKAP_LEN</code></td>
<td>2</td>
<td>12</td>
<td>Length in bytes of this Authentication pattern block in the <code>verb_data</code> (comprising this length field, the following ID field, and the field for the Authentication pattern).</td>
</tr>
<tr>
<td><code>SYM_NMK_MKAP_ID</code></td>
<td>2</td>
<td>X'0F09'</td>
<td>Hexadecimal identifier indicating the contents of the following Authentication pattern field.</td>
</tr>
<tr>
<td><code>SYM_NMK_MKAP</code></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><code>AES_OMK_VP_LEN</code></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><code>AES_OMK_VP_ID</code></td>
<td>2</td>
<td>X'0F0C'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td><code>AES_OMK_VP</code></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><code>AES_CMK_VP_LEN</code></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><code>AES_CMK_VP_ID</code></td>
<td>2</td>
<td>X'0F0B'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td><code>AES_CMK_VP</code></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><code>AES_NMK_VP_LEN</code></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><code>AES_NMK_VP_ID</code></td>
<td>2</td>
<td>X'0F0A'</td>
<td>Hexadecimal identifier indicating the contents of the following Verification pattern field.</td>
</tr>
<tr>
<td><code>AES_NMK_VP</code></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></td>
<td>204</td>
<td></td>
</tr>
</tbody>
</table>

**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 14.

Notes:
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 CEX3C communication.

Table 14. Output data format for STATICSE operational key parts

<table>
<thead>
<tr>
<th>Field name</th>
<th>Number of 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>
<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>
</tbody>
</table>
### Table 14. Output data format for STATICSE operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Number of bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYM_NM_K 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_NM_K 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>
<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>168</td>
<td></td>
<td></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 15 on page 84](#).

**Notes:**

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 CEX3C communication.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Number of 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>
<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>
</tbody>
</table>
Table 15. Output data format for STATICSX operational key parts (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Number of bytes</th>
<th>Field value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<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>156</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Restrictions
You cannot limit the number of returned rule_array elements. Table 11 on page 68 describes the number and meaning of the information in output rule_array elements.

Tip: Allocate a minimum of 30 rule_array elements to allow for extensions of the returned information.

Required commands
None

Usage notes
None

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 18.

The parameters for CSUACFQJ are shown here.
## Cryptographic Facility Query (CSUACFQ)

<table>
<thead>
<tr>
<th>FORMAT</th>
</tr>
</thead>
</table>
| public native void CSUACFQJ(  
|   hikmNativeInteger return_code,  
|   hikmNativeInteger reason_code,  
|   hikmNativeInteger exit_data_length,  
|   byte[] exit_data,  
|   hikmNativeInteger rule_array_count,  
|   byte[] rule_array,  
|   hikmNativeInteger verb_data_length,  
|   byte[] verb_data  
| ); |

---

**Cryptographic Facility Query (CSUACFQ)**

```java
public native void CSUACFQJ(  
    hikmNativeInteger return_code,  
    hikmNativeInteger reason_code,  
    hikmNativeInteger exit_data_length,  
    byte[] exit_data,  
    hikmNativeInteger rule_array_count,  
    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 via the STATCCA rule_array option.

The verb returns information elements in the version_data variable.

Format

CSUACFV(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    version_data_length,
    version_data
)

Parameters

Note that there is no rule_array keyword.

return_code

Direction: Output Type: Integer

The return_code parameter is a pointer to an integer value that expresses the general results of processing. See “Return code and reason code overview” on page 16 for more information about return codes.

reason_code

Direction: Output Type: Integer

The reason_code parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See “Return code and reason code overview” on page 16 for more information about reason codes.

exit_data_length

Direction: Input/Output Type: Integer

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

Direction: Input/Output Type: String

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 0.

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. The 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: `yyyymmdd`, where `yyyy` is the year, `mm` is the month, and `dd` is the day of the month.

Restrictions

None

Required commands

None

Usage notes

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 18.

The parameters for CSUACFVJ are shown here.

```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 Facility Version (CSUACFV)
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 92) to deallocate an allocated cryptographic resource.

Be sure to review "Multi-coprocessor capabilities" on page 30.

Format

```
CSUACRA(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    resource_name_length,
    resource_name)
```

Parameters

**return_code**

Direction: Output  
Type: Integer

The `return_code` parameter is a pointer to an integer value that expresses the general results of processing. See "Return code and reason code overview" on page 16 for more information about return codes.

**reason_code**

Direction: Output  
Type: Integer

The `reason_code` parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See "Return code and reason code overview" on page 16 for more information about reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: Integer
Cryptographic Resource Allocate (CSUACRA)

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 0.

**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 for this verb.

**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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptographic resource (required)</td>
<td></td>
</tr>
<tr>
<td>DEVICE</td>
<td>Specifies a CEX3C Coprocessor.</td>
</tr>
<tr>
<td>HCPUACLAR</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 30) and environment variables that influence CPACF support (see "Environment variables that affect CPACF usage" on page 9).

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
None

Required commands
None.

Usage notes
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 CSUACRA, specifying the correct HCPUACLR 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 9 for more affected verbs.

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 18.

The parameters for CSUACRAJ are shown here.

```java
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, deallocates 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 deallocating 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 30.

Format

```
CSUACRD(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   rule_array_count,
   rule_array,
   resource_name_length,
   resource_name)
```

Parameters

**return_code**

Direction: Output  
Type: Integer

The `return_code` parameter is a pointer to an integer value that expresses the general results of processing. See “Return code and reason code overview” on page 16 for more information about return codes.

**reason_code**

Direction: Output  
Type: Integer

The `reason_code` parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See “Return code and reason code overview” on page 16 for more information about reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String
Cryptographic Resource Deallocate (CSUACRD)

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 0.

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 for this verb.

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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVICE</td>
<td>Specifies a CEX3C Coprocessor.</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 variable that also impacts default card:
CSU_DEFAULT_ADAPTER (see Multi-coprocessor capabilities on page 30, and environment variables that influence CPACF support (see Environment variables that affect CPACF usage on page 9).

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.
Cryptographic Resource Deallocate (CSUACRD)

Restrictions
None

Required commands
None

Usage notes
To disable CPACF usage in your processing thread, make a call to CSUACRD, specifying the correct HCPUACL 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 9 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 18.

The parameters for CSUACRDJ are shown here.

```
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/CEX3C/keys directory.

Format

```c
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

**return_code**

Direction: Output  
Type: Integer

The `return_code` parameter is a pointer to an integer value that expresses the general results of processing. See "Return code and reason code overview" on page 16 for more information about return codes.

**reason_code**

Direction: Output  
Type: Integer

The `reason_code` parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See "Return code and reason code overview" on page 16 for more information about reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String

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 0.

**rule_array_count**

Direction: Input  
Type: Integer
Key Storage Initialization (CSNBKSI)

A pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 2 for this verb.

**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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master-key source (required)</td>
<td>Specifies the current symmetric master-key of the default cryptographic facility is to be used for the initialization.</td>
</tr>
<tr>
<td>CURRENT</td>
<td></td>
</tr>
<tr>
<td>Key-storage selection (one required)</td>
<td>Initialize AES key-storage.</td>
</tr>
<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.</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**
None

**Required commands**
The Key Storage Initialization verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**
None

**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 18.](#)

The parameters for CSNBKSIJ are shown here.

```
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);
```
Master Key Process (CSNBMKP)

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 101 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.

**Format**

```plaintext
CSNBMKP(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_part )
```
Parameters

return_code

Direction: Output  Type: Integer

The return_code parameter is a pointer to an integer value that expresses the general results of processing. See "Return code and reason code overview" on page 16 for more information about return codes.

reason_code

Direction: Output  Type: Integer

The reason_code parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See "Return code and reason code overview" on page 16 for more information about reason codes.

exit_data_length

Direction: Input/Output  Type: Integer

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

Direction: Input/Output  Type: String

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 0.

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, or 3 for this verb.

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 16.

Table 16. Keywords for Master Key Process control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptographic component (optional)</td>
<td></td>
</tr>
<tr>
<td>ADAPTER</td>
<td>Specifies the coprocessor. This is the default.</td>
</tr>
<tr>
<td>Master key register class (one, required)</td>
<td></td>
</tr>
</tbody>
</table>

See Note at the end of this table.
Table 16. Keywords for Master Key Process control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-MK</td>
<td>Specifies operation with the AES master-key registers.</td>
</tr>
<tr>
<td>SYM-MK</td>
<td>Specifies operation with the symmetric master-key registers.</td>
</tr>
<tr>
<td>ASYM-MK</td>
<td>Specifies operation with the asymmetric master-key registers.</td>
</tr>
</tbody>
</table>

**Master-key process (one, required)**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR</td>
<td>Specifies to clear the NMK register.</td>
</tr>
<tr>
<td>FIRST</td>
<td>Specifies to load the first key_part.</td>
</tr>
<tr>
<td>MIDDLE</td>
<td>Specifies to XOR the second, third, or other intermediate key_part into the NMK register.</td>
</tr>
<tr>
<td>LAST</td>
<td>Specifies to XOR the last key_part into the NMK register.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>Generates a random master-key value in the NMK register. This is not valid with master key register class AES-MK.</td>
</tr>
<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**

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 542).

For applications that use this verb:

- When writing your own application, you must link it with the /usr/lib64/libcsulccamk.so library.

**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 AES New Master Key Register</td>
</tr>
<tr>
<td></td>
<td>SYM-MK</td>
<td>X'0032'</td>
<td>Clear New Master Key Register</td>
</tr>
<tr>
<td></td>
<td>ASYM-MK</td>
<td>X'0060'</td>
<td>Clear New Asymmetric Master Key Register</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>FIRST</td>
<td>AES-MK</td>
<td>X'0125'</td>
<td>Load First AES Master Key Part</td>
</tr>
<tr>
<td></td>
<td>SYM-MK</td>
<td>X'0018'</td>
<td>Load First Master Key Part</td>
</tr>
<tr>
<td></td>
<td>ASYM-MK</td>
<td>X'0053'</td>
<td>Load First Asymmetric Master Key Part</td>
</tr>
<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 Master Key Parts</td>
</tr>
<tr>
<td></td>
<td>ASYM-MK</td>
<td>X'0054'</td>
<td>Combine Asymmetric Master Key Parts</td>
</tr>
<tr>
<td>RANDOM</td>
<td>SYM-MK</td>
<td>X'0020'</td>
<td>Generate Random Master Key</td>
</tr>
<tr>
<td></td>
<td>ASYM-MK</td>
<td>X'0120'</td>
<td>Generate Random Asymmetric Master Key</td>
</tr>
<tr>
<td>SET</td>
<td>AES-MK</td>
<td>X'0128'</td>
<td>Activate New AES Master Key (SET)</td>
</tr>
<tr>
<td></td>
<td>SYM-MK</td>
<td>X'001A'</td>
<td>Set Master Key</td>
</tr>
<tr>
<td></td>
<td>ASYM-MK</td>
<td>X'0057'</td>
<td>Set Asymmetric Master Key</td>
</tr>
</tbody>
</table>

Usage notes

None

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 /* weak */
E0 E0 E0 E0 F1 F1 F1 F1 /* weak */
01 FE 01 FE 01 FE 01 FE /* semi-weak */
FE 01 FE 01 FE 01 FE 01 FE /* semi-weak */
1F E0 1F E0 0E F1 0E F1 /* semi-weak */
E0 E0 1F E0 F1 01 F1 01 FE /* semi-weak */
01 0E 01 0F 01 0E 01 0E /* semi-weak */
1E 1F 01 1F 01 0E 01 0E /* semi-weak */
01 0E 01 0F 01 0E 01 0E /* semi-weak */
FE 0E E0 FE F1 FE F1 FE /* weak */
FE E0 FE E0 FE F1 FE F1 /* semi-weak */
1F IF 01 0E 0E 01 01 /* possibly semi-weak */
01 IF 1F 01 01 0E 0E 01 /* possibly semi-weak */
1F 01 01 0E 0E 01 01 /* possibly semi-weak */
01 01 1F 01 01 0E 0E 01 /* possibly semi-weak */
E0 E0 01 01 01 F1 F1 01 /* possibly semi-weak */
FE FE 01 01 FE F1 FE 01 /* possibly semi-weak */
FE E0 1F 01 FE F1 FE 01 /* possibly semi-weak */
FE FE 1F FE FE 0E 0E FE /* possibly semi-weak */
FE FE 01 FE SE FE F1 0E FE /* possibly semi-weak */
E0 1F FE 01 FE 0E 0E 01 /* possibly semi-weak */
E0 01 FE 1F F1 FE F1 FE /* possibly semi-weak */
01 E0 E0 01 01 01 F1 01 /* possibly semi-weak */
1F FE E0 01 0E FE F1 0E 01 /* possibly semi-weak */
01 FE FE 01 01 FE FE F1 0E 01 /* possibly semi-weak */
1F E0 E0 1F 0E F1 0E FE 01 /* possibly semi-weak */
01 FE FE 01 01 FE 0E F1 0E /* possibly semi-weak */
1F E0 FE 1F 01 FE 0E 0E 01 /* possibly semi-weak */
1F FE FE 01 0E FE 0E 0E /* possibly semi-weak */
E0 01 01 E0 F1 01 F1 01 /* possibly semi-weak */
```

Chapter 4. Using the CCA nodes and access control verbs  101
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 18.

The parameters for CSNBMKPJ are shown here.

```
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)

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

```
CSUARNT(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array )
```

Parameters

**return_code**

Direction: Output  Type: Integer

The `return_code` parameter is a pointer to an integer value that expresses the general results of processing. See “Return code and reason code overview” on page 16 for more information about return codes.

**reason_code**

Direction: Output  Type: Integer

The `reason_code` parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See “Return code and reason code overview” on page 16 for more information about reason codes.

**exit_data_length**

Direction: Input/Output  Type: Integer

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**

Direction: Input/Output  Type: String

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 0.

**rule_array_count**

Direction: Input  Type: Integer
Random Number Tests (CSUARNT)

A pointer to an integer variable containing the number of elements in the
rule_array variable. The value must be 1 for this verb.

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 17.

Table 17. Keywords for Random Number Tests control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test selection (one required)</td>
<td></td>
</tr>
<tr>
<td>FIPS-RNT</td>
<td>Perform the FIPS 140-1 specified test on the random number generation output.</td>
</tr>
<tr>
<td>KAT</td>
<td>Perform the FIPS 140-1 specified known-answer tests on DES, RSA, and SHA-1.</td>
</tr>
</tbody>
</table>

Restrictions
None

Required commands
None

Usage notes
None

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 18.

The parameters for CSUARNTJ are shown here.

```java
public native void CSUARNTJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array);
```
Chapter 5. Managing DES and AES cryptographic keys

This chapter describes the verbs that generate and maintain DES and AES 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 DES and AES key storage files.

This chapter describes the following verbs:
- "Clear Key Import (CSNBCKI)"
- "Control Vector Generate (CSNBCVG)" on page 108
- "Control Vector Translate (CSNBCVT)" on page 111
- "Cryptographic Variable Encipher (CSNBCVE)" on page 115
- "Data Key Export (CSNBDEKX)" on page 118
- "Data Key Import (CSNBDEKM)" on page 120
- "Diversified Key Generate (CSNBDEKG)" on page 122
- "Key Export (CSNBKEYX)" on page 127
- "Key Generate (CSNBKEYG)" on page 130
- "Key Import (CSNBKEYI)" on page 138
- "Key Part Import (CSNBKEYPI)" on page 142
- "Key Test (CSNBKEYT)" on page 146
- "Key Test Extended (CSNBKEYTX)" on page 151
- "Key Token Build (CSNBKTB)" on page 157
- "Key Token Change (CSNBKTC)" on page 161
- "Key Token Parse (CSNBKTP)" on page 164
- "Key Translate (CSNBKYT)" on page 169
- "Multiple Clear Key Import (CSNBCKM)" on page 171
- "PKA Decrypt (CSNDPKD)" on page 174
- "PKA Encrypt (CSNDPKE)" on page 178
- "Prohibit Export (CSNPBEX)" on page 182
- "Prohibit Export Extended (CSNPBEXX)" on page 184
- "Random Number Generate (CSNRNG)" on page 186
- "Random Number Generate Long (CSNRNGL)" on page 188
- "Remote Key Export (CSNRKX)" on page 398
- "Trusted Block Create (CSNTBC)" on page 408

DES and AES cryptographic key verbs

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. Clear Key Import 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. The 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 171.
Clear Key Import (CSNBCKI)

Format

```
CSNBCKI(  
  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  clear_key,  
  key_identifier )
```

Parameters

**return_code**
- Direction: Output  
- Type: Integer

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

**reason_code**
- Direction: Output  
- Type: Integer

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

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

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 16 describes the internal key token.

Restrictions

None

Required commands

This verb requires the Encipher Under Master Key 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
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 18.

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
);
```
Control Vector Generate (CSNBCVG)

Control Vector Generate (CSNBCVG)
The Control Vector Generate verb builds a control vector from keywords specified by the key_type and rule_array parameters.

Format

```
CSNBCVG(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_type,
    rule_array_count,
    rule_array,
    reserved,
    control_vector )
```

Parameters

return_code
Direction: Output Type: Integer
The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

reason_code
Direction: Output Type: Integer
The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

exit_data_length
Direction: Input/Output Type: Integer
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
Direction: Input/Output Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

key_type
Direction: Input Type: String
A string variable containing a keyword for the key type. The keyword is eight bytes in length, left justified, and padded on the right with space characters. It is taken from the following list:

- CIPHER
- DATAC
- IKEYXLAT
- OPINENC
- CVARDEC
- DATAM
- IMPORTER
- PINGEN
- CVARENC
- DATAMV
- IPINENC
- PINVER
- CVARPINE
- DECRYPT
- KEYGENKY
- SECMSG
- CVARXCVL
- DKYGENKY
- MAC
- CVARXCVR
- ENCRYPT
- MACVER
- DATA
- EXPORTER
- OKEYXLAT

| Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide | 108 |
Control Vector Generate (CSNBCVG)

**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 justified in 8-byte fields, and padded on the right with blanks. All keywords must be in contiguous storage. "Key Token Build (CSNBKTB)" on page 157 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. The `rule_array` keywords are shown here:

<table>
<thead>
<tr>
<th>AMEX-CSC</th>
<th>DKYL0</th>
<th>EPINGENA</th>
<th>LMTD-KEK</th>
<th>VISA-PVV</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSIX9.9</td>
<td>DKYL1</td>
<td>EPINVER</td>
<td>MIXED</td>
<td>Xlate</td>
</tr>
<tr>
<td>ANY</td>
<td>DKYL2</td>
<td>EXEX</td>
<td>NO-SPEC</td>
<td>XPORT-OK</td>
</tr>
<tr>
<td>ANY-MAC</td>
<td>DKYL3</td>
<td>EXPORT</td>
<td>NO-XPORT</td>
<td></td>
</tr>
<tr>
<td>CLR8-ENC</td>
<td>DKYL4</td>
<td>GBP-PIN</td>
<td>NOOFFSET</td>
<td></td>
</tr>
<tr>
<td>CPINENC</td>
<td>DKYL5</td>
<td>GBP-PINO</td>
<td>NOT-KEK</td>
<td></td>
</tr>
<tr>
<td>CPINGEN</td>
<td>DKYL6</td>
<td>IBM-PIN</td>
<td>OPEX</td>
<td></td>
</tr>
<tr>
<td>CPINGENA</td>
<td>DKYL7</td>
<td>IBM-PINO</td>
<td>OPIN</td>
<td></td>
</tr>
<tr>
<td>CVVKEY-A</td>
<td>DMAC</td>
<td>IMEX</td>
<td>PIN</td>
<td></td>
</tr>
<tr>
<td>CVVKEY-B</td>
<td>DMKEY</td>
<td>IMIM</td>
<td>REFORMAT</td>
<td></td>
</tr>
<tr>
<td>DALL</td>
<td>DMPIN</td>
<td>IMPORT</td>
<td>SINGLE</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>DMV</td>
<td>INBK-PIN</td>
<td>SMKEY</td>
<td></td>
</tr>
<tr>
<td>DDATA</td>
<td>DOUBLE</td>
<td>KEY-PART</td>
<td>SMPIN</td>
<td></td>
</tr>
<tr>
<td>DEXP</td>
<td>DPVR</td>
<td>KEYLN8</td>
<td>TRANSLAT</td>
<td></td>
</tr>
<tr>
<td>DIMP</td>
<td>EPINGEN</td>
<td>KEYLN16</td>
<td>UKPT</td>
<td></td>
</tr>
</tbody>
</table>

See Figure 3 on page 29 for the key usage keywords that can be specified for a given key type.

**Note:** CLR8-ENC or UKPT must be coded in `rule_array` when the KEYGENKY key type is coded. When the SECMSG `key_type` is coded, either SMKEY or SMPIN must be specified in the `rule_array`.

**reserved**
- **Direction:** Input
- **Type:** String

The `reserved` parameter must be a variable of eight bytes of X’00’.

**control_vector**
- **Direction:** Output
- **Type:** String

A 16-byte string variable in application storage where the verb returns the generated control vector.

**Restrictions**
- None

**Required commands**
- None

**Usage notes**
See the `key_type` parameter on page 158 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.
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 18.

The parameters for CSNBCVGJ are shown here.

```
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 Generate verb changes the control vector used to encipher an external DES key.

Format

```
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

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

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

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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**
Control Vector Translate (CSNBCVT)

array_key_left_identifier
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.

mask_array_left
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.

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. The 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 shown in Table 18.

### Table 18. Keywords for Control Vector Translate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity adjustment (one, optional)</td>
<td></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 (one, optional)</td>
<td></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>Keyword</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------</td>
<td>---------</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**

- 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**

None

**Required commands**

This verb requires the Translate Control Vector command (offset X'00D6') to be enabled in the active role.

**Usage notes**

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 18.

The parameters for CSNBCVTJ are shown here.
Control Vector Translate (CSNBCVT)

```
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,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] target_key_token);
```
Cryptographic Variable Encipher (CSNBCVE)

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.

The 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

```
CSNBCVE(
    return_code, 
    reason_code, 
    exit_data_length, 
    exit_data, 
    c-variable_encrypting_key_identifier, 
    text_length, 
    plaintext, 
    initialization_vector, 
    ciphertext )
```

Parameters

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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

**exit_data_length**

Direction: Input/Output  
Type: Integer
Cryptographic Variable Encipher (CSNBCVE)

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
Direction: Input/Output Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

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
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 is 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 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
This verb requires the Encipher Cryptovariable command (offset X'00DA') to be enabled in the active role.

Usage notes
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 18.

The parameters for CSNBCVEJ are shown here.
Cryptographic Variable Encipher (CSNBCVE)

```java
public native void CSNBCVEJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    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

```c
CSNBDKX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    source_key_identifier,
    exporter_key_identifier,
    target_key_identifier
)
```

Parameters

**return_code**
- Direction: Output
- Type: Integer
  
  The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

**reason_code**
- Direction: Output
- Type: Integer
  
  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” lists the reason codes.

**exit_data_length**
- Direction: Input/Output
- Type: Integer
  
  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**
- Direction: Input/Output
- Type: String
  
  The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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
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
This verb requires the Data Key Export command (offset X'010A') to be enabled in
the active role.

By also specifying the Unrestrict Data Key Export 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
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 18.

The parameters for CSNBDKXJ are shown here.

```
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)

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

```
CSNBDKM(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    source_key_token,
    importer_key_identifier,
    target_key_identifier )
```

Parameters

**return_code**

Direction: Output  
Type: Integer

The return code specifies the general result of the verb. Appendix A, "Return codes and reason codes" lists the return codes.

**reason_code**

Direction: Output  
Type: Integer

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" lists the reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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**
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**
This verb requires the Data Key Import command (offset X'0109') to be enabled in the active role.

By also specifying the Unrestrict Data Key Import 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**
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 18.

The parameters for CSNBDKMJ are shown here.

```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

```
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

**return_code**

Direction: Output  Type: Integer

The return code specifies the general result of the verb. [Appendix A, “Return
codes and reason codes”](#) lists the return codes.
reason_code

Direction: Output  Type: Integer

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" lists the reason codes.

exit_data_length

Direction: Input/Output  Type: Integer

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

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

rule_array_count

Direction: Input  Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. The only valid value is 1.

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 justified and padded on the right with blanks. Table 19 lists the keyword choices.

Table 19. Keywords for Diversified Key Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Method for generating or updating diversified keys</strong> (required)</td>
<td></td>
</tr>
<tr>
<td>CLR8-ENC</td>
<td>Specifies that eight bytes of clear data shall be multiply encrypted with the generating key. The generating_key_identifier must be a KEYGENKY key type with bit 19 of the control vector set to 1. The control vector in generated_key_identifier must specify a single-length key. The key type can be DATA, MAC, or MACVER. <strong>Note:</strong> CIPHER class keys are not supported.</td>
</tr>
<tr>
<td>TDES-DEC</td>
<td>Data supplied could be 8 or 16 bytes of clear data. If the generating_key_identifier specifies a single length key, then 8-bytes of data is TDES decrypted under the generating_key_identifier. If the generating_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.</td>
</tr>
</tbody>
</table>
Table 19. Keywords for Diversified Key Generate control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDES-ENC</td>
<td>Data supplied could be 8 or 16 bytes of clear data. If the <code>generated_key_identifier</code> specifies a single length key, then 8-bytes of data is TDES encrypted under the <code>generating_key_identifier</code>. If the <code>generated_key_identifier</code> specifies a double length key, then 16-bytes of data is TDES ECB mode encrypted under the <code>generating_key_identifier</code>. No formatting of data is done before encryption. The <code>generating_key_identifier</code> must be a DKYGENKY key type, with appropriate usage bits for the desired generated key. The <code>generated_key_identifier</code> can be a single or double length key, with a CV that is permitted by the <code>generating_key_identifier</code>.</td>
</tr>
<tr>
<td>TDES-XOR</td>
<td>This option combines the function of the existing TDES-ENC and SESS-XOR into one step.</td>
</tr>
<tr>
<td></td>
<td>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, MAC, MACVER, DATAM, DATAMV, SMPIN and SMKEY. Key type must be allowed by the generating key control vector.</td>
</tr>
<tr>
<td>TDESEMV2</td>
<td>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, MAC, MACVER, DATAM, DATAMV, SMPIN and SMKEY. Key type must be allowed by the generating key control vector.</td>
</tr>
<tr>
<td>TDESEMV4</td>
<td>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, MAC, MACVER, DATAM, DATAMV, SMPIN, and SMKEY. Key type must be allowed by the generating key control vector.</td>
</tr>
</tbody>
</table>

**Processing Method for updating a diversified key**

**SESS-XOR** 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, MAC/DATAM or MACVER/DATAMV key types.

**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**
Diversified Key Generate (CSNBDKG)

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

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

None

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>Generate Diversified Key (CLR8-ENC)</td>
</tr>
<tr>
<td>SESS-XOR</td>
<td>X'0043'</td>
<td>Generate Diversified Key (SESS-XOR)</td>
</tr>
<tr>
<td>TDES-DEC</td>
<td>X'0042'</td>
<td>Generate Diversified Key (TDES-DEC)</td>
</tr>
<tr>
<td>TDES-ENC</td>
<td>X'0041'</td>
<td>Generate Diversified Key (TDES-ENC)</td>
</tr>
<tr>
<td>TDES-XOR</td>
<td>X'0045'</td>
<td>Generate Diversified Key (TDES-XOR)</td>
</tr>
<tr>
<td>TDESEMV2 or</td>
<td>X'0046'</td>
<td>Generate Diversified Key (TDESEMVn)</td>
</tr>
<tr>
<td>TDESEMV4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When a key-generating key of key type DKYGENKY is specified with control vector bits (19 through 22) of B'1111', the Generate Diversified Key (DALL with DKYGENKY Key Type) 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 Enable DKG Single Length Keys and Equal Halves for TDES-ENC, TDES-DEC command (offset X'0044').

Usage notes

Refer to Appendix C, “Control vectors and changing control vectors with the Control Vector Translate verb,” on page 461 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 18.

The parameters for CSNBDKGJ are shown here.

```
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
);
```
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 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

```
CSNBKEX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_type,
    source_key_identifier,
    exporter_key_identifier,
    target_key_identifier )
```

Parameters

**return_code**
- Direction: Output
- Type: Integer
- The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

**reason_code**
- Direction: Output
- Type: Integer
- The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

**exit_data_length**
- Direction: Input/Output
- Type: Integer
- 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**
- Direction: Input/Output
- Type: String
- The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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-justified 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:

- CIPHER EXPORTER OPINENC
- DATA IMPORTER PINGEN
- DATAC IKEYXLAT PINVER
- DATAM IPINENC TOKEN
- DATAMV MAC
- DECRYPTER MACVER
- ENCRYPTER OKEYXLAT

For information about the meaning of the key types, see Table 1 on page 27.

source_key_identifier
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 24 and the NOCV bit is shown in Table 102 on page 431.

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
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
This verb requires the Reencipher from Master Key command (offset X'0013') to be enabled in the active role.
By also specifying the Unrestrict Reencipher from Master Key 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).

Usage notes
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 18.

The parameters for CSNBKEXJ are shown here.

```
FORMAT
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 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.

Format

```c
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

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

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

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**key_form**
Direction: Input
Type: String
A 4-byte keyword that defines the type of key(s) 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-justified, and padded with blanks.

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 20 for their meanings.

**Table 20. Keywords for the Key Generate verb key_form parameter**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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 23 for valid key combinations.
Key Generate (CSNBKGN)

**key_length**

Direction: Input  Type: String

An 8-byte value that defines the length of the key as being 8, 16 or 24 bytes.
The keyword must be left-justified and padded on the right with blanks. You
must supply one of the key length values in the key_length parameter. Keys of
length eight byte are valid only for the DES key type.

DES and AES keys can be 16 or 24 bytes in length.

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 (referred to as replicated key values), specify
key_length as SINGLE, SINGLE-R, or KEYLN8. If you want different values to
be the basis of each key half, specify key_length as DOUBLE or KEYLN16.

Triple-length (24-byte) keys have three 8-byte key parts. This key length is valid
only for DATA keys. To generate a triple-length DATA key with three different
values to be the basis of each key part, specify key_length as KEYLN24.

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.

Use SINGLE or SINGLE-R if you want to create a 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.

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 21 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>
<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>AES TOKEN</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></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MAC VER</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATA</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>DATAM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATAMV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPORTER</td>
<td>Y</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>Y</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>Y</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>Y</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CIPHER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DECRYPTER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ENCIPHER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 21. Key Generate - key lengths for each key type
### Table 21. Key Generate - key lengths for each key type (continued)

<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>IPINENC</td>
<td>Y</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPINENC</td>
<td>Y</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PINGEN</td>
<td>Y</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PINVER</td>
<td>Y</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARDEC*</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARENC*</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARPINE*</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARXCVL*</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARXCVR*</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DKYGENKY*</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYGENKY*</td>
<td>X</td>
<td>X</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:

- CIPHER, DATA, DATAM, DATAMV, DATAXLAT, DECIPHER, ENCIPHER, EXPORTER, IKEYXLAT, IMPORTER, IPINENC, MAC, MACVER, OKEYXLAT, OPINENC, PINGEN, and PINVER
- or the keyword TOKEN

For information on the meaning of the key types, see Table 1 on page 27.

Use the `key_type_1` parameter for the first, or only key, that you want generated. The keyword must be left-justified 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 22 on page 136 and Table 23.

**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 22 and Table 23 for valid key type and key form combinations.

**key_type_2**

**Direction:** Input  
**Type:** String

An 8-byte keyword from the following group:
Key Generate (CSNBKGN)

- CIPHER, DATA, DATAM, DATAMV, DATAXLAT, DECIPHER, ENCIPHER, EXPORTER, IKEYXLAT, IMPORTER, IPINENC, MAC, MACVER, OKEYXLAT, OPINENC, PINGEN, and PINVER
- or the keyword TOKEN

For information on the meaning of the key types, see Table 1 on page 27.

Use the key_type_2 parameter for a key pair, which is shown in Table 23 on page 136. The keyword must be left-justified 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 22 on page 136 and Table 23 on page 136 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-justified 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 24 and the NOCV bit is shown in Table 102 on page 431.

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-justified 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 24 and the NOCV bit is shown in Table 102 on page 431.

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.

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.

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
None

### 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>Generate Key Set</td>
</tr>
<tr>
<td>X'008E'</td>
<td>Generate Key</td>
</tr>
<tr>
<td>X'00D7'</td>
<td>Generate Key Set Extended</td>
</tr>
<tr>
<td>X'00DB'</td>
<td>Replicate Key</td>
</tr>
</tbody>
</table>

**Note:** A role with offset X'00DB' enabled can also use the Remote Key Export verb.
Usage notes

Table 22 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.

Note: Not all key types are valid on all hardware. See Table 1 on page 27.

For key type AES, only key form OP is supported. AES keys cannot be generated in pairs.

Table 22. 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>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 23 shows the valid key type and key form combinations for a key pair. 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.

Table 23. 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>ENCIIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CVARDEC*</td>
<td>CVARENC*</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CVARDEC*</td>
<td>CVARPINE*</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CVARENC*</td>
<td>CVARDEC*</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CVARENC*</td>
<td>CVARXCVL*</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CVARENC*</td>
<td>CVARXCVR*</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CVARPINE*</td>
<td>CVARDEC*</td>
<td>X</td>
<td></td>
<td></td>
<td>X</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>DECIIPHER</td>
<td>CIPHER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
### Table 23. 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>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>IKEYXLAT</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>EXPORTER</td>
<td>IMPORTER</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>EXPORTER</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>OKEYXLAT</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>EXPORTER</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>IMPORTER</td>
<td>OKEYXLAT</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>IPINENC</td>
<td>OPINENC</td>
<td>X</td>
<td>X</td>
<td>X</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></td>
<td>X</td>
</tr>
<tr>
<td>OKEYXLAT</td>
<td>IMPORTER</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OPINENC</td>
<td>IPINENC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OPINENC</td>
<td>OPINENC</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PINVER</td>
<td>PINGEN</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PINGEN</td>
<td>PINVER</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 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 18.

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 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**

```
CSNBKIM(  
  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  key_type,  
  source_key_identifier,  
  importer_key_identifier,  
  target_key_identifier  )
```

**Parameters**

**return_code**

- Direction: Output
- Type: Integer

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

**reason_code**

- Direction: Output
- Type: Integer

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

**exit_data_length**

- Direction: Input/Output
- Type: Integer

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**

- Direction: Input/Output
- Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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-justified 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.

TOKEN is never allowed when the `importer_key_identifier` parameter is NOCV.

Key type values for the Key Import verb are:

- CIPHER  
- EXPORTER  
- OKEYXLAT  
- DATA  
- IMPORTER  
- OPINENC  
- DATAC  
- IKEYXLAT  
- PINGEN  
- DATAM  
- IPINENC  
- PINVER  
- DATAMV  
- MAC  
- TOKEN  
- DECRYPTER  
- MACVER  
- ENCIPHER  
- MACD

For information on the meaning of the key types, see Table 1 on page 27.

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 104 on page 433.

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-justified 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.
Key Import (CSNBKIM)

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
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.

Required commands
This verb requires the Reencipher to Master Key command (offset X'0012') to be enabled in the active role.

By also enabling the Unrestrict Reencipher To Master Key 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
Use of NOCV keys are controlled by an access control point in the CEX3C. 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 CEX3C. 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 18.

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
);
```
Key Part Import (CSNBKPI)

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 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 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 (CSNBKYT) and Key Test Extended (CSNBKYTX) 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 via 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

```
CSNBKPI(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_part,
    key_identifier )
```

Parameters

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

The reason code provides additional information regarding the result of the verb.
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" lists the reason codes.

**exit_data_length**

Direction: Input/Output  Type: Integer

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**

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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

The keyword that provides control information to the verb. The keywords must be eight bytes of contiguous storage with the keyword left-justified in its 8-byte location and padded on the right with blanks. Table 24 describes the choices for this keyword.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Part (Required)</strong></td>
<td></td>
</tr>
<tr>
<td>FIRST</td>
<td>This keyword specifies that an initial key part is being entered. The 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>
</tbody>
</table>
Table 24. Keywords for Key Part Import control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETRKPR</td>
<td>A key label must be passed as the key_identifier. 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_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-justified and padded on the right with zeros. This field is ignored if COMPLETE is specified.

**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**

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**

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>Load First Key Part</td>
</tr>
<tr>
<td>ADD-PART</td>
<td>X'0278'</td>
<td>Add Key Part</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>X'0279'</td>
<td>Complete Key Part</td>
</tr>
<tr>
<td>MIDDLE or LAST</td>
<td>X'001C'</td>
<td>Combine Key Parts</td>
</tr>
<tr>
<td>MIDDLE or LAST</td>
<td>X'027A'</td>
<td>Unrestrict Combine Key Part</td>
</tr>
</tbody>
</table>
Usage notes
None

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 18.

The parameters for CSNBKPIJ are shown here.

<table>
<thead>
<tr>
<th>FORMAT</th>
</tr>
</thead>
</table>
| 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 ); |
Use the Key Test verb to generate or verify a secure, cryptographic verification pattern for keys. 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.

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 493.

With the default method, the verb generates a verification pattern and it creates and cryptographically processes a random number. The verb returns the random number with the verification pattern.

When the verb tests a verification pattern against a key, you must supply the random number and the verification pattern from a previous call to Key Test. The verb returns the verification result in the return and reason codes. See Table 25 for details.

<table>
<thead>
<tr>
<th>Table 25. Verification pattern input and output</th>
</tr>
</thead>
<tbody>
<tr>
<td>For keyword GENERATE</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>On input</td>
</tr>
<tr>
<td>ENC-ZERO</td>
</tr>
<tr>
<td>MDC-4</td>
</tr>
<tr>
<td>SHA-1</td>
</tr>
<tr>
<td>SHA-256</td>
</tr>
</tbody>
</table>

For keyword VERIFY

| Method | random_number variable | verification_pattern variable |
| On input | On output | On input | On output |
Table 25. Verification pattern input and output (continued)

<table>
<thead>
<tr>
<th>ENC-ZERO</th>
<th>Unused</th>
<th>Unused</th>
<th>Contains the 4-byte verification pattern in the high-order four bytes of the variable. The low-order four bytes are unspecified.</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDC-4</td>
<td>Contains the lower (leftmost) eight bytes of the MDC-4 hash.</td>
<td>Unused</td>
<td>Contains the upper (rightmost) eight bytes of the MDC-4 hash.</td>
<td>Unused</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Contains the lower (leftmost) eight bytes of the SHA-1 verification pattern.</td>
<td>Unused</td>
<td>Contains the upper eight bytes of the SHA-1 verification pattern.</td>
<td>Unused</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Unused</td>
<td>Unused</td>
<td>SHA-256 based verification pattern</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Format

```c
CSNBKYT(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier,
    random_number,
    verification_pattern
)
```

Parameters

**return_code**

- Direction: Output
- Type: Integer
- The return code specifies the general result of the verb. [Appendix A, “Return codes and reason codes”](#) lists the return codes.

**reason_code**

- Direction: Output
- Type: Integer
- 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”](#) lists the reason codes.

**exit_data_length**

- Direction: Input/Output
- Type: Integer
Key Test (CSNBKYP)

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
Direction: Input/Output  Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

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 can be 2, 3, 4, or 5.

rule_array
Direction: Input  Type: String
Two to five keywords provide control information to the verb. Table 26 lists the keywords. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks.

### Table 26. Keywords for Key Test control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Rule</strong> (required)</td>
<td></td>
</tr>
<tr>
<td>KEY-CLR</td>
<td>Specifies the key supplied in <code>key_identifier</code> is a single-length clear key.</td>
</tr>
<tr>
<td>KEY-CLRRD</td>
<td>Specifies the key supplied in <code>key_identifier</code> is a double-length clear key.</td>
</tr>
<tr>
<td>KEY-ENC</td>
<td>Specifies the key supplied in <code>key_identifier</code> is a single-length encrypted key.</td>
</tr>
<tr>
<td>KEY-ENCRRD</td>
<td>Specifies 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>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><strong>Master-key selector</strong> (one, optional). Use only with KEY-KM, KEY-NKM, or KEY-OKM keywords.</td>
<td></td>
</tr>
<tr>
<td>AES-MK</td>
<td>Process one of the AES master-key registers.</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>Process Rule</strong> (required)</td>
<td></td>
</tr>
</tbody>
</table>
Table 26. Keywords for Key Test control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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 (optional)**

<table>
<thead>
<tr>
<th>Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**Verification Process Rule (one, optional)**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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, 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>

**key_identifier**

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-justified.

**Note:** If you supply a key label for this parameter, it must be unique in the key storage file.

**random_number**

Direction: Input/Output  Type: String

This is an 8-byte field that contains a random number supplied as input for the test pattern verification process and returned as output with the test pattern generation process. With the ENC-ZERO method, the random number is not used, but it still must be provided.

**verification_pattern**

Direction: Input/Output  Type: String

This is an 8-byte field that contains a verification pattern supplied as input for the test pattern verification process and returned as output with the test pattern generation process. With the ENC-ZERO method, the high-order four bytes contain the verification data. For more detail, see "Cryptographic key-verification techniques" on page 493.
Restricted
None

Required commands
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

Usage notes
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 and 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.

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 18.

The parameters for CSNBKYTJ are shown here.

```
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[] random_number,
                             byte[] verification_pattern );
```
Key Test Extended (CSNBKYTX)

This verb is essentially the same as "Key Test (CSNBKYT)" on page 146, except:

- 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 146 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. The 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.

The 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 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 493. 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. The 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.
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 25 on page 146 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**

```c
CSNBKYTX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier,
    random_number,
    verification_pattern )
```

**Parameters**

**return_code**
- Direction: Output  
- Type: Integer  
- The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

**reason_code**
- Direction: Output  
- Type: Integer
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" lists the reason codes.

**exit_data_length**

Direction: Input/Output  Type: Integer

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**

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 can be 2, 3, 4, or 5.

**rule_array**

Direction: Input  Type: Array

Between two and five keywords provide control information to the verb. Table 27 lists the keywords. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks.

Table 27. Keywords for Key Test Extended control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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 the key supplied in <code>key_identifier</code> is a single-length encrypted key.</td>
</tr>
<tr>
<td>KEY-ENCD</td>
<td>Specifies 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>
</tbody>
</table>
### Table 27. Keywords for Key Test Extended control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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 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 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><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 to use 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 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, 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>

**key_identifier**

Direction: **Input** \hspace{1cm} Type: **String**

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** \hspace{1cm} 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** \hspace{1cm} 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.
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 493.

**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**

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**

This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**

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 18.

The parameters for CSNBKYTXJ are shown here.
Key Test Extended (CSNBKYTX)

```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 an external or internal key-token in application storage from information you supply.

The verb can include a control vector 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

```
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,
    reserved_3,
    control_vector,
    reserved_4,
    reserved_5,
    reserved_6,
    masterkey_verify_parm )
```

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

`return_code`

Direction: Output  
Type: Integer  
The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

`reason_code`

Direction: Output  
Type: Integer  
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” lists the reason codes.

`exit_data_length`

Direction: Input/Output  
Type: Integer  
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.
Key Token Build (CSNBKTB)

exit_data
Direction: Input/Output       Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

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
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 | IKEYXLAT | OPINENC |
| CVARDEC | DATAM | IMPORTER | PINGEN |
| CVARENC | DATAMV | IPINENC | PINVER |
| CVARPINE | DECIPHER | KEYGENKY | SECMSG |
| CVARXCVL | DKYGENKY | MAC |
| CVARXCVR | ENCIPHER | MACVER |
| DATA | EXPORTER | OKEYXLAT |

Specify the USE-CV keyword to indicate that the key type should be obtained from the control_vector variable.

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

One to four keywords that provide control information to the verb. See Table 28 for a list. The keywords must be in contiguous storage with each of the keywords left-justified 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.

Table 28. Keywords for Key Token Build Control Information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token type</td>
<td>(one required)</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>An external key token.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>An internal key token.</td>
</tr>
<tr>
<td>Token algorithm</td>
<td>(one, optional)</td>
</tr>
<tr>
<td>AES</td>
<td>An AES key.</td>
</tr>
<tr>
<td>DES</td>
<td>A DES key.</td>
</tr>
</tbody>
</table>
### Table 28. Keywords for Key Token Build Control Information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Status</strong> (optional) Not valid for CLRDES</td>
<td></td>
</tr>
<tr>
<td>KEY</td>
<td>The key token to build will contain an encrypted key. The <code>key_value</code> parameter identifies the field that contains the key.</td>
</tr>
<tr>
<td>NO-KEY</td>
<td>The key token to build will not contain a key. This is the default key status.</td>
</tr>
<tr>
<td><strong>CV Source</strong> (optional) Not valid for CLRDES</td>
<td></td>
</tr>
<tr>
<td>CV</td>
<td>The verb is to obtain the control vector from the variable identified by the <code>control_vector</code> parameter.</td>
</tr>
<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>
</tbody>
</table>

See Figure 3 on page 29 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.

| AMEX-CSC | DKYL0 | EPINGENA | LMTD-KEK | VISA-PPV |
| ANSI9.9 | DKYL1 | EPINVER | MIXED | XLATE |
| ANY | DKYL2 | EXEX | NO-SPEC | XPORT-OK |
| ANY-MAC | DKYL3 | EXPORT | NO-XPORT |
| CLR8-ENC | DKYL4 | GBP-PIN | NULLSET |
| CPINENC | DKYL5 | GBP-PINO | NOT-KEK |
| CPINGEN | DKYL6 | IBM-PIN | OPEX |
| CPINGENA | DKYL7 | IBM-PINO | OPIM |
| CVKEY-A | DMAC | IMEX | PIN |
| CVKEY-B | DMKEY | IMIM | REFORMAT |
| DALL | DMPIN | IMPORT | SINGLE |
| DATA | DMV | INBK-PIN | SMKEY |
| DDATA | DOUBLE | KEY-PART | SMIP |
| DEXP | DPVR | KEYLN8 | TRANSLAT |
| DIMP | EPINGEN | KEYLN16 | UKPT |

#### `key_value`

**Direction:** Input  
**Type:** String

This parameter is a string variable containing the encrypted key-value incorporated into the encrypted-key portion of the key token if you use the KEY `rule_array` keyword. Single-length keys must be left-aligned in the variable and padded on the right (low-order) with eight bytes of 'X00'.

#### `control_vector`

**Direction:** Input  
**Type:** String

A pointer to a 16-byte string variable. If this parameter is specified, and you use the CV `rule_array` keyword, the variable is copied to the control vector field of the key token. See “Control vector table” on page 461 for additional information.

#### `masterkey_verify_parm`

**Direction:** Input  
**Type:** String

A pointer to an 8-byte string variable. The value is inserted into the key token when you specify both the KEY and INTERNAL keywords in `rule_array`. 
Key Token Build (CSNBKTB)

Restrictions
None

Required commands
None

Usage notes
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 18.

The parameters for CSNBKTBJ are shown here.

```
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[] master_key_verification_pattern,
    hikmNativeInteger reserved1,
    byte[] reserved2,
    byte[] control_vector,
    byte[] reserved3,
    hikmNativeInteger reserved4,
    byte[] reserved5,
    byte[] reserved6 );
```
Key Token Change (CSNBKTC)

Use the Key Token Change verb to re-encipher 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.

Notes:
1. An application system is responsible for keeping all of its keys in a usable form. When the master key is changed, the CEX3C 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 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

```c
CSNBKTC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier )
```

Parameters

return_code

Direction: Output  Type: Integer

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

reason_code

Direction: Output  Type: Integer

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” lists the reason codes.

exit_data_length

Direction: Input/Output  Type: Integer

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

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

rule_array_count
Key Token Change (CSNBKTC)

**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 for this verb.

**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 listed in [Table 29](#).

*Table 29. Keywords for Key Token Change control information*

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCMK</td>
<td>Re-enciphers a DES 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 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 &quot;Storage on z/OS (RTNMK-focused)&quot; on page 236, it is not valid to pass a <em>key_identifier</em> 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 &quot;Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z&quot; on page 235.</td>
</tr>
</tbody>
</table>
Table 29. Keywords for Key Token Change control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm (optional)</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>Specifies that the key token is for an AES key.</td>
</tr>
<tr>
<td>DES</td>
<td>Specifies that the key token is for a DES key. This is the default.</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.

**Restrictions**

None

**Required commands**

If you specify the RTCMK keyword, the Key Token Change verb requires the re-encipher to Current Master Key command (offset X'0090') to be enabled in the active role.

**Usage notes**

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 18.

The parameters for CSNBKTCJ are shown here.

```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 Parse (CSNBKTP)

Key Token Parse (CSNBKTP)

The Key Token Parse verb disassembles a key token into separate pieces of information. The 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.

The 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

```
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

return_code

Direction: Output          Type: Integer

The return_code parameter is a pointer to an integer value that expresses the general results of processing. See “Return code and reason code overview” on page 16 for more information about return codes.

reason_code

Direction: Output          Type: Integer

The reason_code parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See “Return code and reason code overview” on page 16 for more information about reason codes.

exit_data_length

Direction: Input/Output    Type: Integer
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`

**Direction:** Input/Output  
**Type:** String  

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 0.

### `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_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:

<table>
<thead>
<tr>
<th>CIPHER</th>
<th>DATAC</th>
<th>IKEYXLAT</th>
<th>OPINENC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVARDEC</td>
<td>DATAM</td>
<td>IMPORTER</td>
<td>PINGEN</td>
</tr>
<tr>
<td>CVARENC</td>
<td>DATAMV</td>
<td>IPINENC</td>
<td>PINVER</td>
</tr>
<tr>
<td>CVARPINE</td>
<td>DECIPHER</td>
<td>KEYGENKY</td>
<td>SECMSG</td>
</tr>
<tr>
<td>CVARXCVL</td>
<td>DKYGENKY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVARXCVR</td>
<td>ENCIIPHER</td>
<td>MACVER</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>EXPORTER</td>
<td>OKEYXLAT</td>
<td></td>
</tr>
</tbody>
</table>

### `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 and should be a minimum of 20 for this verb.

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 described in Table 30 on page 166.
Table 30. Keywords for Key Token Parse control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Token type</strong></td>
<td>(one returned)</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></td>
<td>(one returned)</td>
</tr>
<tr>
<td>KEY</td>
<td>Indicates the key token contains a key. The key_value 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>Control-vector (CV) status</strong></td>
<td>(one returned)</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>

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 EPINGENA LMTD-KEK VISA-PVV
ANSIX9.9 DKYL1 EPINVER MIXED XLATE
ANY DKYL2 EXEX NO-SPEC XPORT-OK
ANY-MAC DKYL3 EXPORT NO-XPORT
CL8-ENC DKYL4 GBP-PIN NOOFFSET
CPINENC DKYL5 GBP-PINO NOT-KEK
CPINGEN DKYL6 IBM-PIN OPEX
CPINGENA DKYL7 IBM-PINO OPIM
CVVKEY-A DMAC IMEX PIN
CVVKEY-B DMKEY IMIM REFORMAT
DALL DMPIN IMPORT SINGLE
DATA DMV INBK-PIN SMKEY
DDATA DOUBLE KEY-PART SMPIN
DEXP DPVR KEYLN8 TRANSLAT
DIMP EPINGEN KEYLN16 UKPT

**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.

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

None

Required commands

None

Usage notes

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 18.

The parameters for CSNBKTPJ are shown here.
Key Token Parse (CSNBKTP)

```
<table>
<thead>
<tr>
<th>FORMAT</th>
</tr>
</thead>
</table>
| 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,
|     byte[] reserved_field_3,
|     byte[] control_vector,
|     byte[] reserved_field_4,
|     hikmNativeInteger reserved_field_5,
|     byte[] reserved_field_6,
|     byte[] master_key_verification_pattern_v0); |
```
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**

```
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**

**return_code**

- Direction: Output
- Type: Integer

The *return_code* specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

**reason_code**

- Direction: Output
- Type: Integer

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

**exit_data_length**

- Direction: Input/Output
- Type: Integer

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**

- Direction: Input/Output
- Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 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 1.

**output key token**

Direction: Output  Type: String

A 64-byte string variable containing an external key token. The external key
token contains the re-enciphered key.

**Restrictions**

Triple length DATA key tokens are not supported.

**Required commands**

This verb requires the Translate Key command (offset X'001F') to be enabled in the
active role.

**Usage notes**

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 18.

The parameters for CSNBKTRJ are shown here.

```
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
);```

---

**FORMAT**

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 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 1.

**output_key_token**

**Direction**: Output  **Type**: String

A 64-byte string variable containing an external key token. The external key
token contains the re-enciphered key.

**Restrictions**

Triple length DATA key tokens are not supported.

**Required commands**

This verb requires the Translate Key command (offset X'001F') to be enabled in the
active role.

**Usage notes**

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 18.

The parameters for CSNBKTRJ are shown here.

```
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
);```

---
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

```c
CSNBCKM(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    clear_key_length,
    clear_key,
    key_identifier_length,
    key_identifier
)
```

Parameters

**return_code**
- Direction: Output
- Type: Integer

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

**reason_code**
- Direction: Output
- Type: Integer

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

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

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**
- Direction: Input
- Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. The rule_array_count parameter must be 0 or 1.

**rule_array**
- Direction: Input
- Type: String
Multiple Clear Key Import (CSNBCKM)

Zero or one keyword that supplies control information to the verb. The keyword must be in eight bytes of contiguous storage, left-justified and padded on the right with blanks. Refer to Table 31 for a list of keywords.

Table 31. Keywords for Multiple Clear Key Import control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm (optional)</td>
<td></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>
</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.

key_identifier_length
Direction: Input/Output     Type: Integer
The byte length of the key_identifier parameter. This must be exactly 64 bytes.

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 429 describes the key tokens.

Restrictions
None

Required commands
This verb requires the following commands to be enabled in the active role based on the algorithm:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>X'0129'</td>
<td>Encipher Under AES Master Key</td>
</tr>
<tr>
<td>DES</td>
<td>X'00C3'</td>
<td>Encipher Under DES Master Key</td>
</tr>
</tbody>
</table>

Note: Note: A role with offset X'00C3' can also use the Clear Key Import verb.

Usage notes
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 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 18.

The parameters for CSNBCKMJ are shown here.

```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);
```
PKA Decrypt (CSNDPKD)

Use this verb to decrypt (unwrap) a formatted key value. The 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 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

```
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

**return_code**
- Direction: Output
- Type: Integer

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

**reason_code**
- Direction: Output
- Type: Integer

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

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

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.
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-justified in an 8-byte field and padded on the right with blanks. See
Table 32

Table 32: Keywords for PKA Decrypt control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery Method (required)</td>
<td>specifies the method to use to recover the key value.</td>
</tr>
<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-justified within the PKA_enciphered_keyvalue parameter.

data_structure_length
Direction: Input Type: Integer
The 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
PKA Decrypt (CSNDPKD)

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 format. The corresponding public key was used to wrap the key value.

target_keyvalue_length
Direction: Input/Output Type: Integer
The length of the target_keyvalue parameter. The maximum size that you can specify is 256 bytes. On return, this field is updated with the actual length of target_keyvalue.

If ZERO-PAD is specified, this length will be the same as the PKA_enciphered_keyvalue_length which is equal to the RSA modulus byte length.

target_keyvalue
Direction: Output Type: String
This field will contain the decrypted, parsed key value. If ZERO-PAD is specified, the decrypted key value, including leading zeros, will be returned.

Restrictions
The exponent of the RSA public key must be odd.

Required commands
This verb requires the PKA Decipher Clear Key command (offset X'011F') to be enabled in the active role.

Usage notes
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.

JNI version
This verb has a Java Native Interface (JNI) version, which is named CSNDPKDlj.
See "Building Java applications to use with the CCA JNI" on page 18.

The parameters for CSNDPKD are shown here.
PKA Decrypt (CSNDPKD)

**FORMAT**

```java
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
);
```
PKA Encrypt (CSNDPKE)

This verb encrypts a supplied clear key value under an RSA public key. The rule_array keyword specifies the format of the key prior to encryption.

Format

```c
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

**return_code**

- Direction: Output
- Type: Integer

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

**reason_code**

- Direction: Output
- Type: Integer

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

**exit_data_length**

- Direction: Input/Output
- Type: Integer

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**

- Direction: Input/Output
- Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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-justified in an 8-byte field and padded on the right with blanks. See Table 33.

Table 33. Keywords for PKA Encrypt control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatting Method (required)</td>
<td>specifies the method to use to format the key value prior to encryption.</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>
<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 180.

**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**
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-justified within the `PKA_enciphered_keyvalue` parameter.

**Restrictions**

**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**

This verb requires the PKA Encipher Clear Key command (offset X'011E') to be enabled in the active role.

**Usage notes**

- 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 18.

The parameters for CSNDPKEJ are shown here.
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)

Prohibit Export (CSNBPEX)

Use this verb to modify an operational key so that it cannot be exported.

Format

```plaintext
CSNBPEX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_identifier )
```

Parameters

**return_code**
- Direction: Output
- Type: Integer

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

**reason_code**
- Direction: Output
- Type: Integer

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

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

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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

None

Required commands

This verb requires the Lower Export Authority command (offset X'00CD') to be enabled in the active role.

Usage notes

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 18.

The parameters for CSNBPEXJ are shown here.

```
FORMAT
public native void CSNBPEXJ(
     hikmNativeInteger return_code,
     hikmNativeInteger reason_code,
     hikmNativeInteger exit_data_length,
     byte[] exit_data,
     byte[] key_identifier);
```
Prohibit Export Extended (CSNBPEXX)

Use this verb to modify an exportable external CCA DES key-token so that its key can no longer be exported.

The 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

```c
CSNBPEXX( return_code, reason_code, exit_data_length, exit_data, source_key_token, KEK_key_identifier )
```

Parameters

**return_code**

Direction: Output  
Type: Integer  
The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

**reason_code**

Direction: Output  
Type: Integer  
The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer  
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**

Direction: Input/Output  
Type: String  
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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
This verb does not support version X'10' external DES key tokens (RKX key tokens).

Required commands
This verb requires the Lower Export Authority, Extended command (offset X'0301') to be enabled in the active role.

Usage notes
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 18.

The parameters for CSNBPEXXJ are shown here.

```
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);
```
Random Number Generate (CSNBRNG)

Random Number Generate (CSNBRNG)

The verb uses the cryptographic feature to generate a cryptographic-quality random number.

Format

```
CSNBRNG(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    form,
    random_number )
```

Parameters

return_code

Direction: Output          Type: Integer

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

reason_code

Direction: Output          Type: Integer

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

exit_data_length

Direction: Input/Output    Type: Integer

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

Direction: Input/Output    Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

form

Direction: Input            Type: String

The 8-byte keyword that defines the characteristics of the random number should be left-justified and padded on the right with blanks. The keywords are listed in Table 34.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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**
None

**Required commands**
This verb requires the Generate Key command (offset X'008E') to be enabled in the active role.

**Usage notes**
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 18.

The parameters for CSNBRNGJ are shown here.

```java
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)

The verb uses the cryptographic feature to generate a cryptographic-quality random number from 1 to 8192 bytes in length. Choose the parity of each generated random byte as even, odd, or random. The 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

```
CSNBRNL(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    seed_length,
    seed,
    random_number_length,
    random_number )
```

Parameters

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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**
Random Number Generate Long (CSNBRNGL)

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length, and must be left-justified and padded on the right with space characters. The values are listed in Table 35.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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
Direction: Input    Type: Integer
A pointer to an integer variable containing the number of bytes in the seed variable. The value must be 0.

seed
Direction: Input    Type: String
A pointer to a string variable containing a seed value. A seed can be supplied so that the same value of the random number can be obtained in multiple instances. This can be useful in testing situations. This variable is currently unused.

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
None

Required commands
None

Usage notes
None
Random Number Generate Long (CSNBRNGL)

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 18.

The parameters for CSNBRNGLJ are shown here.

```
public native void CSNBRNGLJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    hikmNativeInteger reserved_seed_length,
    byte[] reserved_seed,
    hikmNativeInteger random_number_length,
    byte[] random_number);
```
Chapter 6. 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 193
- "Encipher (CSNBENC)" on page 198
- "Symmetric Algorithm Decipher (CSNBSAD)" on page 203
- "Symmetric Algorithm Encipher (CSNBSAE)" on page 210

DES modes of operation

To encipher or decipher 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.

The CCA Encipher and Decipher verbs operate in DES CBC (Cipher Block Chaining) mode.

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 using 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.

Processing rules

CCA handles chaining for each 8-byte block of data, from the first block until the last complete 8-byte block of data in each encipher call. There are different types of
processing rules you can choose for cipher block chaining. You choose the type of processing rule that the verb should use for CBC mode:

**ANSI X9.23**
Not necessarily in exact multiples of eight bytes. This processing rule pads the plaintext so the ciphertext produced is in exact multiples of eight bytes.

**Cipher block chaining (CBC)**
In exact multiples of eight bytes.

**Cryptographic Unit Support Program (CUSB)**
CBC mode (cipher block chaining) that is compatible with IBM's CUSB 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.

“Ciphering methods” on page 497 describes the cipher processing rules in detail.

The resulting chaining value, after an encipher call, is known as an *output chaining vector* (OCV). When there are multiple cipher requests, the application can pass the output chaining vector from the previous encipher call as the ICV in the next 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 that an application can use to select record chaining with the CBC processing rule.

---

## AES Modes of Operation

CCA also provides for enciphering or deciphering data the U.S. National Institute of Standards and Technology (NIST) Advanced Encryption Standard (AES) algorithm, with 16-byte, 24-byte or 32-byte keys.

### Processing rules

CCA handles chaining for each block of data. There are different types of processing rules you can choose for cipher block chaining:

**Cipher block chaining (CBC)**
Data must be an exact multiple of eight bytes and output will have the same length.

**Electronic Code Book (ECB)**
The data length must be a multiple of the block size.

**PKCS-PAD**
Specifies that data is padded on the right with between one and 16 bytes of pad characters, making cipher text a multiple of the block size.

The resulting chaining value (except for ECB mode), after a 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 Symmetric Algorithm Encipher call, as the input chaining vector (ICV) in the next 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 that an application can use to select record chaining with the CBC processing rule.
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><strong>Cipher Block Chaining (CBC)</strong></td>
</tr>
<tr>
<td>The ciphertext must be an exact multiple of eight bytes and the plaintext will have the same length.</td>
</tr>
</tbody>
</table>

| **Cryptographic Unit Support Program (CUSB)** |
| CBC mode (cipher block chaining) that is compatible with IBM's CUSB 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. |

| **X9.23** |
| For cipher block chaining. The ciphertext must be an exact multiple of eight bytes, but the plaintext will be one through eight bytes shorter than the ciphertext. The text_length will also be reduced to show the original length of 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 X9.23 padding scheme.

**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 8.

**Format**

```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**

- **return_code**
  - Direction: Output
  - Type: Integer
  - The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

- **reason_code**
  - Direction: Output
  - Type: Integer
  - The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

- **exit_data_length**
  - Direction: Input/Output
  - Type: Integer
  - 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**
  - Direction: Input/Output
  - Type: String
  - The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

- **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 DECRYPTER 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.

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. The 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. See the keywords in Table 36. 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.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Rule (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 (CUSB) cipher block chaining.</td>
</tr>
</tbody>
</table>
Table 36. Keywords for Decipher control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**ICV Selection (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 497](#) 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 verb will fail if the key token contains double or triple-length keys and triple-DES is not enabled.

**Required commands**

This verb requires the Decipher command (offset X'000F') to be enabled in the active role.

**Usage notes**

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 18.

The parameters for CSNBDECJ are shown here.
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,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] chaining_vector,
    byte[] plaintext
);
Encipher (CSNBENC)

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 the Encipher verb should use for the block chaining.

<table>
<thead>
<tr>
<th>Processing Rule</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cipher block chaining (CBC)</td>
<td>In exact multiples of eight bytes.</td>
</tr>
<tr>
<td>Cryptographic Unit Support Program (CUSP)</td>
<td>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.</td>
</tr>
<tr>
<td>Information Protection System (IPS)</td>
<td>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.</td>
</tr>
<tr>
<td>X9.23</td>
<td>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.</td>
</tr>
</tbody>
</table>

For more information about the processing rules, see Table 37 on page 201 and Ciphering methods.

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 schemes. 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 8.

Format

```c
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

**return_code**

Direction: Output  
Type: Integer  

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

**reason_code**

Direction: Output  
Type: Integer  

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

**exit_data_length**

Direction: Input/Output  
Type: Integer
Encipher (CSNBENC)

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 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 ENCIPHER 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. The 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. The 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. See the keywords in Table 37 on page 201. 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.

Table 37. Keywords for Encipher control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Rule (required)</strong></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>Performs cipher block chaining with 1 to eight bytes of padding. This is compatible with the requirements in ANSI standard 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>
<tr>
<td><strong>ICV Selection (optional)</strong></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><strong>Encryption Algorithm (optional)</strong></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 497] describes the cipher processing rules in detail.

**pad_character**

Direction: Input  
Type: Integer  
An integer, 0 to 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.
Encipher (CSNBENC)

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 verb will fail if the key token contains double-length or triple-length keys and triple-DES is not enabled.

Required commands
This verb requires the Encipher command (offset X'000E') to be enabled in the active role.

Usage notes
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 18.

The parameters for CSNBENCJ are shown here.

```
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
);
```
Use the Symmetric Algorithm Decipher verb to decipher data using the AES cipher block chaining mode. CCA supports the following processing rules to decipher data. You choose the type of processing rule that the verb should use for block chaining.

<table>
<thead>
<tr>
<th>Processing Rule</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBC</td>
<td>For cipher block chaining. The ciphertext must be an exact multiple of eight bytes and the plaintext will have the same length.</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>
<tr>
<td>PKCS-PAD</td>
<td>Specifies that the cleartext was padded on the right with one through 16 bytes of pad characters, making the padded text a multiple of the block size, before the data was enciphered.</td>
</tr>
</tbody>
</table>

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 AES key-token.
3. An encrypted key contained in an internal 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 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 64-byte internal AES key-token or the key label of an internal AES key-token.
- A block size of 16 for the cryptographic algorithm.
- For cipher block chaining, either one of these:
  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.
Symmetric Algorithm Decipher (CSNBSAD)

This verb does the following when it deciphers 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 through 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 4.0.0 (the most recent release of CCA on the CEX3C feature), AES keys were only available as DATA keys. For details about CPACF, see “CPACF support” on page 8.

Format

```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 )
```

Parameters

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer
The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

**exit_data_length**

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

- **Direction:** Input/Output
- **Type:** String
- The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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:** Array
- An array of 8-byte keywords providing the processing control information. The keywords must be left-justified and padded on the right with space characters. See the keywords in Table 38.

Table 38. Keywords for Symmetric Algorithm Decipher control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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>
### Table 38. Keywords for Symmetric Algorithm Decipher control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKCS-PAD</td>
<td>Specifies that the cleartext was padded on the right with one through 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 one through 16 bytes less than the ciphertext length.</td>
</tr>
</tbody>
</table>

**Key rule** (one, optional)

<table>
<thead>
<tr>
<th>KEY-CLR</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**ICV Selection** (one, optional)

<table>
<thead>
<tr>
<th>CONTINUE</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL</td>
<td>This specifies taking the initialization vector from the initialization_vector parameter. INITIAL is the default value.</td>
</tr>
</tbody>
</table>

"Ciphering methods" on page 497 describes the cipher processing rules in detail.

**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 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 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. The same initialization vector must have been used to encipher the data.

**chain_data_length**
- Direction: Input/Output
- Type: Integer
- 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 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. The value must not be 0. If PKCS-PAD is specified, the output cleartext_length variable will be from one through 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.
Symmetric Algorithm Decipher (CSNBSAD)

**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 one through 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**
- 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**
None.

**Required commands**
This verb requires the Decipher Data Using AES command (offset X'012B') to be enabled in the active role.

**Usage notes**
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 18.

The parameters for CSNBSADJ are shown here.
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,
    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 cipher block chaining mode. CCA supports the following processing rules to encipher data. You choose the type of processing rule that the verb should use for block chaining.

<table>
<thead>
<tr>
<th>Processing Rule</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBC</td>
<td>For cipher block chaining. The plaintext must be an exact multiple of eight bytes and the ciphertext will have the same length.</td>
</tr>
<tr>
<td>ECB</td>
<td>Specifies enciphering in electronic code book mode. The plaintext length must be a multiple of the block size.</td>
</tr>
<tr>
<td>PKCS-PAD</td>
<td>Specifies that the plaintext was padded on the right with one through 16 bytes of pad characters, making the padded text a multiple of the block size.</td>
</tr>
</tbody>
</table>

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 AES key-token.
3. An encrypted key contained in an internal 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, either in application storage or in 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.

- A key identifier containing a 16-byte, 24-byte, or 32-byte clear key for a key rule of KEY-CLR, or a 64-byte internal AES key-token or the key label of an internal AES key-token for a key rule of KEYIDENT.
- A block size of 16 for the cryptographic algorithm.
- For cipher block chaining, either one of these:
  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 one through 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 4.0.0 (the most recent release of CCA on the CEX3C adapter), AES keys were only available as DATA keys. For details about CPACF, see "CPACF support" on page 8.

**Format**

```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**

**return_code**

- Direction: Output
- Type: Integer

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

**reason_code**

- Direction: Output
- Type: Integer

The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.
Symmetric Algorithm Encipher (CSNBSAE)

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

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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: Array

A pointer to a string variable containing an array of keywords. The keywords are eight bytes in length, and must be left-justified and padded on the right with space characters. See the keywords in Table 39.

### Table 39. Keywords for Symmetric Algorithm Encipher control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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 <code>chaining_vector</code> 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 one through 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.</td>
</tr>
<tr>
<td><strong>Key rule</strong> (one, optional)</td>
<td></td>
</tr>
<tr>
<td>KEY-CLR</td>
<td>Specifies that the <code>key_identifier</code> 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 <code>key_identifier</code> parameter points to an internal AES key-token or the label of an internal key-token in AES key-storage.</td>
</tr>
</tbody>
</table>
Table 39. Keywords for Symmetric Algorithm Encipher control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICV Selection</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>CONTINUE</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 497 describes the cipher processing rules in detail.

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 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 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 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
Symmetric Algorithm Encipher (CSNBSAE)

- **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. The same initialization vector must be used when deciphering the data.

- **chain_data_length**
  - **Direction:** Input/Output  **Type:** Integer
  - 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 one through 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 one through 16 bytes of data more than the cleartext input buffer contains.

optional_data_length
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
None.

Required commands
This verb requires the Encipher Data Using AES command (offset X'012A') to be enabled in the active role.

Usage notes
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 18.

The parameters for CSNBSAEJ are shown here.

```
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,
    byte[] cipher_text,
    hikmNativeInteger optional_data_length,
    byte[] optional_data);
```
Symmetric Algorithm Encipher (CSNBSAE)
Chapter 7. Verifying data integrity and authenticating messages

CCA provides the following methods to verify the integrity of transmitted messages and stored data:

- 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 10, “Using digital signatures,” on page 343) 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 the following topics include:

- "MAC Generate (CSNBMGN)" on page 219
- "MAC Verify (CSNBMVR)" on page 224
- "One-Way Hash (CSNBOWH)" on page 229

**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 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 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 Digital Signature verbs (see Chapter 10, “Using digital signatures,” on page 343). 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. The 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 495.

---

**Data integrity and authentication verbs**

**MAC Generate (CSNBMGN)**

Use the MAC Generate verb to generate a 4-byte, 6-byte, or 8-byte message authentication code (MAC) for an application-supplied text string. You can specify that the verb uses either the ANSI X9.9-1 procedure or the ANSI X9.19 optional double key MAC procedure to compute the MAC. For the ANSI X9.9-1 procedure you identify either a MAC generate key or a DATA key and the message text. For the ANSI X9.19 optional double key MAC procedure, you identify a double-length MAC key and the message text.

The MAC Generate verb also supports the padding rules specified in the EMV Specification.

**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 8.

**Format**

```
CSNBMGN(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   key_identifier,
   text_length,
   text,
   rule_array_count,
   rule_array,
   chaining_vector,
   mac )
```

**Parameters**

**return_code**

- Direction: Output
- Type: Integer

The return code specifies the general result of the verb. [Appendix A, “Return codes and reason codes”](#) lists the return codes.

**reason_code**

- Direction: Output
- Type: Integer
The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

exit_data_length
Direction: Input/Output  Type: Integer
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
Direction: Input/Output  Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

direction: Output  Type: String
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. The value can 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 shown in Table 40 on page 221. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-justified 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. They are described in Table 40 on page 221
Table 40. Keywords for MAC Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAC Process Rules (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 one through eight 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 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 X9.9-1.</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 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>
<tr>
<td><strong>Segmenting Control (optional)</strong></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>
<tr>
<td><strong>MAC Length and Presentation (optional)</strong></td>
<td></td>
</tr>
<tr>
<td>HEX-8</td>
<td>Generates a 4-byte MAC value and presents it as 8 hexadecimal characters.</td>
</tr>
<tr>
<td>HEX-9</td>
<td>Generates a 4-byte MAC value and presents it as 2 groups of 4 hexadecimal characters with a space between the groups.</td>
</tr>
<tr>
<td>MACLEN4</td>
<td>Generates a 4-byte MAC value. This is the default value.</td>
</tr>
<tr>
<td>MACLEN6</td>
<td>Generates a 6-byte MAC value.</td>
</tr>
<tr>
<td>MACLEN8</td>
<td>Generates an 8-byte MAC value.</td>
</tr>
</tbody>
</table>

**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**
MAC Generate (CSNBMGN)

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
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 8) 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 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 CEX3C 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 CEX3C.

Required commands
This verb requires the Generate MAC command (offset X'0010') to be enabled in the active role.

Usage notes
None

Related information
The MAC Verify verb is described in MAC Verify (CSNBMVR) on page 224.

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 18.

The parameters for CSNBMGNJ are shown here.
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);
Use the MAC Verify verb to verify a 4-byte, 6-byte, or 8-byte message authentication code (MAC) for an application-supplied text string. You can specify that the verb uses either the ANSI X9.9-1 procedure or the ANSI X9.19 optional double key MAC procedure to compute the MAC. For the ANSI X9.9-1 procedure you identify either a MAC verify key, a MAC generation key, or a DATA key, and the message text. For the ANSI X9.19 optional double key MAC procedure, you identify either a double-length MAC verify key or a double-length MAC generation key and the message text. The cryptographic feature compares the generated MAC with the one sent with the message. A return code indicates whether the MACs are the same. If the MACs are the same, the receiver knows the message was not altered. The generated MAC never appears in storage and is not revealed outside the cryptographic feature.

The MAC Verify verb also supports the padding rules specified in the EMV Specification.

Host CPU acceleration: CPACF
For details about CPACF, see "CPACF support" on page 8.

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).

Format

```
CSNBMVR(
  return_code,
  reason_code,
  exit_data_length,
  exit_data,
  key_identifier,
  text_length,
  text,
  rule_array_count,
  rule_array,
  chaining_vector,
  mac
)
```

Parameters

return_code
Direction: Output Type: Integer
The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

reason_code
Direction: Output Type: Integer
The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

exit_data_length
Direction: Input/Output Type: Integer
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**

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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. The value can 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 shown in Table 41. The keywords must be in 24 bytes of contiguous storage with each of the keywords left-justified 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. For a list of values, see Table 41.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAC Process Rules (optional)</strong></td>
<td></td>
</tr>
</tbody>
</table>

---

Chapter 7. Verifying data integrity and authenticating messages 225
Table 41. Keywords for MAC Verify control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMVMAC</td>
<td>EMV padding rule with a single-length MAC key. The key_identifier parameter must identify a single-length MAC, MACVER, or DATA key. The text is always padded with one through eight 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>EMVMACD</td>
<td>EMV padding rule with a double-length MAC key. The key_identifier parameter must identify a double-length MAC or MACVER key. The padding rules are the same as for 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>X9.9-1</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)**

<table>
<thead>
<tr>
<th>Segmenting Control</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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; the application program does not employ segmenting. This is the default value.</td>
</tr>
</tbody>
</table>

**MAC Length and Presentation (optional)**

<table>
<thead>
<tr>
<th>MAC Length and Presentation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEX-8</td>
<td>Verifies a 4-byte MAC value represented as 8 hexadecimal characters.</td>
</tr>
<tr>
<td>HEX-9</td>
<td>Verifies a 4-byte MAC value represented as 2 groups of 4 hexadecimal characters with a space character between the groups.</td>
</tr>
<tr>
<td>MACLEN4</td>
<td>Verifies a 4-byte MAC value. This is the default value.</td>
</tr>
<tr>
<td>MACLEN6</td>
<td>Verifies a 6-byte MAC value.</td>
</tr>
<tr>
<td>MACLEN8</td>
<td>Verifies an 8-byte MAC value.</td>
</tr>
</tbody>
</table>

**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.
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-justified 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
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 8) 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 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 CEX3C 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 CEX3C.

Required commands
This verb requires the Verify MAC command (offset X'0011') to be enabled in the active role.

Usage notes
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
The MAC Generate verb is described in "MAC Generate (CSNBMGN)” on page 219.

J NI 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 18.

The parameters for CSNBMVRJ are shown here.
MAC Verify (CSNBMVR)

<table>
<thead>
<tr>
<th>FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>public native void CSNBMVRJ(</td>
</tr>
<tr>
<td>hikmNativeInteger return_code,</td>
</tr>
<tr>
<td>hikmNativeInteger reason_code,</td>
</tr>
<tr>
<td>hikmNativeInteger exit_data_length,</td>
</tr>
<tr>
<td>byte[] exit_data,</td>
</tr>
<tr>
<td>byte[] key_identifier,</td>
</tr>
<tr>
<td>hikmNativeInteger text_length,</td>
</tr>
<tr>
<td>byte[] text,</td>
</tr>
<tr>
<td>hikmNativeInteger rule_array_count,</td>
</tr>
<tr>
<td>byte[] rule_array,</td>
</tr>
<tr>
<td>byte[] chaining_vector,</td>
</tr>
<tr>
<td>byte[] MAC);</td>
</tr>
</tbody>
</table>
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, SHA-512. For details about CPACF, see “CPACF support” on page 8.

Format

```
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

**return_code**

- Direction: Output
- Type: Integer

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

**reason_code**

- Direction: Output
- Type: Integer

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” lists the reason codes.

**exit_data_length**

- Direction: Input/Output
- Type: Integer

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**

- Direction: Input/Output
- Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 or 2.

**rule_array**
Keywords that provide control information to the verb are listed in Table 42. The optional chaining flag keyword indicates whether calls to this verb are chained together logically to overcome buffer size limitations. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

### Table 42. Keywords for One-Way Hash control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hash Method (required)</strong></td>
<td></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>
<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>
<tr>
<td><strong>Chaining Flag (optional)</strong></td>
<td></td>
</tr>
<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.
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.

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-justified. 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
None

Required commands
None

Usage notes
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 18.

The parameters for CSNBOWHJ are shown here.

```java
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);
```
One-Way Hash (CSNBOWH)
Chapter 8. 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 the following topics include:

- "AES Key Record Create (CSNBAKRC)" on page 239
- "AES Key Record Delete (CSNBAKRD)" on page 242
- "AES Key Record List (CSNBAKRL)" on page 244
- "AES Key Record Read (CSNBAKRR)" on page 247
- "AES Key Record Write (CSNBAKRW)" on page 249
- "DES Key Record Create (CSNBKRC)" on page 252
- "DES Key Record Delete (CSNBKRD)" on page 254
- "DES Key Record List (CSNBKRL)" on page 256
- "DES Key Record Read (CSNBKRR)" on page 259
- "DES Key Record Write (CSNBKRW)" on page 261
- "PKA Key Record Create (CSNDKRC)" on page 263
- "PKA Key Record Delete (CSNDKRD)" on page 265
- "PKA Key Record List (CSNDKRL)" on page 267
- "PKA Key Record Read (CSNDKRR)" on page 270
- "PKA Key Record Write (CSNDKRW)" on page 272
- "Retained Key Delete (CSNRKRD)" on page 275
- "Retained Key List (CSNRKRL)" on page 278

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 549.

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' through 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' through 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 through 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 through 7 name tokens, each separated by a period (.). The key label must not end with a period.
- A name token consists of 1 through 8 characters in the character set A through Z, 0 through 9, and three additional characters relating to different character symbols in the various national language character sets as listed in Table 43.

<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.

Notes:
1. Some CCA implementations accept the characters a through z and fold these to their uppercase equivalents, A through 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 44.

<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>11111111.2.3.4.5555</td>
<td>First character cannot be numeric</td>
</tr>
<tr>
<td>A1111111.2.3.4.55555.6.7.8</td>
<td>Number of name tokens exceeds 7</td>
</tr>
<tr>
<td>BANKSYS.XXXXX.<em>43</em>.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 three master key registers for each of three types of master keys (Symmetric DES, AES, and Asymmetric RSA). In other words, there are a total of nine master key registers. The master key registers are one of three types:

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 98.

FULL
The LAST key part has been loaded, but the SET command has not yet been called. See "Master Key Process (CSNBMKP)" on page 98.

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 is invoked with "Master Key Process (CSNBMKP)" on page 98.
The SET command performs these operations:
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 (RTN MK-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 RTN MK rule_array keyword of "Key Token Change (CSNBKTC)" on page 161 (for AES or DES) or "PKA Key Token Change (CSNDKTC)" on page 377 (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" on page 236. 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" on page 236.
- 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 161 or "PKA Key Token Change (CSNDKTC)" on page 377.
  - 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 CEX3C has its master key changed.

These two properties force the user to use the same process to change the master key for all CEX3Cs after the first CEX3C. 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 CEX3C to have a conflict with the key storage header. Therefore, using the same process to change the master keys in all the CEX3C 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 CEX3C adapters in a group.
- An operator can directly change the master keys for a domain on a CEX3C 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 CEX3C by invoking "Cryptographic Resource Allocate (CSUACRA)" on page 89.
  2. Change the master key.
  3. Deallocate each adapter in the group before exiting, by invoking "Cryptographic Resource Deallocate (CSUACRD)" on page 92.

- 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 551.

### 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 RTNMK keyword for "Key Token Change (CSNBKTC)" on page 161 and "PKA Key Token Change (CSNDKTC)" on page 377. 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 237.

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 an System z platform and likely to collaborate with z/OS software. Therefore, the RTNMK keyword is provided for "Key Token Change (CSNBKTC)" on page 161 and "PKA Key Token Change (CSNDKTC)" on page 377 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 161 and "PKA Key Token Change (CSNDKTC)" on page 377 for re-encipherment, and stored outside CCA host key storage.
When re-encipherment is complete and the "Master Key Process (CSNBMKP)" on page 98 SET command has been issued, the re-enciphered key tokens can be reintroduced to CCA host key storage if desired, using the standard mechanisms.

Key storage management verbs

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
- PKA Symmetric Key Generate
- PKA Symmetric Key Import

Notes:
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

```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

return_code

Direction: Output  Type: Integer

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

reason_code

Direction: Output  Type: Integer

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” lists the reason codes.
AES Key Record Create (CSNBAKRC)

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

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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: Array

A pointer to a string variable containing an array of keywords. This verb does not use keywords.

**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 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**
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**
None

**Related information**
See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.
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 18.

The parameters for CSNBAKRCJ are shown here.

```
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 following tasks 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**

```
CSNBAKRD (  
    return_code,  
    reason_code,  
    exit_data_length,  
    exit_data,  
    rule_array_count,  
    rule_array,  
    key_label )
```

**Parameters**

**return_code**
- Direction: Output  
- Type: Integer

The return code specifies the general result of the verb. [Appendix A, “Return codes and reason codes”](#) lists the return codes.

**reason_code**
- Direction: Output  
- Type: Integer

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”](#) lists the reason codes.

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

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**
- Direction: Input  
- Type: Integer
A pointer to an integer variable containing the number of elements in the
`rule_array` variable. This number 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 45.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOKEN-DL</td>
<td>Deletes a key token from a key record in AES key storage. This is the default.</td>
</tr>
<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 record defined by the `key_label` must be unique.

**Required commands**
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**
None

**Related information**
See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 235.

**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 18.

The parameters for CSNBAKRDJ are shown here.

```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);
```
AES Key Record List (CSNBAKRL)

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.

Notes:
1. To list all the labels in key storage, specify the key_label parameter with *,&rbl;&rbl;*,&rbl;*,&rbl;*,&rbl;*, 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.

The 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 to \( n \) detail records, where \( n \) is the number of key records with matching key-labels.

The file is kept in the /opt/IBM/CEX3C/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

```snippet
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

return_code
- Direction: Output
- Type: Integer

The `return_code` parameter is a pointer to an integer value that expresses the general results of processing. See "Return code and reason code overview" on page 16 for more information about return co.

reason_code
- Direction: Output
- Type: Integer
The `reason_code` parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See "Return code and reason code overview" on page 16 for more information about reason codes.

**exit_data_length**

Direction: Input/Output  Type: Integer

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**

Direction: Input/Output  Type: String

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 0.

**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: 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.
AES Key Record List (CSNBAKRL)

The verb returns the file name as a fully qualified file specification (for example, /opt/IBM/CEX3C/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
None

Required commands
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

Usage notes
None

Related information
See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 235.

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 18.

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

```
CSNBAKRR (  
   return_code,  
   reason_code,  
   exit_data_length,  
   exit_data,  
   rule_array_count,  
   rule_array,  
   key_label,  
   key_token_length,  
   key_token )
```

Parameters

return_code

Direction: Output Type: Integer

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

reason_code

Direction: Output Type: Integer

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 indicating specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

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

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

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
AES Key Record Read (CSNBAKRR)

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 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.

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

None.

Required commands

This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

Usage notes

None

Related information

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

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 18.

The parameters for CSNBAKRRJ are shown here.

```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. The 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 239 to create a key record in the key storage file.

Format

```c
CSNBAKRW ( return_code, reason_code, exit_data_length, exit_data, rule_array_count, rule_array, key_label, key_token_length, key_token )
```

Parameters

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

Direction: Input Type: Integer

APPENDIX A: DIRECTORY OF RETURN AND REASON CODES

APPENDIX A: DIRECTORY OF RETURN AND REASON CODES
AES Key Record Write (CSNBAKRW)

A pointer to an integer variable containing the number of elements in the
*rule_array* variable. The 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 46**.

**Table 46. Keywords for AES Key Record Write control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing option</td>
<td><em>(one, optional)</em></td>
</tr>
<tr>
<td>CHECK</td>
<td>Specifies that the record is written only if a record of the same label in AES</td>
</tr>
<tr>
<td>key-storage contains a null key-token. This is the default.</td>
<td></td>
</tr>
<tr>
<td>OVERLAY</td>
<td>Specifies that the record is overwritten regardless of the current content of</td>
</tr>
<tr>
<td>the record in AES key-storage.</td>
<td></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 record defined by the *key_label* parameter must be unique and must already
exist in the key storage file.

**Required commands**

This verb requires the Compute Verification Pattern command (offset X'001D') to be
enabled in the active role.

**Usage notes**

None

**Related information**

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 235.
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 18.

The parameters for CSNBAKRWJ are shown here.

```
FORMAT
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.

DES key records are stored in the external key-storage file defined by the CSUDESDDS environment variable.

Format

```c
CSNBKRC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_label
)
```

Parameters

- **return_code**
  - Direction: Output
  - Type: Integer
  - The return code specifies the general result of the verb. [Appendix A, “Return codes and reason codes”](#) lists the return codes.

- **reason_code**
  - Direction: Output
  - Type: Integer
  - 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”](#) lists the reason codes.

- **exit_data_length**
  - Direction: Input/Output
  - Type: Integer
  - 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**
  - Direction: Input/Output
  - Type: String
  - The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

- **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 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**

This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**

None

**Related information**

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

**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 18.

The parameters for CSNBKRCJ are shown here.

```
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 CSUDESDDS environment variable.

Format

```c
CSNBKRD(
   return_code,
   reason_code,
   exit_data_length,
   exit_data,
   rule_array_count,
   rule_array,
   key_label
)
```

Parameters

**return_code**

- **Direction:** Output
- **Type:** Integer

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

**reason_code**

- **Direction:** Output
- **Type:** Integer

  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” lists the reason codes.

**exit_data_length**

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

  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**

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

  The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

- **Direction:** Input
- **Type:** Integer

  A pointer to an integer variable containing the number of elements in the `rule_array` variable. This number must always be 1.

**rule_array**

- **Direction:** Input
- **Type:** String
The 8-byte keyword that defines the action to be performed. The keywords are listed in Table 47.

Table 47. Keywords for DES Key Record Delete control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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 record defined by the key_label must be unique.

**Required commands**
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**
None

**Related information**
See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 235.

**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 18.

The parameters for CSNBKRDJ are shown here.

```java
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 );
```
DES Key Record List (CSNBKRL)

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 

```
*.*.*.*.*.*.
```

The 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 to \( n \) detail records, where \( n \) is the number of key records with matching key-labels. The file is kept in the \!/opt/IBM/CEX3C/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.

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Format

```
CSNBKRL(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    key_label,
    dataset_name_length,
    dataset_name,
    security_server_name )
```

Parameters

**return_code**

Direction: Output Type: Integer

The **return_code** parameter is a pointer to an integer value that expresses the general results of processing. See “Return code and reason code overview” on page 16 for more information about return codes.

**reason_code**

Direction: Output Type: Integer

The **reason_code** parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See “Return code and reason code overview” on page 16 for more information about reason codes.

**exit_data_length**
Direction: Input/Output  
Type: Integer
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
Direction: Input/Output  
Type: String
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 0.

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 wildcard (*) to identify multiple records in key storage.

dataset_name_length
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
None

Required commands
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.
DES Key Record List (CSNBKRL)

**Usage notes**
None

**Related information**
See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

**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 18.

The parameters for CSNBKRLJ are shown here.

```
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 CSUDESDS environment variable.

Format

```
CSNBKRR(  
   return_code,  
   reason_code,  
   exit_data_length,  
   exit_data,  
   key_label,  
   key_token  
)
```

Parameters

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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 indicating specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 record defined by the `key_label` parameter must be unique and must already exist in the key storage file.

**Required commands**
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**
None

**Related information**
See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

**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 18.

The parameters for CSNBKRRJ are shown here.

```java
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.

DES key records are stored in the external key-storage file defined by the CSUDESDS environment variable.

Note: Before you use this verb, use the DES Key Record Create verb (see “DES Key Record Create (CSNBKRC)” on page 252) to create a key record in the key storage file.

Format

```
CSNBKRW(  
return_code,  
reason_code,  
exit_data_length,  
exit_data,  
key_token,  
key_label )
```

Parameters

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 record defined by the key_label parameter must be unique and must already exist in the key storage file.

Required commands
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

Usage notes
None

Related information
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 235.

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 18.

The parameters for CSNBKRWJ are shown here.

```java
public native void CSNBKRWJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger 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

```
CSNDKRC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label,
    token_length,
    token )
```

Parameters

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no
exit data is allowed in this parameter.

**rule_array_count**

Direction: Input Type: Integer

A pointer to an integer variable containing the number of elements in the
rule_array variable. This parameter is ignored.

**rule_array**

Direction: Input Type: String

This parameter is ignored.
PKA Key Record Create (CSNDKRC)

**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
- 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**
None

**Required commands**
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**
None

**Related information**
See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

**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 18.

The parameters for CSNDKRCJ are shown here.

```java
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 );
```
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

```
CSNDKRD(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label
)
```

Parameters

**return_code**

Direction: Output  Type: Integer

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

**reason_code**

Direction: Output  Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output  Type: Integer

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**

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

Direction: Input  Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This parameter is ignored except that its value must be 0 or 1.

**rule_array**

Direction: Input  Type: String
PKA Key Record Delete (CSNDKRD)

Keywords that provide control information to the verb. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage.

Table 48. Keywords for PKA Key Record Delete control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deletion Mode (optional)</td>
<td>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
None

Required commands
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

Usage notes
None

Related information
See Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z on page 235.

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 18.

The parameters for CSNDKRDJ are shown here.

```
FORMAT
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 (`***.****`).

The verb creates the list file and returns the name of the file and the length of the file name to the calling application. The 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 to \( n \) detail records, where \( n \) is the number of key records with matching key labels. The file is kept in the `/opt/IBM/CEX3C/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

```c
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

**return_code**

- Direction: Output
- Type: Integer

The `return_code` parameter is a pointer to an integer value that expresses the general results of processing. See "Return code and reason code overview" on page 16 for more information about return codes.

**reason_code**

- Direction: Output
- Type: Integer
PKA Key Record List (CSNDKRL)

The reason_code parameter is a pointer to an integer value that expresses the specific results of processing. Each possible result is assigned a unique reason code value. See "Return code and reason code overview" on page 16 for more information about reason codes.

exit_data_length

Direction: Input/Output Type: Integer

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

Direction: Input/Output Type: String

The exit_data parameter is a pointer to a variable-length string that contains installation-exit-dependent data that is exchanged with a preprocessing user exit or a post processing user exit.

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

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 for this verb.

rule_array

Direction: Input Type: Array

The rule_array parameter is a pointer to a string variable containing an array of keywords. This verb does not use keywords.

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**

None

**Required commands**

This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**

None

**Related information**

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

**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 18.

The parameters for CSNDKRLJ are shown here.

```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)

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

```plaintext
CSNDKRR(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label,
    token_length,
    token)
```

Parameters

return_code

Direction: Output  Type: Integer

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

reason_code

Direction: Output  Type: Integer

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” lists the reason codes.

exit_data_length

Direction: Input/Output  Type: Integer

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

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

rule_array_count

Direction: Input  Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. This parameter is ignored.

rule_array

Direction: Input  Type: String

This parameter is ignored.
PKA Key Record Read (CSNDKRR)

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
Area into which the returned record will be written. The area should be at least as long as the record.

Restrictions
None

Required commands
This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

Usage notes
None

Related information
See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

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 18.

The parameters for CSNDKRRJ are shown here.

```
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)

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

```c
CSNDKRW(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    label,
    token_length,
    token )
```

Parameters

return_code

Direction: Output Type: Integer

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

reason_code

Direction: Output Type: Integer

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” lists the reason codes.

exit_data_length

Direction: Input/Output Type: Integer

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

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

rule_array_count

Direction: Input Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. Its value must be 0 or 1.

rule_array

Direction: Input Type: String
Keywords that provide control information to the verb. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. Table 49 lists the keywords.

Table 49. Keywords for PKA Key Record Write control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Write Mode (optional)</strong> specifies the circumstances under which the record is to be written.</td>
<td></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.

**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**

None

**Required commands**

This verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

**Usage notes**

None

**Related information**

See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z" on page 235.

**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 18.

The parameters for CSNDKRWJ are shown here.
PKA Key Record Write (CSNDKRW)

```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 and PKA Public Key Register. 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” on page 276.

Format

```
CSNDRKD(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_label )
```

Parameters

return_code
Direction: Output Type: Integer
The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

reason_code
Direction: Output Type: Integer
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” lists the reason codes.

exit_data_length
Direction: Input/Output Type: Integer
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
Direction: Input/Output Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

rule_array_count
Direction: Input Type: Integer
Retained Key Delete (CSNDRKD)

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
This parameter should be a null address pointer.

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 CEX3C Cryptographic adapter, by definition.

- 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 363 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 278 lists the retained keys inside an
adapter.
- "Retained Key Delete (CSNDRKD)" on page 275 deletes a retained key from
adapter internal storage.

Restrictions
None
**Required commands**
This verb requires the Delete Retained Key command (offset X'0203'') to be enabled in the active role.

**Usage notes**
None

**Related information**
See “Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z” on page 235.

**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 18.

The parameters for CSNDRKDJ are shown here.

```java
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
);```

Chapter 8. Key storage mechanisms 277
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 and the PKA Public Key Register verbs. 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 276.

Format

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

return_code
Direction: Output  Type: Integer
The return code specifies the general result of the verb. Appendix A, "Return codes and reason codes" lists the return codes.

reason_code
Direction: Output  Type: Integer
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" lists the reason codes.

exit_data_length
Direction: Input/Output  Type: Integer
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
  Direction: Input/Output  Type: String
  The data that is passed to an installation exit. Exits are not supported and no
  exit data is allowed in this parameter.
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: Array
  This parameter should be a null address pointer.
key_label_mask
  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
  None

Required commands
  This verb requires the List Retained Key command (offset X'0230'') to be enabled in
  the active role.

Usage notes
  None

Related information
  See "Key storage with Linux for IBM System z, in contrast to z/OS for IBM System
  z" on page 235.
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 18.

The parameters for CSNDRKLJ are shown here.

```
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 9. 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 the following topics:

- "Clear PIN Encrypt (CSNBCPE)" on page 287
- "Clear PIN Generate (CSNBPGN)" on page 291
- "Clear PIN Generate Alternate (CSNBCPA)" on page 295
- "CVV Generate (CSNBCSG)" on page 300
- "CVV Verify (CSNBCSV)" on page 304
- "Encrypted PIN Generate (CSNBEPG)" on page 308
- "Encrypted PIN Translate (CSNBPTR)" on page 313
- "Encrypted PIN Verify (CSNBVPV)" on page 319
- "PIN Change/Unblock (CSNBPUC)" on page 324
- "Secure Messaging for Keys (CSNBSKY)" on page 330
- "Secure Messaging for PINs (CSNBSPN)" on page 334
- "Transaction Validation (CSNBTRV)" on page 338

How personal identification numbers (PINs) are used

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. CCA allows your applications to generate PINs, to verify supplied PINs, and to translate PINs from one format or encryption key to another.

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 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.

3. The VISA CVV and the MasterCard CVC refer to the same value. CVV is used here to mean both CVV and CVC.
Translation 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.

PIN verbs

You use the PIN verbs to generate, verify, and translate PINs. This section discusses 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.

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 verb generates a verification PIN. The 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 discussed in detail in Appendix D, “PIN formats and algorithms,” on page 477.

**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 discussed in detail in Appendix D, “PIN formats and algorithms,” on page 477.

**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 can not be mixed.

**Encrypted PIN translate**

The UKPTIPIN, IPKTOPIN, and UKPTBOTH keywords will cause the verb to generate single-length keys. 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 generate 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.

The PIN profile
The PIN profile consists of the following:
- PIN block format (see "PIN block format")
- Format control (see "Format control" on page 286)
- Pad digit (see "Pad digit" on page 286)
- Current Key Serial Number (for UKPT and DUKPT – see "Current key serial
  number" on page 287)

Table 50 shows the format of a PIN profile.
Table 50. 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-justified and padded with blanks. Refer to Table 51 for a list of valid values.

Table 51. 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

The PIN extraction method keywords specify a PIN extraction method for a PIN block format. The first PIN extraction method keyword listed for a PIN block format is the default. Refer to Table 52 for a list of valid values.

The PINLENnn keywords are disabled for the Clear PIN Generate Alternate (CSNBCPA), Encrypted PIN Translate (CSNBPTR), and Encrypted PIN Verify (CSNBPV) verbs by default. If these keywords are used, return code 8 with reason code 33 is returned. To enable them, the PTR Enhanced PIN Security Mode access control point (bit X’0313’) must be enabled using a TKE.

Table 52. 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>
<tr>
<td>3621</td>
<td>PADDIGIT, HEخد, PINLEN04 to 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, HEخد, PINLEN04 to 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>
Format control

This keyword specifies whether there is any control on the user-supplied PIN format. The 8-byte value must be left-justified 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 53 shows the format of a pad digit. The PIN profile pad digit must be specified in upper case.

Table 53. Format of a pad digit

<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 through 9, letters A through F, or a blank.</td>
</tr>
</tbody>
</table>

Each PIN format supports only a pad digit in a certain range. Table 54 lists the valid pad digits for each PIN block format.

Table 54. Pad digits for PIN block formats

<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>
<tr>
<td>VISA-2</td>
<td>0 through 9</td>
<td>Pad digit is not used</td>
</tr>
<tr>
<td>VISA-3</td>
<td>0 through 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 through F</td>
<td>0 through F</td>
</tr>
<tr>
<td>3624</td>
<td>0 through F</td>
<td>0 through F</td>
</tr>
<tr>
<td>4704-EPP</td>
<td>F</td>
<td>Pad digit is not used</td>
</tr>
</tbody>
</table>

Recommendations for the pad digit

IBM recommends you use a non-decimal pad digit in the range of A through 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 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 55 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 55. Format of the Current Key Serial Number Field

<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 justified and padded with 4 blanks.</td>
</tr>
</tbody>
</table>

Financial services verbs

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
Clear PIN Encrypt (CSNBCPE)

Format

```
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

**return_code**

Direction: Output Type: Integer

The return code specifies the general result of the verb. Appendix A, "Return codes and reason codes" lists the return codes.

**reason_code**

Direction: Output Type: Integer

The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that indicate specific processing problems. Appendix A, "Return codes and reason codes" lists the reason codes.

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 1.

**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, 1, and 2.

**rule_array**
Clear PIN Encrypt (CSNBCPE)

Direction: Input  Type: String

Keywords that provide control information to the verb. The keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The rule_array keywords are shown in Table 56.

Table 56. Process rules for the Clear PIN Encryption verb

<table>
<thead>
<tr>
<th>Process Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 based on the value in the clear_PIN variable. Set the value of the clear PIN to zero and use as many digits as the desired random PIN; pad the 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-justified 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 284 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.

sequence_number

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 format control specified in the PIN profile must be NONE.
Clear PIN Encrypt (CSNBCPE)

**Required commands**
This verb requires the Format and Encrypt PIN 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 Mode command (offset X'0313') in the active role.

**Usage notes**
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 18.

The parameters for CSNBCPEJ are shown here.

```java
public native void CSNBCPEJ(
    hikmNativeInteger return_code,
    hikmNativeInteger reason_code,
    hikmNativeInteger exit_data_length,
    byte[] exit_data,
    byte[] PIN_encrypting_key_identifier,
    hikmNativeInteger rule_array_count,
    byte[] rule_array,
    byte[] clear_PIN,
    byte[] 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

```c
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

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**PIN_generating_key_identifier**

Direction: Input/Output  
Type: String
Clear PIN Generate (CSNBPGN)

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. The value must be 1.

**rule_array**

Direction: Input  
Type: String

The process rule provides control information to the verb. Specify one of the values in Table 57. The keyword is left-justified in an 8-byte field and padded on the right with blanks.

*Table 57. Process rules for the Clear PIN Generate verb*

<table>
<thead>
<tr>
<th>Process Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 through 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 through 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 58 on page 293 describes the array elements.
Table 58. 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 to 12 digits of 0 through 9. Left-justified and padded with spaces. For IBM-PINO, 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 through 9.</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. 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. One to sixteen characters of hexadecimal account data left-justified and padded on the right with blanks.</td>
</tr>
</tbody>
</table>

Table 59 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 59. 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>3</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</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-justified, and padded on the right with blanks.

Restrictions

None

Required commands

This verb requires the Generate Clear 3624 PIN command (offset X'00A0') to be enabled in the active role.

Usage notes

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

The algorithms are discussed in detail in Appendix D, “PIN formats and algorithms,” on page 477.
Clear PIN Generate (CSNBPGN)

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 18.

The parameters for CSNBPGNJ are shown here.

```
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

```
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

**return_code**
- Direction: Output
- Type: Integer
- The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

**reason_code**
- Direction: Output
- Type: Integer
- The reason code specifies the result of the verb that is returned to the application program. Each return code has different reason codes that are assigned to it that indicate specific processing problems. Appendix A, “Return codes and reason codes” lists the reason codes.

**exit_data_length**
- Direction: Input/Output
- Type: Integer
- 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**
- Direction: Input/Output
- Type: String
- The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**PIN_encryption_key_identifier**
- Direction: Input/Output
- Type: String
Clear PIN Generate Alternate (CSNBCPA)

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 284 for additional information.

**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. The value can be 1 or 2. If the default extraction method for a PIN block format is desired, you can code 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-justified, and padded with blanks. The `rule_array` points to an array of one or two 8-byte elements, as described in Table 60.

<table>
<thead>
<tr>
<th><code>rule_array</code> Element</th>
<th>Function of <code>rule_array</code> keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PIN calculation method</td>
</tr>
<tr>
<td>2</td>
<td>PIN extraction method</td>
</tr>
</tbody>
</table>
The first element in the rule_array must specify one of the keywords that indicate the PIN calculation method as shown in Table 61.

Table 61. Keywords for Clear PIN Generate Alternate (first rule_array element)

<table>
<thead>
<tr>
<th>PIN Calculation Method Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>

If the second element in the rule_array is provided, one of the PIN extraction method keywords shown in Table 52 on page 285 can be specified for the given PIN block format. See “PIN block format and PIN extraction method keywords” on page 284 for additional information. If the default extraction method for a PIN block format is desired, you can code the rule_array_count value as 1.

The PIN extraction methods operate as follows:

**PINBLOCK**
- Specifies that the verb use one of the following:
  - 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 to 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 Mode access control point (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 through 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
Clear PIN Generate Alternate (CSNBCPA)

Three 16-byte elements. Table 62 describes the format when IBM-PINO is specified. Table 63 describes the format when VISA-PVV is specified.

Table 62. 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 to 9) that are used to convert hexadecimal digits (X'0' to X'F') of the enciphered validation data to the decimal digits (X'0' to X'9').</td>
</tr>
<tr>
<td>validation_data</td>
<td>This element contains 1 to 16 characters of account data. The data must be left justified 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 63. 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-justified and padded with blanks.

Restrictions

None

Required commands

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>Generate Clear 3624 PIN Offset</td>
</tr>
<tr>
<td>VISA-PVV</td>
<td>X'00BB'</td>
<td>Generate Clear Visa PVV Alternate</td>
</tr>
</tbody>
</table>

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 Mode command (offset X'0313') in the active role.

Usage notes

None

JNI version

This verb has a Java Native Interface (JNI) version, which is named CSNCBPAJ. See [Building Java applications to use with the CCA JNI](#) on page 18.
The parameters for CSNBCPAJ are shown here.

```
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. The 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. 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.

Format

```c
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

* **return_code**
  
  Direction: Output  
  Type: Integer  
  
  The return code specifies the general result of the verb. Section Appendix A, "Return codes and reason codes" lists the return codes.

* **reason_code**
  
  Direction: Output  
  Type: Integer  
  
  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. Section Appendix A, "Return codes and reason codes" lists the reason codes.

* **exit_data_length**
  
  Direction: Input/Output  
  Type: Integer  
  
  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**
  
  Direction: Input/Output  
  Type: String  
  
  The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

* **rule_array_count**
  
  Direction: Input  
  Type: Integer
A pointer to an integer variable containing the number of elements in the `rule_array` variable. The parameter `rule_array_count` must be 0, 1, or 2.

### `rule_array`

**Direction:** Input  
**Type:** String

Keywords that provide control information to the verb. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. [Table 64](#) describes these keywords.

#### Table 64. Keywords for CVV Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAN data length</strong> (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>
</tbody>
</table>

| **CVV length** (optional) |                                                                 |
| CVV-1 | Specifies that the CVV is to be computed as one byte, followed by four blanks. **CVV-1 is the default value.** |
| CVV-2 | Specifies that the CVV is to be computed as two bytes, followed by three blanks. |
| CVV-3 | Specifies that the CVV is to be computed as three bytes, followed by two blanks. |
| CVV-4 | Specifies that the CVV is to be computed as four bytes, followed by one blank. |
| CVV-5 | Specifies that the CVV is to be computed as five bytes. |

### `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 '000' is supported.

CVV_key_A_Identifier

Direction: Input/Output    Type: String

The CVV_key_A_Identifier parameter specifies an address that contains 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-A key that encrypts information in the CVV process.

CVV_key_B_Identifier

Direction: Input/Output    Type: String

The CVV_key_B_Identifier parameter specifies an address that contains 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.

CVV_value

Direction: Output          Type: String

The CVV_value parameter specifies an address that points to the place in application data storage that will be used to store the computed 5-byte character output value.

Restrictions

None

Required commands

This verb requires the Generate CVV command (offset X'00DF') to be enabled in the active role.

Usage notes

None

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 18.

The parameters for CSNBCSGJ are shown here.
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[] key_a_id,
    byte[] key_b_id,
    byte[] generated_cvv);
CVV Verify (CSNBCSV)

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. The 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.

Format

```
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

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

Direction: Input Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. The parameter rule_array_count must be 0, 1, or 2.
rule_array

Direction: Input  Type: String

Keywords that provide control information to the verb. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage. Table 65 describes these keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAN data length</strong> (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><strong>CVV length</strong> (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
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 ‘000’ is supported.

**CVV_key_A_Identifier**

Direction: Input/Output  
Type: String

The `CVV_key_A_Identifier` parameter specifies an address that contains a 64-byte internal key token or a key label of a single-length DATA, MAC, or MACVER key that decrypts information in the CCV process. The internal key token contains the Key-A key that encrypts information in the CVV process.

**CVV_key_B_Identifier**

Direction: Input/Output  
Type: String

The `CVV_key_B_Identifier` parameter specifies an address that contains a 64-byte internal key token or a key label of a single-length DATA, MAC, or MACVER key that decrypts information in the CCV process. The internal key token contains the Key-B key that decrypts information in the CVV process.

**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**

None

**Required commands**

This verb requires the Verify CVV command (offset X'00E0') to be enabled in the active role.

**Usage notes**

None

**JNI version**

This verb has a Java Native Interface (JNI) version, which is named CSNBCCSVJ. See [Building Java applications to use with the CCA JNI](#) on page 18.

The parameters for CSNBCCSVJ are shown here.
public native void CSNCSVJ(
    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[] key_a_id,
    byte[] key_b_id,
    byte[] generated_cvv);
Encrypted PIN Generate (CSNBEPG)

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

```
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

return_code

Direction: Output  Type: Integer

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

reason_code

Direction: Output  Type: Integer

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

Direction: Input/Output Type: Integer

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

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

PIN_generating_key_identifier

Direction: Input/Output Type: String

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 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 1.

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

Keywords that provide control information to the verb. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage. The rule_array keywords are shown in Table 66.

Table 66. Encrypted PIN Generate - Process rules

<table>
<thead>
<tr>
<th>Process Rule</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 to 16 bytes in length, uppercase, left-justified, and padded on the right with space characters. Table 67 describes the array elements.

<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' to X'F') of the encrypted validation data to decimal digits (X'0' to X'9').</td>
</tr>
<tr>
<td>Trans_sec_parm</td>
<td>For Interbank only, sixteen digits. Eleven right-most 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 to 16 characters of hexadecimal account data left-justified and padded on the right with blanks.</td>
</tr>
</tbody>
</table>

Table 68 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>
<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 284 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 or keyword for the PIN block format. Otherwise, ensure 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 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.

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 data storage. The information in the variable will be ignored, but the variable must be declared. To enter a sequence number, do the following:
- 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 to 65,535.
- For the 4704-EPP PIN block format, enter a value in the range from 0 to 255.

encrypted_PIN_block
Direction: Output Type: String
The field where the verb returns the 8-byte encrypted PIN.

Restrictions
The format control specified in the PIN profile must be NONE.

Required commands
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>Generate Formatted and Encrypted 3624 PIN</td>
</tr>
<tr>
<td>GBP-PIN</td>
<td>X'00B1'</td>
<td>Generate Formatted and Encrypted German Bank Pool PIN</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>X'00B2'</td>
<td>Generate Formatted and Encrypted InterBank PIN</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.

Usage notes
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 18.

The parameters for CSNBEPGJ are shown here.
Encrypted PIN Generate (CSNBEPG)

```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**

```
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**

**return_code**
- **Direction:** Output
- **Type:** Integer

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

**reason_code**
- **Direction:** Output
- **Type:** Integer

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

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

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**
- **Direction:** Input/Output
- **Type:** String
Encrypted PIN Translate (CSNBPTR)

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 *rule_array* parameter, 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 284 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/DUKPT-IP or UKPTBOTH/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 284 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 PINLENnnn 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 Mode 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**
Encrypted PIN Translate (CSNBPTR)

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. The value can be 1, 2, or 3.

rule_array
Direction: Input  Type: String
The process rule for the verb, described in Table 69.

Table 69. Keywords for Encrypted PIN Translate

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Rules</strong> (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><strong>PIN Block Format and PIN Extraction Method</strong> (optional)</td>
<td></td>
</tr>
<tr>
<td>PIN Block Format and PIN Extraction Method</td>
<td>See &quot;PIN block format and PIN extraction method keywords&quot; on page 284 for additional information and a list of PIN block formats and PIN extraction method keywords.</td>
</tr>
<tr>
<td>Note:</td>
<td>If a PIN extraction method is not specified, the first one listed in Table 52 on page 285 for the PIN block format will be the default.</td>
</tr>
<tr>
<td><strong>DUKPT Keywords - Single length key derivation</strong> (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>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** (optional)
### Table 69. Keywords for Encrypted PIN Translate (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUKPT-IP</td>
<td>The <code>input_PIN_encrypting_key_identifier</code> is derived as a double length key. The <code>input_PIN_encrypting_key_identifier</code> must be a KEYGENKY key with the UKPT usage bit enabled. The <code>input_PIN_profile</code> must be 48 bytes and contain the key serial number.</td>
</tr>
<tr>
<td>DUKPT-OP</td>
<td>The <code>output_PIN_encrypting_key_identifier</code> is derived as a double length key. The <code>output_PIN_encrypting_key_identifier</code> must be a KEYGENKY key with the UKPT usage bit enabled. The <code>output_PIN_profile</code> must be 48 bytes and contain the key serial number.</td>
</tr>
<tr>
<td>DUKPT-BH</td>
<td>Both the <code>input_PIN_encrypting_key_identifier</code> and the <code>output_PIN_encrypting_key_identifier</code> are derived as a double length key. Both the <code>input_PIN_encrypting_key_identifier</code> and the <code>output_PIN_encrypting_key_identifier</code> must be KEYGENKY keys with the UKPT usage bit enabled. Both the <code>input_PIN_profile</code> and the <code>output_PIN_profile</code> 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.
- 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 284 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 284 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**
- None

**Required commands**
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>Translate PIN with No Format-Control to No Format-Control</td>
</tr>
<tr>
<td>REFORMAT</td>
<td>NONE</td>
<td>NONE</td>
<td>X'00B7'</td>
<td>Reformat PIN with No Format-Control to No Format-Control</td>
</tr>
</tbody>
</table>

This verb also requires the Unique Key Per Transaction, ANSI X9.24 command (offset X'00E1') to be enabled if you employ UKPT processing.

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.

**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 Mode 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 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 through 9) as the PAD digit.
Encrypted PIN Translate (CSNBPTR)

Usage notes
Some PIN block formats are known by several names. The following table shows the additional names.

Table 70. Additional names for PIN formats

<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 18.

The parameters for CSNBPTRJ are shown here.

```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_PAN_data,
    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 (CSNBPV)

Use the Encrypted PIN Verify verb to verify that one of the following customer selected trial 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

```
CSNBPV(
    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

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

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

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String
Encrypted PIN Verify (CSNBPRV)

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 284](#) 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 284](#) 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 Mode 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. The value can be 1, 2, or 3.

**rule_array**

Direction: Input  
Type: String  
The process rule for the PIN verify algorithm, described in [Table 71](#).

**Table 71. Keywords for Encrypted PIN Verify control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm Value</strong> (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>
</tbody>
</table>

**PIN Block Format and PIN Extraction Method** (optional)

See “PIN block format and PIN extraction method keywords” on page 284 for additional information and a list of PIN block formats and PIN extraction method keywords.

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 Mode access control point (bit X'0313') must be enabled using a TKE workstation.

**Note:** If a PIN extraction method is not specified, the first one listed in [Table 52 on page 285](#) for the PIN block format will be the default.

**DUKPT Keyword - Single length key derivation** (optional)

UKPTIPIN  
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.

**DUKPT Keyword - double length key derivation** (optional)

DUKPT-IP  
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.

**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 through 16, for the PIN to be verified.

**data_array**

Direction: Input  
Type: Integer
Encrypted PIN Verify (CSNBPRVR)

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 72 describes the array elements.

Table 72. 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 through 9.</td>
</tr>
<tr>
<td>PIN_offset</td>
<td>Offset data for IBM-PINO. One to twelve numeric characters, 0 through 9, left-justified 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-justified. 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 to 16 characters of hexadecimal account data left-justified and padded on the right with blanks.</td>
</tr>
</tbody>
</table>

Table 73 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 73. 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>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPVV</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Restrictions
None

Required commands
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>Verify Encrypted 3624 PIN</td>
</tr>
<tr>
<td>GBP-PIN</td>
<td>X'00AC'</td>
<td>Verify Encrypted 3624 PIN</td>
</tr>
<tr>
<td>VISA-PVV, VISAPVV4</td>
<td>X'00AD'</td>
<td>Verify Encrypted Visa PVV</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>X'00AE'</td>
<td>Verify Encrypted InterBank PIN</td>
</tr>
</tbody>
</table>
This verb also requires the Unique Key Per Transaction, ANSI X9.24 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 Mode command (offset X'0313') in the active role.

**Usage notes**
None

**Related information**
The algorithms are discussed in detail in Appendix D, “PIN formats and algorithms,” on page 477.

**JNI version**
This verb has a Java Native Interface (JNI) version, which is named CSNBPVRJ. See “Building Java applications to use with the CCA JNI” on page 18.

The parameters for CSNBPVRJ are shown here.

```java
public native void CSNBPVRJ(
    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
);
```
PIN Change/Unblock (CSNBPCU)

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

```
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

return_code

Direction: Output  Type: Integer
PIN Change/Unblock (CSNBPCU)

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

**reason_code**

Direction: Output Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

Direction: Input Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. The valid values are 1 and 2.

**rule_array**

Direction: Input Type: String

Keywords that provide control information to the verb. The keywords are left-justified in an 8-byte field and padded on the right with blanks. The keywords must be in contiguous storage. Specify one or two of the options from Table 74.

Table 74. Keywords for PIN Change/Unblock control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong> (optional)</td>
<td></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><strong>PIN processing method</strong> (required)</td>
<td></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
PIN Change/Unblock (CSNBPCU)

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 DKYL0 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

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 122 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
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 IPIENC 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.
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
PIN Change/Unblock (CSNBPCU)

Currently this field is reserved. The value of this parameter should be 0.

**output_PIN_data**

Direction: Input  
Type: String  
Currently this field is reserved.

**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 VISAPCU2), 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**

None

**Required commands**

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.

<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>Generate PIN Change Using OPINENC</td>
<td>Required if either the <code>new_reference_PIN_key</code> or the <code>current_reference_PIN_key</code> are permitted to be an OPINENC key type.</td>
</tr>
<tr>
<td>IPINENC</td>
<td>X'00BD'</td>
<td>Generate PIN Change Using IPINENC</td>
<td>Required if either the <code>new_reference_PIN_key</code> or the <code>current_reference_PIN_key</code> 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 through 22) of B'1111', the Generate Diversified Key (DALL with DKYGENKY Key Type) 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 Mode command (offset X’0313’) in the active role.

**Usage notes**

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 18.

The parameters for CSNBPCUJ are shown here.

```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,
    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

```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

**return_code**

- **Direction:** Output
- **Type:** Integer

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

**reason_code**

- **Direction:** Output
- **Type:** Integer

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” lists the reason codes.

**exit_data_length**

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

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**

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

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

- **Direction:** Input
- **Type:** Integer
A pointer to an integer variable containing the number of elements in the `rule_array` variable. The valid values are 0 and 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. See Table 75.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enciphering mode (optional)</strong></td>
<td></td>
</tr>
<tr>
<td>TDES-CBC</td>
<td>Use CBC mode to encipher the message (default).</td>
</tr>
<tr>
<td>TDES-ECB</td>
<td>Use EBC mode to encipher the message.</td>
</tr>
</tbody>
</table>

**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 4K.

**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
Secure Messaging for Keys (CSNBSKY)

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

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**

None

**Required commands**

This verb requires the Secure Messaging for Keys command (offset X'0273') to be enabled in the active role.

**Usage notes**

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 18.

The parameters for CSNBSKYJ are shown here.
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,  
    byte[] initialization_vector,  
    hikmNativeInteger key_offset,  
    hikmNativeInteger key_offset_field_length,  
    byte[] cipher_text,  
    byte[] output_chaining_value);
Secure Messaging for PINs (CSNBSPN)

Secure Messaging for PINs (CSNBSPN)

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

```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

**return_code**

Direction: Output Type: Integer

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

**reason_code**

Direction: Output Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output Type: Integer

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**

Direction: Input/Output Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

Direction: Input Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. The valid values are 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 justified and padded on the right with blanks. See Table 76.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enciphering mode (optional)</strong></td>
<td></td>
</tr>
<tr>
<td>TDES-CBC</td>
<td>Use CBC mode to encipher the message (default).</td>
</tr>
<tr>
<td>TDES-ECB</td>
<td>Use EBC mode to encipher the message.</td>
</tr>
<tr>
<td><strong>PIN encryption (optional)</strong></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 (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 284 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.
Secure Messaging for PINs (CSNBSPN)

**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 4K.

**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**

None

**Required commands**

This verb requires the Secure Messaging for PINs command (offset X'0274') to be enabled in the active role.

**Usage notes**

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 18.

The parameters for CSNBSPNJ are shown here.

```java
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

```
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

**return_code**

Direction: Output, Type: Integer

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

**reason_code**

Direction: Output, Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output, Type: Integer

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**

Direction: Input/Output, Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**
Transaction Validation (CSNBTRV)

Direction: Input  Type: Integer
A pointer to an integer variable containing the number of elements in the
rule_array variable. The valid values are 1 or 2.

rule_array
Direction: Input  Type: String
Keywords that provide control information to the verb. The keywords are
left-justified in an 8-byte field and padded on the right with blanks. The
keywords must be in contiguous storage. Specify one or two of the values in
Table 77

Table 77. Keywords for Transaction Validation control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>American Express card security codes (required)</strong></td>
<td></td>
</tr>
<tr>
<td>CSC-3</td>
<td>3-digit card security code (CSC) located on the signature panel. VERIFY implied. This is the default.</td>
</tr>
<tr>
<td>CSC-4</td>
<td>4-digit card security code (CSC) located on the signature panel. VERIFY implied.</td>
</tr>
<tr>
<td>CSC-5</td>
<td>5-digit card security code (CSC) located on the signature panel. VERIFY 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. GENERATE implied.</td>
</tr>
<tr>
<td><strong>Operation (optional)</strong></td>
<td></td>
</tr>
<tr>
<td>VERIFY</td>
<td>Specifies verification of the value presented in the validation values variable.</td>
</tr>
<tr>
<td>GENERATE</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 78.

Table 78. 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:</td>
</tr>
<tr>
<td></td>
<td>55555 = CSC 5 value</td>
</tr>
<tr>
<td></td>
<td>4444 = CSC 4 value</td>
</tr>
<tr>
<td></td>
<td>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
None

Required commands
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>Generate CSC 3, 4, and 5 Values</td>
</tr>
<tr>
<td>VERIFY</td>
<td>CSC-3</td>
<td>X'0292'</td>
<td>Verify CSC 3 Values</td>
</tr>
<tr>
<td></td>
<td>CSC-4</td>
<td>X'0293'</td>
<td>Verify CSC 4 Values</td>
</tr>
<tr>
<td></td>
<td>CSC-5</td>
<td>X'0294'</td>
<td>Verify CSC 5 Values</td>
</tr>
</tbody>
</table>

Usage notes
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 18.

The parameters for CSNBTRVJ 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,
    byte[] rule_array,
    hikmNativeInteger transaction_key_length,
    byte[] transaction_key,
    hikmNativeInteger transaction_info_length,
    byte[] transaction_info,
    hikmNativeInteger validation_values_length,
    byte[] validation_values);
```

---

Transaction Validation (CSNBTRV)

Chapter 9. Financial services
Chapter 10. Using digital signatures

This chapter describes the PKA verbs that support using digital signatures to authenticate messages.

"Digital Signature Generate (CSNDDSG)"
"Digital Signature Verify (CSNDDSV)" on page 348
"MDC Generate (CSNBMDG)" on page 352

Digital signature verbs

Digital Signature Generate (CSNDDSG)

Use the Digital Signature Generate verb to generate a digital signature using an RSA private key.

The RSA private key must be valid for signature usage. This verb supports the following methods:

- ANSI X9.30
- ANSI X9.31
- ISO 9796-1
- RSA DSI PKCS 1.0 and 1.1
- Padding on the left with zeros

Note: The maximum signature length is 256 bytes (2048 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 516.

You select the method of formatting the text through the rule_array parameter.

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 79 on page 344). D is the text string supplied in the hash variable.

Format

```
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  )
```
Digital Signature Generate (CSNDDSG)

Parameters

return_code

direction: output type: integer

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

reason_code

direction: output type: integer

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” lists the reason codes.

exit_data_length

direction: input/output type: integer

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

direction: input/output type: string

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

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 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. Table 79 lists the keywords. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

Table 79. Keywords for Digital Signature Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital Signature Formatting Method (optional)</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>
</tbody>
</table>
Table 79. Keywords for Digital Signature Generate control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**Hash Method Specification: Required with X9.31**

<table>
<thead>
<tr>
<th>Hash Method</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**PKA_private_key_identifier_length**

Direction: Input  Type: Integer

The length of the `PKA_private_key_identifier` field. The maximum size is 2500 bytes.

**PKA_private_key_identifier**

Direction: Input  Type: String

An internal token or label of the PKA 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.

**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 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.

**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 RIPEMD-160. See the `rule_array` parameter for more information.

**signature_field_length**

Direction: Input/Output  Type: Integer
Digital Signature Generate (CSNDDSG)

The length in bytes of the `signature_field` to contain the generated digital signature.

**Note:** 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). This field is updated with the minimum byte length of the digital signature. The maximum size is 256 bytes.

**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.

**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-justified) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-justified within the `signature_field`. Any unused bytes to the right are undefined.

**Restrictions**

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. It 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**

This verb requires the PKA96 Digital Signature Generate command (offset X'0100') to be enabled in the active role.

With the use of the Override DSG ZERO-PAD Length Restriction command (offset X'030C'), the hash-length restriction does not apply when using ZERO-PAD formatting.

**Usage notes**

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 18.
The parameters for CSNDDSGJ are shown here.

```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)

Use the Digital Signature Verify verb to verify a digital signature using a PKA public key. This service supports the following methods:
- ANSI X9.30
- ANSI X9.31
- ISO 9796
- RSA DSI PKCS 1.0 and 1.1
- Padding on the left with zeros

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 516.

Note: The maximum signature length is 256 bytes (2048 bits).

Format

```
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

return_code
Direction: Output   Type: Integer
The return code specifies the general result of the verb. Appendix A, "Return codes and reason codes" lists the return codes.

reason_code
Direction: Output   Type: Integer
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" lists the reason codes.

exit_data_length
Direction: Input/Output   Type: Integer
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
Direction: Input/Output   Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 or 1.

**rule_array**

**Direction:** Input  
**Type:** String

Keywords that provide control information to the verb. A keyword specifies the method to use to verify the RSA digital signature. Table 80 lists the keywords. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

**Table 80. Keywords for Digital Signature Verify control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>TPK-ONLY</td>
<td>Permits the use of only public keys contained in trusted blocks. By specifying this Trusted Public Key only 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>
</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 2500 bytes.

**PKA_public_key_identifier**

**Direction:** Input  
**Type:** String

A token or label of the PKA public key.
Digital Signature Verify (CSNDDSV)

**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 256 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 256 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-justified) of a string whose length is the minimum number of bytes that can contain the digital signature. This string is left-justified within the *signature_field*.

**Restrictions**
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**
This verb requires the PKA96 Digital Signature Verify command (offset X'0101') to be enabled in the active role.

**Usage notes**
None

**Related information**
- 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 18.

The parameters for CSNDDSVJ are shown here.
Digital Signature Verify (CSNDDSV)

```java
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 );
```
MDC Generate (CSNBMDG)

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 355 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 495.

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, PADMDMC-2, or PADMDMC-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 FIRST 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**

```c
CSNBMDG(    return_code,    reason_code,    exit_data_length,    exit_data,    text_length,    text,    rule_array_count,    rule_array,    chaining_vector,    MDC )
```

**Parameters**

*return_code*

- **Direction:** Output
- **Type:** Integer

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

*reason_code*

- **Direction:** Output
- **Type:** Integer

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” lists the reason codes.

*exit_data_length*

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

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*

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

The data that is passed to an installation exit. Exits are not supported and no
exit data is allowed in this parameter.

*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 355.
MDC Generate (CSNBMDG)

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. The 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. Table 81 lists the keywords.
Each keyword is left-justified in an 8-byte field and padded on the right with
blanks. All keywords must be in contiguous storage.

Table 81. Keywords for MDC Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Segmenting and key control</strong> (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><strong>Algorithm mode</strong> (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**
- 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**
This verb requires the Generate MDC command (offset X'008A') to be enabled in the active role.

The user must enable the Generate MDC command with a Trusted Key Entry (TKE) workstation before using this verb.

**Usage notes**
None

**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 18.

The parameters for CSNBMDGJ are shown here.

```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);
```

**Related information**
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, please read the following
important information related to the integrity of your data.

**Overview:** It was discovered that under certain conditions, the CSNBMDG verb of
the CCA Support Program generates incorrect MDC values. This section describes
in detail each scenario that results in these incorrect MDC values.

**Terminology:** The following terminology 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 through 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

**Notes:**
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 **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 **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. **MIDDLE** calls process text in multiples of 8 (for example,
16, 24, 32). As with **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 **ONLY** or **LAST**.

**Examples:**
1. Assume a text length of 19 for **FIRST**, 6 for **MIDDLE**, and 10 for **LAST**.

   \[ \text{TTL} = 19 + 6 + 10 = 35 \text{ bytes.} \]

   When **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 **MIDDLE** is called, \(\text{RTL} = 19\), \(\text{COL} = 19 - 16 = 3\), and \(\text{NTL} = \text{COL} + \text{text length} = 3 + 6 = 9\) text bytes to process.
After the **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. **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 **PADMDC-2** or **PADMDC-4** method is required.

2. Assume a text length of 19 for **FIRST**, 25 for **MIDDLE**, and 12 for **LAST**.

TTL = 19 + 25 + 12 = 56 bytes.

When **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 **MIDDLE** is called, RTL = 19, COL = 19 - 16 = 3, and NTL = COL + text length = 3 + 25 = 28 bytes to process.

After the **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.

**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.
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.

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.

The MDC value is calculated incorrectly whenever:

- Algorithm PADMDC-2, PADMDC-4
- Segmenting and key control ONLY, LAST

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.
- 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:
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 (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:
• 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.
Chapter 11. Managing PKA cryptographic keys

This chapter describes the verb that generate and manage PKA keys.

- "PKA Key Generate (CSNDPKG)"
- "PKA Key Import (CSNDPKI)" on page 367
- "PKA Key Token Build (CSNDPKB)" on page 370
- "PKA Key Token Change (CSNDKTC)" on page 377
- "PKA Key Translate (CSNDPKT)" on page 380
- "PKA Public Key Extract (CSNDPKX)" on page 384
- "PKA Symmetric Key Export (CSNDSYX)" on page 386
- "PKA Symmetric Key Generate (CSNDSYG)" on page 389
- "PKA Symmetric Key Import (CSNDSYI)" on page 394

Verbs to manage PKA keys

PKA Key Generate (CSNDPKG)

Use the PKA Key Generate verb to generate RSA keys for use on the cryptographic coprocessor or other CCA systems. 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 PKA Key Generate verb will generate a key with the same modulus length and the same exponent.

Input to the PKA Key Generate verb is either a skeleton key token that has been built by the PKA key token build server 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.

RSA key generation requires the following information in the input skeleton token:
- Size of the modulus in bits. The modulus for modulus-exponent form keys is between 512 and 1024 bits in length.
- The CRT modulus is between 512 and 2048 bits in length.

RSA key generation has the following restrictions:
- For modulus-exponent, there are restrictions on modulus, public exponent, and private exponent.
- For CRT, there are restrictions on dp, dq, U, and public exponent.

See the Key value structure in "PKA Key Token Build (CSNDPKB)" on page 370 for a summary of restrictions.
PKA Key Generate (CSNDPKG)

Format

```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

code

return_code

Direction: Output  Type: Integer

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

reason_code

Direction: Output  Type: Integer

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” lists the reason codes.

exit_data_length

Direction: Input/Output  Type: Integer

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

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

rule_array_count

Direction: Input  Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. Value can be 1 or 2.

rule_array

Direction: Input  Type: String

A keyword that provides control information to the verb. See Table 82 on page 365 for a list. A keyword is left-justified in an 8-byte field and padded on the right with blanks.
Table 82. Keywords for PKA Key Generate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Private Key Encryption (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.</td>
</tr>
<tr>
<td>MASTER</td>
<td>Encipher the private key under the master key.</td>
</tr>
<tr>
<td>RETAIN</td>
<td>Retains the private key within the cryptographic engine and returns the public key.</td>
</tr>
<tr>
<td></td>
<td>Before using this keyword, see the information about retained keys in &quot;Using retained keys&quot; on page 276.</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.</td>
</tr>
</tbody>
</table>

**regeneration_data_length**

Direction: Input  
Type: Integer

The `regeneration_data_length` parameter must be between 8 and 256 bytes inclusive.

**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.

**skeleton_key_identifier_length**

Direction: Input  
Type: Integer

The length of the `skeleton_key_identifier` parameter in bytes. The maximum allowed value is 2500 bytes.

**skeleton_key_identifier**

Direction: Input  
Type: String

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.

**transport_key_identifier**

Direction: Input  
Type: String

A 64-byte field to contain a DES key identifier. This field must be binary zeros, unless the XPORT rule is specified. For XPORT rule, this is an IMPORTER or EXPORTER key or the label of an IMPORTER or EXPORTER key that is used to encrypt the generated key. If you specify a label, it must resolve uniquely to either an IMPORTER or EXPORTER key. Only valid for RSA keys.

**generated_key_token_length**

Direction: Input/Output  
Type: Integer
PKA Key Generate (CSNDPKG)

The length of the generated key token. The field is checked to ensure that it is at least equal to the token being returned. The maximum size is 2500 bytes. On output, this field is updated with the actual token length.

generated_key_token
Direction: Input/Output  Type: String

The internal token or label of the generated RSA key. If a label is specified in the generated_key_token parameter, the generated_key_token_length returned to the application will be the same as the input length. If a label is specified in the generated_key_token parameter, a record must already exist in the PKA key storage file with this same label or the service will fail.

Restrictions
None

Required commands
This verb requires the PKA96 PKA Key Generate command (offset X'0103') to be enabled in the active role.

With the CLEAR rule-array keyword, enable PKA Clear Key Generate (offset X'0205' in the hardware.

To generate keys based on the value supplied in the regeneration_data variable, you must enable one of these commands:
- When not using the RETAIN keyword, enable the Permit Regeneration Data command (offset X'027D').
- When using the RETAIN keyword, enable the Permit Regeneration Data for Retained Keys command (offset X'027E').

Usage notes
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 18.

The parameters for CSNDPKGJ are shown here.

```
FORMAT
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 private key token. (This consists of a PKA private key and public key.) The secret values of the key can be clear or encrypted under a limited-authority DES importer key.

This verb can also import a clear PKA key. The PKA Key Token Build verb creates a clear PKA key token.

Output of this verb is a CCA internal token of the RSA private key.

**Format**

```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**

- **return_code**
  - Direction: Output
  - Type: Integer
  - The return code specifies the general result of the verb. Appendix A, “Return codes and reason codes” lists the return codes.

- **reason_code**
  - Direction: Output
  - Type: Integer
  - 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” lists the reason codes.

- **exit_data_length**
  - Direction: Input/Output
  - Type: Integer
  - 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**
  - Direction: Input/Output
  - Type: String
  - The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

- **rule_array_count**
  - Direction: Input
  - Type: Integer
  - A pointer to an integer variable containing the number of elements in the rule_array variable. This must be 0.
PKA Key Import (CSNDPKI)

**rule_array**
- Direction: Input
- Type: String
- Reserved field. This field is not used, but you must specify it.

**source_key_identifier_length**
- Direction: Input
- Type: Integer
- The length of the `source_key_identifier` parameter. The maximum size is 2500 bytes.

**source_key_identifier**
- Direction: Input
- Type: String
- The external token or label of a PKA private key. This is the output of the PKA Key Generate (CSNDPKG) verb or the PKA Key Token Build (CSNDPKB) verb. If encrypted, it was created on another platform.

**importer_key_identifier**
- Direction: Input/Output
- Type: String
- A DES internal token or the label of an IMP-PKA key. This is a limited authority key-encrypting key. It 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 2500 bytes.

**target_key_identifier**
- Direction: Input/Output
- Type: String
- This field contains the internal token or label of the imported PKA private key. 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. If no label is specified on input, this field should be set to binary zeros on input.

**Restrictions**
This verb imports RSA keys of 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.

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**
This verb requires the PKA96 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 Convert Trusted Block from External to Internal Form command (offset X'0311').
Usage notes
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. See “Building Java applications to use with the CCA JNI” on page 18.

The parameters for CSNDPKIJ are shown here.

```
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
);
```
Use this utility to build external PKA key tokens containing unenciphered private RSA 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 you can use as input to the PKA Key Generate verb (see Table 82 on page 365). You can also input to this verb a clear unenciphered public RSA key and return the public key in a token format that other PKA verbs can use directly.

You can also use this verb to build a key token for an RSA private key in optimized Chinese Remainder Theorem (CRT) form.

**Format**

```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,
    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,
    key_token_length,
    key_token)
```

**Parameters**

**return_code**

- **Direction:** Output
- **Type:** Integer

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

**reason_code**

- **Direction:** Output
- **Type:** Integer

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” lists the reason codes.

**exit_data_length**

- **Direction:** Ignored
- **Type:** Integer

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**

Direction: Input/Output  
Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**rule_array_count**

Direction: Input  
Type: Integer

A pointer to an integer variable containing the number of elements in the `rule_array` variable. Value must be 1 or 2.

**rule_array**

Direction: Input  
Type: String

One or two keywords that provide control information to the verb. Table 83 lists the keywords. The keywords must be in contiguous storage with each of the keywords left-justified in its own 8-byte location and padded on the right with blanks.

**Table 83. Keywords for PKA Key Token Build control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Type</strong> (required)</td>
<td></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) form. The parameter <code>key_value_structure</code> 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 <code>key_value_structure</code> 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 <code>key_value_structure</code> identifies the input values, if supplied.</td>
</tr>
<tr>
<td><strong>Key Usage Control</strong> (optional)</td>
<td></td>
</tr>
</tbody>
</table>
| KEY-MGMT | Indicates that an RSA private key can be used in both the PKA 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. |
| KM-ONLY | Indicates that an RSA 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. |
| NO-XLATE | The RSA key cannot be used as a key-encrypting-key for "PKA Key Translate (CSNDPKT)" on page 380.  
**Note:** 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. |
| RSAMEVAR | RSA-Modulus-Exponent-Variant (RSAMEVAR) is a type ‘09’ key token for RSA, named VAR.OPK.  
**Note:** Key tokens created with this key usage cannot be passed to the PKA Key Generate verb for creating RETAIN (retained) keys. |
| SIG-ONLY | Indicates that an RSA 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. |
### PKA Key Token Build (CSNDPKB)

#### Table 83. Keywords for PKA Key Token Build control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLATE-OK</td>
<td>The RSA key can be used as a key-encrypting-key for the PKA Key Translate (CSNDPKT) on page 380.  Note: Key tokens created with this key usage 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 84.

#### Table 84. 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>RSA-CRT</td>
<td>2500</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-justified within their respective structure elements and padded on the left with binary zeros. If the leading bits of the modulus are zeros, don’t count them in the length. Table 85 defines the structure and contents as a function of key type.

#### Table 85. 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</strong> (Optimized RSA, Chinese Remainder Theorem form, RSA-CRT)</td>
</tr>
<tr>
<td>000</td>
<td>002</td>
<td>Modulus length in bits (512 to 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>
<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 skeleton_key_token in the PKA Key Generate verb. Maximum size of p + q is 256 bytes.</td>
</tr>
<tr>
<td>Offset</td>
<td>Length (bytes)</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>-------------</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 <code>skeleton_key_token</code> 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 <code>skeleton_key_token</code> 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 <code>skeleton_key_token</code> 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 <code>skeleton_key_token</code> 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 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 RSA-PUBL keyword is used and must not exceed 128 when the 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>
</tbody>
</table>
### Table 85. 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>004</td>
<td>002</td>
<td>Public exponent field length in bytes, “YYY.” This value must not exceed 256 when the RSA-PUBL keyword is used and must not exceed 128 when the 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>
<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 +</td>
<td>XXX +</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 + 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); the You need not specify this value if you specify RSA-PUBL in the rule_array parameter.</td>
</tr>
</tbody>
</table>

### Notes:
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-justified within the key structure element field.
3. You must supply all values in the structure to create a token containing an RSA 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 upper case 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.
reserved_1_length
Direction: Input        Type: Integer
Length in bytes of a reserved parameter. You must set this variable to 0.

reserved_1
Direction: Input        Type: String
The reserved_1 parameter identifies a string that is reserved. The verb ignores it.

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
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
PKA Key Token Build (CSNDPKB)

- The field is updated with the exact length of the `key_token` created. On input, a size of 2500 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
None

Required commands
None

Usage notes
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](#) on page 18.

The parameters for CSNDPKBJ are shown here.

```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 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 token_length,
      byte[] token);
```
PKA Key Token Change (CSNDKTC)

The PKA Key Token Change verb changes 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.

Format

```
CSNDKTC(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    key_identifier_length,
    key_identifier)
```

Parameters

**return_code**

Direction: Output  Type: Integer

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

**reason_code**

Direction: Output  Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output  Type: Integer

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**

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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

The process rule for the verb. The keyword must be in eight bytes of contiguous storage, left-justified, and padded on the right with blanks. See Table 86 on page 378.
### Keywords for PKA Key Token Change control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCMK</td>
<td>Re-enciphers a DES 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 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 &quot;Key Storage on z/OS (RTNMK-focused)&quot; on page 236), 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 63 will be returned. For more information, see &quot;Key storage with Linux for IBM System z, in contrast to z/OS for IBM System z&quot; on page 235.</td>
</tr>
</tbody>
</table>

**key_identifier_length**
- Direction: Input
- Type: Integer
- The length of the key_identifier parameter. The maximum size is 2500 bytes.

**key_identifier**
- Direction: Input/Output
- Type: String
- An internal RSA private key token.

**Restrictions**
- None

**Required commands**
- This verb requires the PKA96 PKA Key Token Change command (offset X'0102') to be enabled in the active role.

**Usage notes**
- 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 18.
The parameters for CSNDKTCJ are shown here.

```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

```
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

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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.
PKA Key Translate (CSNDPKT)

exit_data
Direction: Input/Output    Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

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
The process rule for the verb. The keyword must be in eight bytes of contiguous storage, left-justified, and padded on the right with blanks. See Table 87.

Table 87. Keywords for PKA Key Translate control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smartcard Format (required)</td>
<td></td>
</tr>
<tr>
<td>SCVISA</td>
<td>This keyword indicates translating the key into the smart card Visa proprietary format.</td>
</tr>
<tr>
<td>SCCOMME</td>
<td>This keyword indicates translating the key into the smart card Modulus-Exponent format.</td>
</tr>
<tr>
<td>SCCOMCRT</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 382 for details on the type of transport key that can be used.

target_transport_key_identifier_length
PKA Key Translate (CSNDPKT)

**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 parameter `target_key_token`. See **Usage notes** for details on the type of transport key that can be used.

**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 callable service. 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**

CCA RSA ME tokens will not be translated to the SCCOMCRT format. CCA RSA CRT tokens will not be translated to the SCCOMME format. SCVISA supports only Modulus-Exponent (ME) keys.

**Required commands**

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>Translate from CCA RSA to SC Visa format</td>
</tr>
<tr>
<td>SCCOMME</td>
<td>X'0319'</td>
<td>Translate from CCA RSA to SC ME format</td>
</tr>
<tr>
<td>SCCOMCRT</td>
<td>X'031A'</td>
<td>Translate from CCA RSA to SC CRT format</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transport key control</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-PIN</td>
<td>X'031B'</td>
<td>XLATE from encryption under source EXP KEK to target EXP KEK</td>
</tr>
<tr>
<td>GBP-PIN</td>
<td>X'031C'</td>
<td>XLATE from encryption under source IMP KEK to target EXP KEK</td>
</tr>
<tr>
<td>INBK-PIN</td>
<td>X'031D'</td>
<td>XLATE from encryption under source IMP KEK to target IMP KEK</td>
</tr>
</tbody>
</table>

**Usage notes**

There are access control points that control use of the format `rule_array` keys and the type of transport keys that can be used. All of these access control points are enabled in the default role.
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 18.

The parameters for CSNDPKTJ are shown here.

```
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)

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

```
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

**return_code**

Direction: Output Type: Integer

The return code specifies the general result of the verb. [Appendix A, “Return codes and reason codes”](#) lists the return codes.

**reason_code**

Direction: Output Type: Integer

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”](#) lists the reason codes.

**exit_data_length**

Direction: Ignored Type: Integer

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**

Direction: Ignored Type: String

The data that is passed to an installation edit. Exits are not supported and no exit data is allowed in this parameter.

**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

Reserved field. This field is not used, but you must specify it.
**source_key_identifier_length**
- **Direction:** Input
- **Type:** Integer

The length of the `source_key_identifier` parameter. The maximum size is 2500 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 PKA Key Import or from PKA Key Generate.

**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**
None

**Required commands**
None

**Usage notes**
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 18.

The parameters for CSNDPKXJ are shown here.

```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
);
```
PKA Symmetric Key Export (CSNDSYX)

Use this verb to transfer an application-supplied symmetric key (a DATA key) from encryption under the DES master key to encryption under an application-supplied RSA public key. The application-supplied DATA key must be a DES internal key token or the label of such a token in the DES key storage file. The PKA Symmetric Key Import verb can import the PKA-encrypted form at the receiving node.

Format

```c
CSNDSYX(
    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    rule_array_count,
    rule_array,
    source_key_identifier_length,
    source_key_identifier,
    RSA_public_key_identifier_length,
    RSA_public_key_identifier,
    RSA_enciphered_key_length,
    RSA_enciphered_key )
```

Parameters

return_code

Direction: Output  Type: Integer

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

reason_code

Direction: Output  Type: Integer

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” lists the reason codes.

exit_data_length

Direction: Input/Output  Type: Integer

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

Direction: Input/Output  Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

rule_array_count

Direction: Input  Type: Integer

A pointer to an integer variable containing the number of elements in the rule_array variable. Value must be 1 or 2.
Keywords that provide control information to the verb. Table 88 lists the keywords. Each keyword is left-justified in 8-byte fields and padded on the right with blanks. All keywords must be in contiguous storage.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>(one, optional)</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</td>
<td>(required)</td>
</tr>
<tr>
<td>PKCSOAEP</td>
<td>Specifies using the method found in RSA DSI PKCS #1 V2 OAEP. See “PKCS #1 formats” on page 516.</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 516.</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>The clear key is right-justified 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>

source_key_identifier_length

Direction: Input                     Type: Integer

The length of the source_key_identifier parameter. The maximum length is 3500 bytes.

source_key_identifier

Direction: Input                     Type: String

A pointer to a string variable containing either an operational AES or DES key-token, or the key label of an operational AES or DES key-token to be exported.

RSA_public_key_identifier_length

Direction: Input                     Type: Integer

The length of the RSA_public_key_identifier parameter. The maximum size is 3500 bytes.

RSA_public_key_identifier

Direction: Input                     Type: String

A pointer to a string variable containing a PKA96 RSA internal or external key-token with the RSA public-key of the remote node that is to import the exported key.

RSA_enciphered_key_length

Direction: Input/Output              Type: Integer

The length of the RSA_enciphered_key parameter. On input, this is a pointer to an integer variable containing the number of bytes of data in the RSA_enciphered_key variable. On output, the variable is updated with the actual length of the RSA_enciphered_key variable. The maximum length is 3500 bytes.

RSA_enciphered_key
PKA Symmetric Key Export (CSNDSYX)

<table>
<thead>
<tr>
<th>Direction: Output</th>
<th>Type: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>This field contains the output RSA-enciphered key, protected by the public key specified in the RSA_public_key_identifier field.</td>
<td></td>
</tr>
</tbody>
</table>

**Restrictions**

None

**Required commands**

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'0130'</td>
<td>Export AES Key (PKCSOAEP or PKCS-1.2)</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>X'0105'</td>
<td>Symmetric Key Export (PKCS-1.2/OAEP)</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>AES</td>
<td>X'0131'</td>
<td>Export AES Key (ZERO-PAD)</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>X'023E'</td>
<td>ZERO-PAD Symmetric Key Export</td>
</tr>
</tbody>
</table>

**Usage notes**

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.

**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 18.

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 RSA_public_key_token_length,
    byte[] RSA_public_key_token,
    hikmNativeInteger RSA_enciphered_key_length,
    byte[] RSA_enciphered_key
);```

---

388 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
PKA Symmetric Key Generate (CSNDSYG)

Use the PKA Symmetric Key Generate verb to 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 encryption can be in the form of an internal token encrypted under the DES master key or in the external form encrypted under a key-encrypting key. You can import the PKA-encrypted form by using the PKA Symmetric Key Import verb at the receiving node.

Also use the PKA Symmetric Key Generate verb to generate any 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 514 for more details about PKA92 formatting.

Format

```
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,
    DES_enciphered_key_token_length,
    DES_enciphered_key_token,
    RSA_enciphered_key_length,
    RSA_enciphered_key)
```

Parameters

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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” lists the reason codes.

**exit_data_length**

Direction: Input/Output  
Type: Integer

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**

Direction: Input/Output  
Type: String
The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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

Keywords that provide control information to the verb. Table 89 lists the keywords. The recovery method is the method to use to recover the symmetric key. Each keyword is left-justified in an 8-byte field and padded on the right with blanks. All keywords must be in contiguous storage.

**Table 89. Keywords for PKA Symmetric Key Generate control information**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</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>Key-formatting method</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 using the method found in RSA DSI PKCS #1V2 OAEP.</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-justified 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>Key length</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>Key length</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>KEYLNN16</td>
<td>Generates a double-length DES DATA key. This is the default for AES keys.</td>
</tr>
<tr>
<td>KEYLNN24</td>
<td>Generates a triple-length DES DATA key. Valid only for AES keys.</td>
</tr>
<tr>
<td>KEYLNN32</td>
<td>Generates a 32-byte AES key. Valid only for AES keys</td>
</tr>
<tr>
<td>Encipherment method for the local enciphered copy of the key</td>
<td>(one, optional for use with PKCSOAEP, PKCS-1.2, and ZERO-PAD)</td>
</tr>
</tbody>
</table>
Table 89. Keywords for PKA Symmetric Key Generate control information (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
</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

Chapter 11. Managing PKA cryptographic keys 391
PKA 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

None

Required commands

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>Generate AES DATA Key (PKCSOAEP or PKCS-1.2) Symmetric Key Generate PKCS-1.2/OAEP</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>X'023F'</td>
<td>Symmetric Key Generate PKCS-1.2/OAEP</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>AES</td>
<td>X'012D'</td>
<td>Generate AES DATA Key (ZERO-PAD) ZERO-PAD Symmetric Key Generate</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>X'023C'</td>
<td>Symmetric Key Generate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'010D'</td>
<td>PKA92 Symmetric Key Generate</td>
</tr>
</tbody>
</table>

Usage notes

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 514). 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 DES_enciphered_key_token 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 18.

The parameters for CSNDSYGJ are shown here.
### PKA Symmetric Key Generate (CSNDSYG)

**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
);
```
PKA Symmetric Key Import (CSNDSYI)

Use the PKA Symmetric Key Import verb to import a symmetric (DES) DATA key enciphered under an RSA public key. It 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

```c
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

**return_code**
- Direction: Output
- Type: Integer
- The return code specifies the general result of the verb. [Appendix A, “Return codes and reason codes”](#) lists the return codes.

**reason_code**
- Direction: Output
- Type: Integer
- 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”](#) lists the reason codes.

**exit_data_length**
- Direction: Input/Output
- Type: Integer
- 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**
- Direction: Input/Output
- Type: String
- The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.

**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 or 2.
The keyword that provides control information to the verb [Table 90] provides a list. The recovery method is the method to use to recover the symmetric key. The keyword is left-justified in an 8-byte field and padded on the right with blanks.

Table 90. Keywords for PKA Symmetric Key Import control information

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</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>
<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 using the method found in RSA DSI PKCS #1V2 OAEP.</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-justified 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>

The length of the RSA_enciphered_key parameter. The maximum size is 3500 bytes.

The key to import, protected under an RSA public key. The encrypted key is in the low-order bits (right-justified) of a string whose length is the minimum number of bytes that can contain the encrypted key. This string is left-justified within the RSA_enciphered_key parameter.

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.

An internal RSA private key token or label whose corresponding public key protects the symmetric key.

The length of the target_key_identifier parameter.
PKA Symmetric Key Import (CSNDSYI)

- **target_key_identifier**
  - 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**

None.

**Required commands**

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'012E'</td>
<td>Import AES Key (PKCSOAEP or PKCS-1.2)</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>X'0106'</td>
<td>Symmetric Key Import PKCS-1.2/OAEP</td>
</tr>
<tr>
<td>ZERO-PAD</td>
<td>AES</td>
<td>X'012F'</td>
<td>Import AES Key (ZERO-PAD)</td>
</tr>
<tr>
<td></td>
<td>DES</td>
<td>X'023C'</td>
<td>ZERO-PAD Symmetric Key Import</td>
</tr>
<tr>
<td>PKA92 and DATA, MAC,</td>
<td>DES</td>
<td>X'0235'</td>
<td>PKA92 Symmetric Key Import</td>
</tr>
<tr>
<td>MACVER, KEYGENKEY,</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>PINGEN, PINVER, IPINENC,</td>
<td>DES</td>
<td>X'0236'</td>
<td>PKA92 Symmetric Key Import with PIN Keys</td>
</tr>
<tr>
<td>or OPINENC key</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Usage notes**

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 CEX3C, an Environment Identifier (EID) of zero will be set in the coprocessor. This will be interpreted by the PKA 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 18.
The parameters for CSNDSYIJ are shown here.

```
FORMAT

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
);
```
Remote Key Export (CSNDRKX)

Remote Key Export (CSNDRKX)

The 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 408. 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.

The 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" on page 399.

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.
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.
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” on page 398 Step 2 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” on page 398 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” on page 398 Step 2 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

```
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

**return_code**

- Direction: Output
- Type: Integer

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

**reason_code**

- Direction: Output
- Type: Integer

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

**exit_data_length**

- Direction: Input/Output
- Type: Integer

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**

- Direction: Input/Output
- Type: String

The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.
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
A pointer to a string variable. This variable is currently 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 516.

certificateParms_length
Direction: Input    Type: Integer
A pointer to an integer variable containing the number of bytes of data in the certificateParms variable. The length must be 36 bytes if the certificate_length variable is 0, else the length must be 0.
Remote Key Export (CSNDRKX)

**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 91.

**Table 91. Keywords for Remote Key Export certificate_parms parameter**

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description (offsets and lengths are in bytes)</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 digital signature hash formatting method. The following value is defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’01’</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>X’01’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’02’</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-justified 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:

- 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.

**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 the 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 RKX key token or an internal CCA DES key-token.

**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 RKX key-token.

**Notes:**

1. If the key token is a CCA DES key-token, its XPORT-OK control vector bit (bit 17) must be 1, or 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.

- 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**

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 495 and "Modification Detection Code calculation" on page 495. 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.
Remote Key Export (CSNDRKX)

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 through 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
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
This verb requires the 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 Replicate Key command (offset X’00DB’) to be enabled 
to replicate a single-length source key (either from a CCA DES key-token or an 
RKX 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.

Note: A role with X’00DB’ enabled can also use the Key Generate verb with the 
SINGLE-R key-length keyword.

Usage notes
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][1] on page 18.

The parameters for CSNDRKXJ are shown here.
**Remote Key Export (CSNDRKX)**

```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_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,
    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 444.

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 Create a Trusted 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 Activate an Inactive Trusted 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.

The creation of an external trusted block typically takes place in a highly secure environment. Use “PKA Key Import (CSNDPKI)” on page 367 to import an active external trusted block into the desired node. The imported internal trusted block can then be used as input to “Remote Key Export (CSNDRKX)” on page 398 in order to generate or export DES keys.

Create 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.

The 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.
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.

Create 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**

```plaintext
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**

**return_code**

Direction: Output  
Type: Integer

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

**reason_code**

Direction: Output  
Type: Integer

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

**exit_data_length**
Trusted Block Create (CSNDTBC)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction: Input/Output</th>
<th>Type: Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A pointer to an integer value containing the length of the string (in bytes) that is returned by the <code>exit_data</code> value. This parameter should point to a value of zero, to ensure compatibility with any future extension or other operating environment.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction: Input/Output</th>
<th>Type: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>The data that is passed to an installation exit. Exits are not supported and no exit data is allowed in this parameter.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction: Input</th>
<th>Type: Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A pointer to an integer variable containing the number of elements in the <code>rule_array</code> variable. The value must be 1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction: Input</th>
<th>Type: Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 <code>rule_array</code> keywords are shown in Table 92.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 92. Keywords for Trusted Block Create control information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Keyword</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
</tr>
<tr>
<td>INACTIVE</td>
</tr>
<tr>
<td>ACTIVE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction: Input</th>
<th>Type: Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A pointer to an integer variable containing the number of bytes in the <code>input_block_identifier</code> variable. The maximum length is 3500 bytes.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction: Input</th>
<th>Type: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 444. The trusted block key-token will be updated by the verb and returned in the <code>trusted_block_identifier</code> 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.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction: Input</th>
<th>Type: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>A pointer to a string variable containing the transport key identifier.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

410 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
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 108 with a key type of IMPORTER and a rule_array keyword of IMPORT.

trusted_block_identifier_length

Direction: Input/Output  
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. 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**

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 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>rule_array keyword</th>
<th>Offset</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>INACTIVE</td>
<td>X'030F'</td>
<td>Create a Trusted Block in Inactive Form.</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>X'0310'</td>
<td>Activate an Inactive Trusted Block.</td>
</tr>
</tbody>
</table>

**Usage notes**

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 18.

The parameters for CSNDTBCJ are shown here.
Trusted Block Create (CSNDTBC)

```
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,
    byte[] transport_key_identifier,
    hikmNativeInteger trusted_block_length,
    byte[] trusted_block_identifier);
```
Appendix A. Return codes and reason codes

This section 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 15.

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 93.

<table>
<thead>
<tr>
<th>Hex value</th>
<th>Decimal value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 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 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 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 12</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 16</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') through 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 are listed in Table 94.

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</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.</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>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>
</tbody>
</table>

**Reason codes that accompany return code 4**

Reason codes that accompany return code 4 are listed in Table 95.

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</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>
</tbody>
</table>
### Table 95. Reason codes for return code 4 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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 to 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>
</tbody>
</table>

### Reason codes that accompany return code 8

Reason codes that accompany return code 8 are listed in Table 96.

### Table 96. Reason codes for return code 8

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</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 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>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>
</tbody>
</table>
Table 96. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
Table 96. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>8</td>
<td>066 (042)</td>
<td>The recovered encryption block failed validation checking.</td>
</tr>
<tr>
<td>8</td>
<td>067 (043)</td>
<td>RSA encryption failed.</td>
</tr>
<tr>
<td>8</td>
<td>068 (044)</td>
<td>RSA decryption failed.</td>
</tr>
<tr>
<td>8</td>
<td>072 (048)</td>
<td>The value the size parameter specifies is not valid (too small, too large, negative, or zero).</td>
</tr>
<tr>
<td>8</td>
<td>085 (055)</td>
<td>The date or the time value is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>090 (05A)</td>
<td>Access control checking failed. See the Required Commands descriptions for the failing verb.</td>
</tr>
<tr>
<td>8</td>
<td>091 (05B)</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</td>
<td>092 (05C)</td>
<td>The user profile is expired.</td>
</tr>
<tr>
<td>8</td>
<td>093 (05D)</td>
<td>The user profile has not yet reached its activation date.</td>
</tr>
<tr>
<td>8</td>
<td>094 (05E)</td>
<td>The authentication data (for example, passphrase) is expired.</td>
</tr>
<tr>
<td>8</td>
<td>095 (05F)</td>
<td>Access to the data is not authorized.</td>
</tr>
<tr>
<td>8</td>
<td>096 (060)</td>
<td>An error occurred reading or writing the secure clock.</td>
</tr>
<tr>
<td>8</td>
<td>100 (064)</td>
<td>The PIN length is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>101 (065)</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</td>
<td>102 (066)</td>
<td>The value of the decimalization table is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>103 (067)</td>
<td>The value of the validation data is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>104 (068)</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</td>
<td>105 (069)</td>
<td>The value of the transaction_security parameter is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>106 (06A)</td>
<td>The PIN-block format keyword is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>107 (06B)</td>
<td>The format control keyword is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>108 (06C)</td>
<td>The value or the placement of the padding data is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>109 (06D)</td>
<td>The extraction method keyword is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>110 (06E)</td>
<td>The value of the PAN data is not numeric character data.</td>
</tr>
<tr>
<td>8</td>
<td>111 (06F)</td>
<td>The sequence number is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>112 (070)</td>
<td>The PIN offset is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>114 (072)</td>
<td>The PVV value is not valid.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>8</td>
<td>116 (074)</td>
<td>The clear PIN value is not valid. For example, digits other than 0 - 9 were found.</td>
</tr>
<tr>
<td>8</td>
<td>120 (078)</td>
<td>An origin or destination identifier is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>121 (079)</td>
<td>The value of the <code>inbound_key</code> or <code>source_key</code> parameter is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>122 (07A)</td>
<td>The value of the <code>inbound_KEY_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>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 the <code>generated_key_identifier</code> parameter specifies is not valid or it is not consistent with the value 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 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 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>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>---------</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>8</td>
<td>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>
</tr>
<tr>
<td>8</td>
<td>205 (0CD)</td>
<td>The X9.23 ciphering method is not consistent with the use of the CONTINUE keyword.</td>
</tr>
<tr>
<td>8</td>
<td>323 (143)</td>
<td>The ciphering method the Decipher verb used does not match the ciphering method the Encipher verb used.</td>
</tr>
<tr>
<td>8</td>
<td>335 (14F)</td>
<td>Either the specified cryptographic hardware component or the environment cannot implement this function.</td>
</tr>
<tr>
<td>8</td>
<td>340 (154)</td>
<td>One of the input control vectors has odd parity.</td>
</tr>
<tr>
<td>8</td>
<td>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>
</tr>
<tr>
<td>8</td>
<td>374 (176)</td>
<td>Less data was supplied than expected or less data exists than was requested.</td>
</tr>
<tr>
<td>8</td>
<td>377 (179)</td>
<td>A key-storage error occurred.</td>
</tr>
<tr>
<td>8</td>
<td>382 (17E)</td>
<td>A time-limit violation occurred.</td>
</tr>
<tr>
<td>8</td>
<td>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>
</tr>
<tr>
<td>8</td>
<td>387 (183)</td>
<td>The cryptographic hardware component reported that the user ID or role ID is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>393 (189)</td>
<td>The command was not processed because the profile cannot be used.</td>
</tr>
<tr>
<td>8</td>
<td>394 (18A)</td>
<td>The command was not processed because the expiration date was exceeded.</td>
</tr>
<tr>
<td>8</td>
<td>397 (18D)</td>
<td>The command was not processed because the active profile requires the user to be verified first.</td>
</tr>
<tr>
<td>8</td>
<td>398 (18E)</td>
<td>The command was not processed because the maximum PIN or password failure limit is exceeded.</td>
</tr>
<tr>
<td>8</td>
<td>407 (197)</td>
<td>There is a PIN-block consistency-check-error.</td>
</tr>
<tr>
<td>8</td>
<td>442 (1BA)</td>
<td>DES keys with replicated halves are not allowed.</td>
</tr>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>8</td>
<td>605 (25D)</td>
<td>The number of output bytes is greater than the number that is permitted.</td>
</tr>
<tr>
<td>8</td>
<td>703 (2BF)</td>
<td>A new master-key value is one of the weak DES keys.</td>
</tr>
<tr>
<td>8</td>
<td>704 (2C0)</td>
<td>A new master key cannot have the same master key version number as the current master-key.</td>
</tr>
<tr>
<td>8</td>
<td>705 (2C1)</td>
<td>Both exporter keys specify the same key-encrypting key.</td>
</tr>
<tr>
<td>8</td>
<td>706 (2C2)</td>
<td>Pad count in deciphered data is not valid.</td>
</tr>
<tr>
<td>8</td>
<td>707 (2C3)</td>
<td>The master-key registers are not in the state required for the requested function.</td>
</tr>
<tr>
<td>8</td>
<td>714 (2CA)</td>
<td>A reserved parameter must be a null pointer or an expected value.</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>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>
</tbody>
</table>
Table 96. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 782 (30E)</td>
<td>The function-control-vector header is not valid.</td>
<td></td>
</tr>
<tr>
<td>8 783 (30F)</td>
<td>The command is not permitted by the function-control-vector value.</td>
<td></td>
</tr>
<tr>
<td>8 784 (310)</td>
<td>The operation you requested cannot be performed because the user profile is in use.</td>
<td></td>
</tr>
<tr>
<td>8 785 (311)</td>
<td>The operation you requested cannot be performed because the role is in use.</td>
<td></td>
</tr>
<tr>
<td>8 1025 (401)</td>
<td>The registered public key or retained private key name already exists.</td>
<td></td>
</tr>
<tr>
<td>8 1026 (402)</td>
<td>The key name (registered public key or retained private key) does not exist.</td>
<td></td>
</tr>
<tr>
<td>8 1027 (403)</td>
<td>Environment identifier data is already set.</td>
<td></td>
</tr>
<tr>
<td>8 1028 (404)</td>
<td>Master key share data is already set.</td>
<td></td>
</tr>
<tr>
<td>8 1029 (405)</td>
<td>There is an error in the Environment Identifier (EID) data.</td>
<td></td>
</tr>
<tr>
<td>8 1030 (406)</td>
<td>There is an error in using the master key share data.</td>
<td></td>
</tr>
<tr>
<td>8 1031 (407)</td>
<td>There is an error in using registered public key or retained private key data.</td>
<td></td>
</tr>
<tr>
<td>8 1032 (408)</td>
<td>There is an error in using registered public key hash data.</td>
<td></td>
</tr>
<tr>
<td>8 1033 (409)</td>
<td>The public key hash was not registered.</td>
<td></td>
</tr>
<tr>
<td>8 1034 (40A)</td>
<td>The public key was not registered.</td>
<td></td>
</tr>
<tr>
<td>8 1035 (40B)</td>
<td>The public key certificate signature was not verified.</td>
<td></td>
</tr>
<tr>
<td>8 1037 (40D)</td>
<td>There is a master key shares distribution error.</td>
<td></td>
</tr>
<tr>
<td>8 1038 (40E)</td>
<td>The public key hash is not marked for cloning.</td>
<td></td>
</tr>
<tr>
<td>8 1039 (40F)</td>
<td>The registered public key hash does not match the registered hash.</td>
<td></td>
</tr>
<tr>
<td>8 1040 (410)</td>
<td>The master key share enciphering key failed encipher.</td>
<td></td>
</tr>
<tr>
<td>8 1041 (411)</td>
<td>The master key share enciphering key failed decipher.</td>
<td></td>
</tr>
<tr>
<td>8 1042 (412)</td>
<td>The master key share digital signature generate failed.</td>
<td></td>
</tr>
<tr>
<td>8 1043 (413)</td>
<td>The master key share digital signature verify failed.</td>
<td></td>
</tr>
<tr>
<td>8 1044 (414)</td>
<td>There is an error in reading VPD data from the adapter.</td>
<td></td>
</tr>
<tr>
<td>8 1045 (415)</td>
<td>Encrypting the cloning information failed.</td>
<td></td>
</tr>
<tr>
<td>8 1046 (416)</td>
<td>Decrypting the cloning information failed.</td>
<td></td>
</tr>
<tr>
<td>8 1047 (417)</td>
<td>There is an error loading the new master key from the master key shares.</td>
<td></td>
</tr>
<tr>
<td>8 1048 (418)</td>
<td>The clone information has one or more sections that are not valid.</td>
<td></td>
</tr>
<tr>
<td>8 1049 (419)</td>
<td>The master key share index is not valid.</td>
<td></td>
</tr>
<tr>
<td>8 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>
<td></td>
</tr>
</tbody>
</table>
Table 96. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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’ to 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 Public 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’12’ 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>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 callable service 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>
</tbody>
</table>
### Table 96. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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, CSNDKPI, CSNDRXX, 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>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>
<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>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>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>---------</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>The RSA public key is too small to encrypt the DES key.</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>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>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 (832)</td>
<td>Invalid encrypted key length in the AES token, when an encrypted key is present.</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>A clear key was provided when a secure key was required.</td>
</tr>
<tr>
<td>8</td>
<td>6000 (1770)</td>
<td>The specified device is already allocated.</td>
</tr>
</tbody>
</table>
### Table 96. Reason codes for return code 8 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<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>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 are listed in Table 97.

### Table 97. Reason codes for return code 12

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</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>
</tbody>
</table>
### Table 97. Reason codes for return code 12 (continued)

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<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>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>2073 (819)</td>
<td>TKE command received when TKE disabled.</td>
</tr>
<tr>
<td>12</td>
<td>2074 (81A)</td>
<td>CPRB version is incorrect. An incorrect control structure was passed from the host to the card.</td>
</tr>
<tr>
<td>12</td>
<td>2101 (835)</td>
<td>The AES flags in the Function Control Vector are invalid.</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 are listed in Table 98.

<table>
<thead>
<tr>
<th>Return code Decimal</th>
<th>Reason code Decimal (Hex)</th>
<th>Meaning</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>
<tr>
<td>Return code Decimal</td>
<td>Reason code Decimal (Hex)</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<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 CSND syi, 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>
</tbody>
</table>
Appendix B. Key token formats

For debugging purposes, this appendix provides the formats for: DES internal key tokens, DES external key tokens, AES internal key tokens, null key tokens, PKA key tokens, and trusted blocks.

Format of the internal AES key token

Table 99 shows the format for an internal AES 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 99. Internal AES 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'000000'.</td>
</tr>
<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;Internal AES key-token flag byte&quot; on page 430.</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-justified 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>
</tbody>
</table>
### Key token formats

#### Table 99. Internal AES key token format, version X'04' (continued)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>56 - 57</td>
<td>Clear-key bit length&lt;br&gt;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&lt;br&gt;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>

#### Internal AES key-token flag byte

Table 100 shows the format for an internal AES key token.

#### Table 100. Internal AES 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. 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 twos 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

Table 101 shows the format for a clear internal key token.

#### Table 101. Internal clear key token format

<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>
</tbody>
</table>
### Table 101. Internal clear key token format (continued)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Flag byte</td>
</tr>
<tr>
<td></td>
<td>Bit</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'00000000000000000000000000000000' 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'00000000')</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>B'00' Indicates single-length key (version 0 only).</td>
</tr>
<tr>
<td></td>
<td>B'01' Indicates double-length key (version 1 only).</td>
</tr>
<tr>
<td></td>
<td>B'10' 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 (TVV).</td>
</tr>
</tbody>
</table>

### Format of the DES internal key token

Table 102 shows the format for a DES internal key token.

### Table 102. Internal key token format

<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>Bit</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: When bit 3 is on and bit 4 is off, AKEK is a single-length key (eight bytes).</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved (X'00')</td>
</tr>
<tr>
<td>8 - 15</td>
<td>Master key verification pattern (MKVP)</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. The value is encrypted under the master key.</td>
</tr>
</tbody>
</table>
### Key token formats

**Table 102. Internal key token format (continued)**

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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. The right half of the double-length key or Part B of the triple-length key is encrypted under the master key.</td>
</tr>
<tr>
<td>32 - 39</td>
<td>The control vector (CV) for a single-length key or the left half of the control vector for a double-length key.</td>
</tr>
<tr>
<td>40 - 47</td>
<td>X'0000000000000000' if a single-length key or the right half of the control vector for a double-length operational key.</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. Part C of a triple-length key is encrypted under the master key.</td>
</tr>
<tr>
<td>56 - 58</td>
<td>Reserved (X'000000')</td>
</tr>
<tr>
<td>59 bits 0 and 1</td>
<td>B'10' Indicates KEK.</td>
</tr>
<tr>
<td></td>
<td>B'00' Indicates DES for DATA keys or the system default algorithm for a KEK.</td>
</tr>
<tr>
<td></td>
<td>B'01' Indicates DES for a KEK.</td>
</tr>
<tr>
<td>59 bits 2 and 3</td>
<td>B'00' Indicates single-length key (version 0 only).</td>
</tr>
<tr>
<td></td>
<td>B'01' Indicates double-length key (version 1 only).</td>
</tr>
<tr>
<td></td>
<td>B'10' 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 (TVV).</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

**Table 103** shows the format for a DES external key token.

**Table 103. Format of DES external key tokens**

<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>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'0000000000000000')</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'0000000000000000' 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>
</tbody>
</table>

Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
Table 103. Format of DES external key tokens (continued)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 - 47</td>
<td>X'000000000000000000000000' if single-length key or right half of CV for</td>
</tr>
<tr>
<td></td>
<td>double-length key</td>
</tr>
<tr>
<td>48 - 55</td>
<td>X'00000000000000000000000000000000' if a single-length key, double-length</td>
</tr>
<tr>
<td></td>
<td>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>B'01'</td>
</tr>
<tr>
<td>59 bits 4 - 7</td>
<td>B'10'</td>
</tr>
<tr>
<td>60 - 63</td>
<td>Token validation value (see “Token validation value” on page 430 for a</td>
</tr>
<tr>
<td></td>
<td>description).</td>
</tr>
</tbody>
</table>
### Table 105. RSA Public Key Token (continued)

**RSA Public Key Section (required)**

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>012 xxx</td>
<td>Public key exponent (this is generally a 1, 3, or 64 to 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>
<td></td>
</tr>
<tr>
<td>12 + xxx yyy</td>
<td>Modulus, n.</td>
<td></td>
</tr>
</tbody>
</table>

### Format of RSA private external key tokens

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 following section identifiers:
  - X'02' indicates a modulus-exponent form RSA private key section (not optimized) with modulus length of up to 1024 bits for use with the CEX3C
  - X'08' indicates an optimized Chinese Remainder Theorem form private key section with modulus bit length of up to 2048 bits for use with the CEX3C
- 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 106 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-justified and padded with zeros to the left.

### Table 106. RSA private external key token basic record format

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of 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. The private key is either in cleartext or enciphered with a transport key-encrypting key.</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 zero.</td>
<td></td>
</tr>
</tbody>
</table>
Table 106. RSA private external key token basic record format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RSA Private Key Section (required)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• For 1024-bit Modulus-Exponent form refer to “RSA private key token, 1024-bit modulus-exponent external form”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• For 2048-bit Chinese Remainder Theorem form refer to “RSA private key token, 2048-bit Chinese remainder theorem external form” on page 436</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RSA Public Key Section (required)</strong></td>
<td></td>
<td></td>
</tr>
<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. <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 to 256-byte quantity). e must be odd and 1&lt;e&lt;n. (Frequently, the value of e is 2&lt;sup&gt;16&lt;/sup&gt; + 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>
<tr>
<td><strong>Private Key Name (optional)</strong></td>
<td></td>
<td></td>
</tr>
<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-justified, 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 form**

This RSA private key token and the external X’02’ token is supported on the CEX3C. **Table 107** shows the format.

Table 107. RSA private key token, 1024-bit modulus-exponent external format

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of 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>
</tbody>
</table>
Table 107. RSA private key token, 1024-bit modulus-exponent external format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
</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, binary zero.  |
| 030              | 020             | SHA-1 hash of the optional key-name section. If there is no key-name section, then 20 bytes of X'00'.  |
| 050              | 004             | Key use flag bits.  
|                  |                 | Bit | Meaning | When Set On |
|                  |                 | 0   | Key management usage permitted. |
|                  |                 | 1   | Signature usage not permitted. |
|                  |                 |     | All other bits reserved, set to binary zero. |
| 054              | 006             | Reserved; set to binary zero.  |
| 060              | 024             | Reserved; set to binary zero.  |
| 084              |                 | Start of the optionally-encrypted secure subsection.  |
| 084              | 024             | Random number, confounder.  |
| 108              | 128             | Private-key exponent, \(d \cdot d = e^\prime \mod ((p-1)(q-1))\), 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 encrypted 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, \(n = pq\) where \(p\) and \(q\) are prime and \(1 < n < 2^{1024}\).  |

RSA private key token, 2048-bit Chinese remainder theorem external form

This RSA private key token is supported on the CEX3C. Table 108 shows the format.

Table 108. RSA private key token, 2048-bit Chinese remainder theorem 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'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'.  |
### Table 108. RSA private key token, 2048-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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</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: &quot;ppp.&quot;</td>
</tr>
<tr>
<td>056</td>
<td>002</td>
<td>Length of prime number, q, in bytes: &quot;qqq.&quot;</td>
</tr>
<tr>
<td>058</td>
<td>002</td>
<td>Length of ( d_p ), in bytes: &quot;rrr.&quot;</td>
</tr>
<tr>
<td>060</td>
<td>002</td>
<td>Length of ( d_q ), in bytes: &quot;sss.&quot;</td>
</tr>
<tr>
<td>062</td>
<td>002</td>
<td>Length of ( U ), in bytes: &quot;uuu.&quot;</td>
</tr>
<tr>
<td>064</td>
<td>002</td>
<td>Length of modulus, ( n ), in bytes: &quot;nnn.&quot;</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: &quot;xxx.&quot;</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>pp</td>
<td>Prime number, ( p ).</td>
</tr>
<tr>
<td>132 + pp</td>
<td>qqq</td>
<td>Prime number, ( q ).</td>
</tr>
<tr>
<td>132 + pp + qq</td>
<td>rrr</td>
<td>( d_p = d \mod (p - 1) )</td>
</tr>
<tr>
<td>132 + pp + qq + rrr</td>
<td>sss</td>
<td>( d_q = d \mod (q - 1) )</td>
</tr>
<tr>
<td>132 + pp + qq + rrr + sss</td>
<td>uuu</td>
<td>( U = q^{-1} \mod (p) ).</td>
</tr>
<tr>
<td>132 + pp + qq + 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></td>
<td></td>
<td>End of the optionally-encrypted secure 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 the private key is enciphered. They are enciphered under a double-length transport key using the TDES (CBC outer chaining) algorithm.</td>
</tr>
<tr>
<td>132 + pp + qq + rrr + sss + uuu + xxx</td>
<td>nnn</td>
<td>Modulus, ( n = pq ) where ( p ) and ( q ) are prime and ( 1 &lt; n &lt; 2^{2048} ).</td>
</tr>
</tbody>
</table>

### Format of the RSA private internal key token

An RSA private internal key token contains the following sections:

- A required PKA token header, starting with the token identifier X'1F'.
- 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-justified and padded with zeros to the left.
### Table 109. RSA private internal key token basic record format

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of 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>
<tr>
<td><strong>RSA Private Key Section and Secured Subsection (required)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>For 1024-bit X'02' Modulus-Exponent form refer to <a href="#">RSA private key token, 1024-bit modulus-exponent internal form for cryptographic coprocessor feature</a> on page 439</td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>For 1024-bit X'06' Modulus-Exponent form refer to <a href="#">RSA private key token, 1024-bit modulus-exponent internal form for CEX3C</a> on page 440</td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>For 2048-bit X'08' Chinese Remainder Theorem form refer to <a href="#">RSA private key token, 2048-bit Chinese remainder theorem internal form</a> on page 441</td>
<td></td>
</tr>
<tr>
<td><strong>RSA Public Key Section (required)</strong></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td>012</td>
<td>xxx</td>
<td>Public key exponent (this is generally a 1, 3, or 64 to 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>
<tr>
<td><strong>Private Key Name (optional)</strong></td>
<td></td>
<td></td>
</tr>
<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-justified, 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>
<tr>
<td><strong>Internal Information Section (required)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>004</td>
<td>Eye catcher ‘PKTN’.</td>
</tr>
</tbody>
</table>
Table 109. RSA private internal key token basic record format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>004</td>
<td>004</td>
<td>PKA token type.</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>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<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 form for cryptographic coprocessor feature

Table 110 shows the format of the RSA private key token, 1024-bit modulus-exponent internal form for cryptographic coprocessor feature.

Table 110. RSA private internal key token, 1024-bit modulus-exponent form for cryptographic coprocessor feature

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of 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>
<tr>
<td>029</td>
<td>001</td>
<td>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.</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>Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other bits reserved, set to binary zero.</td>
</tr>
</tbody>
</table>
Key token formats

Table 110. RSA private internal key token, 1024-bit modulus-exponent form for cryptographic coprocessor feature (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 = pq) where (p) and (q) are prime and (1 &lt; n &lt; 2^{1024}).</td>
</tr>
</tbody>
</table>

RSA private key token, 1024-bit modulus-exponent internal form for CEX3C

Table 111 shows the format for the RSA private key token, 1024-bit modulus-exponent internal form for CEX3C.

Table 111. RSA private internal key token, 1024-bit modulus-exponent form for CEX3C

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of 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: X'02' RSA private key.</td>
</tr>
<tr>
<td>029</td>
<td>001</td>
<td>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.</td>
</tr>
<tr>
<td>030</td>
<td>020</td>
<td>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.</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>054</td>
<td>006</td>
<td>Reserved; set to binary zero.</td>
</tr>
<tr>
<td>060</td>
<td>048</td>
<td>Object Protection Key (OPK) encrypted under the Asymmetric Keys Master Key using the ede3 algorithm.</td>
</tr>
<tr>
<td>108</td>
<td>128</td>
<td>Private key exponent d, encrypted under the OPK using the ede5 algorithm. (d = e^{-1} \mod ((p-1)(q-1))), and (1 &lt; d &lt; n) where (e) is the public exponent.</td>
</tr>
<tr>
<td>236</td>
<td>128</td>
<td>Modulus, (n = pq) where (p) and (q) are prime and (2^{512} &lt; n &lt; 2^{1024}).</td>
</tr>
</tbody>
</table>
### RSA private internal key token, 1024-bit modulus-exponent form for CEX3C (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</th>
<th>Number of Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>364</td>
<td>016</td>
<td>Asymmetric-Keys Master Key hash pattern.</td>
</tr>
<tr>
<td>380</td>
<td>020</td>
<td>SHA-1 hash value of the blinding information subsection cleartext, offset 400 to the end of the section.</td>
</tr>
<tr>
<td>400</td>
<td>002</td>
<td>Length of the random number r, in bytes: rrr.</td>
</tr>
<tr>
<td>402</td>
<td>002</td>
<td>Length of the random number r⁻¹, in bytes: iii.</td>
</tr>
<tr>
<td>404</td>
<td>002</td>
<td>Length of the padding field, in bytes: xxx.</td>
</tr>
<tr>
<td>406</td>
<td>002</td>
<td>Reserved; set to binary zeros.</td>
</tr>
<tr>
<td>408</td>
<td>Start of the encrypted blinding subsection</td>
<td></td>
</tr>
<tr>
<td>408 + rrr</td>
<td>iii</td>
<td>Random number r⁻¹ (used in blinding).</td>
</tr>
<tr>
<td>408 + rrr + iii</td>
<td>xxx</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

---

### RSA private key token, 2048-bit Chinese remainder theorem internal form

This RSA private key token is supported on the CEX3C. Table 112 shows the format.

#### Table 112. RSA private internal key token, 2048-bit Chinese remainder theorem internal format

<table>
<thead>
<tr>
<th>Offset (Decimal)</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>
<tr>
<td>028</td>
<td>001</td>
<td>Key format and security: X'08' Encrypted RSA private-key subsection identifier, Chinese Remainder form.</td>
</tr>
<tr>
<td>029</td>
<td>001</td>
<td>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.</td>
</tr>
<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>
</tbody>
</table>
### Key token formats

Table 112. RSA private internal key token, 2048-bit Chinese remainder theorem internal format (continued)

<table>
<thead>
<tr>
<th>Offset (Decimal)</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>Bit</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</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: &quot;ppp.&quot;</td>
</tr>
<tr>
<td>056</td>
<td>002</td>
<td>Length of prime number, q, in bytes: &quot;qqq.&quot;</td>
</tr>
<tr>
<td>058</td>
<td>002</td>
<td>Length of (d_p), in bytes: &quot;rrr.&quot;</td>
</tr>
<tr>
<td>060</td>
<td>002</td>
<td>Length of (d_q), in bytes: &quot;sss.&quot;</td>
</tr>
<tr>
<td>062</td>
<td>002</td>
<td>Length of (U), in bytes: &quot;uuu.&quot;</td>
</tr>
<tr>
<td>064</td>
<td>002</td>
<td>Length of modulus, n, in bytes: &quot;nnn.&quot;</td>
</tr>
<tr>
<td>066</td>
<td>002</td>
<td>Length of the random number (r), in bytes: &quot;ttt.&quot;</td>
</tr>
<tr>
<td>068</td>
<td>002</td>
<td>Length of the random number (r^{-1}), in bytes: &quot;iii.&quot;</td>
</tr>
<tr>
<td>070</td>
<td>002</td>
<td>Length of padding field, in bytes: &quot;xxx.&quot;</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>Asymmetric-Keys Master Key hash pattern.</td>
</tr>
<tr>
<td>092</td>
<td>032</td>
<td>Object Protection Key (OPK) encrypted under the Asymmetric-Keys 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>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</td>
<td>rrr</td>
<td>(d_p = d \mod (p - 1))</td>
</tr>
<tr>
<td>132 + ppp + qqq  + rrr</td>
<td>sss</td>
<td>(d_q = d \mod (q - 1))</td>
</tr>
<tr>
<td>132 + ppp + qqq  + rrr + sss</td>
<td>uuu</td>
<td>(U = q^{-1}\mod(p)).</td>
</tr>
<tr>
<td>132 + ppp + qqq  + rrr + sss + uuu + ttt</td>
<td>ttt</td>
<td>Random number (r) (used in blinding).</td>
</tr>
<tr>
<td>132 + ppp + qqq  + rrr + sss + uuu + ttt + iii</td>
<td>iii</td>
<td>Random number (r^{-1}) (used in blinding).</td>
</tr>
<tr>
<td>132 + ppp + qqq  + rrr + sss + uuu + ttt + iii + xxx</td>
<td>xxx</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>nnn</td>
<td>Modulus, (n = pq) where (p) and (q) are prime and (1 &lt; n &lt; 2^{2048}).</td>
</tr>
</tbody>
</table>

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.
**RSA variable modulus-exponent token formats**

Table 113 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 113. RSA variable modulus-exponent token formats

<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 through end of section (clear values)</td>
<td>sha1Hash</td>
<td>Hash over fields 7 through 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</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>

**PKA null key token**

Table 114 shows the format for a PKA null key token.

Table 114. Format of PKA null key tokens

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X'00' Token identifier (indicates 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>
</tbody>
</table>
Key token formats

Table 114. Format of PKA null key tokens (continued)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4–7</td>
<td>Ignored (should be zero).</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 PKA key tokens and is an integral part of a remote key-loading process. See “Remote key-loading” on page 445.

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).

A trusted block is an extension of CCA PKA 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.

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.

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.
Remote key-loading

Remote key-loading refers to the process of installing DES 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 DES encryption key that is used to encrypt other keys so they can be securely transmitted over unprotected paths.
- Distribution of operational DES keys or replacement KEKs, enciphered under a KEK currently installed in the device.

A new 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.

Trusted block sections

A trusted block is a concatenation of a header followed by an unordered set of sections. The data structures of these sections are summarized in Table 115.

<table>
<thead>
<tr>
<th>Section</th>
<th>Reference</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Table 116 on page 447</td>
<td>Trusted block token header</td>
</tr>
<tr>
<td>X'11'</td>
<td>Table 117 on page 447</td>
<td>Trusted block public key</td>
</tr>
<tr>
<td>X'12'</td>
<td>Table 118 on page 449</td>
<td>Trusted block rule</td>
</tr>
<tr>
<td>X'13'</td>
<td>Table 125 on page 456</td>
<td>Trusted block name (key label)</td>
</tr>
<tr>
<td>X'14'</td>
<td>Table 126 on page 456</td>
<td>Trusted block information</td>
</tr>
<tr>
<td>X'15'</td>
<td>Table 130 on page 458</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

Appendix B. Key token formats
Key token formats

- 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

- 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
- 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

Format of 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 116.

Table 116. Trusted block header

<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>X’1E’   External trusted block token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X’1F’   Internal trusted block token</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 446.

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’), and section X’14’ has two (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 sections.

**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 the following table:

Table 117. 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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+y).</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 public-key modulus field length in bytes, yyy</td>
</tr>
</tbody>
</table>
### Table 117. Trusted block trusted RSA public-key section (X'11') (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
</table>
| 012           | xxx            | Public-key exponent, $e$ (this field length is typically 1, 3, or 64 through 512 bytes). $e$ must be odd and $1 \leq e < n$. ($e$ is frequently valued to 3 or $2^{16} + 1 = 65537$, otherwise $e$ is of the same order of magnitude as the modulus).
|               |                | **Note:** 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. |
| 012 + xxx     | yyy            | RSA public-key modulus, $n$. $n = pq$, where $p$ and $q$ are prime and $2^{312} < n < 2^{4096}$. The field length is 64 through 512 bytes.
|               |                | **Note:** Keys with a modulus greater than 2048 bits are not supported in releases before Release 3.30. |
| 012 + xxx + yyy | 004           | Flags:
|               |                | X'00000000' Trusted block public key can be used in digital signature operations only |
|               |                | X'80000000' Trusted block public key can be used in both digital signature and key management operations |
|               |                | X'C0000000' Trusted block public key can be used in key management operations only |

**Note:** See "Number representation in trusted blocks" on page 446.

### Trusted block section X'12'

Trusted block section X'12' contains information that defines a rule. 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 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 RKX key-token
   - Export an RKX key-token
   - Export a CCA DES key-token
   - 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 allowed to have multiple sections. Section X'12' is optional. Multiple sections are allowed.

**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 the following table:
Table 118. 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 justified 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). 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>
<tr>
<td>017</td>
<td>001</td>
<td>Key-check algorithm identifier (all others are reserved and must not be used): Value Meaning 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 495. X'02' Compute the MDC-2 hash of the key. See Modification Detection Code calculation on page 495.</td>
</tr>
<tr>
<td>018</td>
<td>001</td>
<td>Symmetric encrypted output key format flag (all other values are reserved and must not be used). Return the indicated symmetric key-token using the sym_encrypted_key_identifier parameter. Value Meaning X'00' Return an RKX key-token encrypted under a variant of the MAC key. Note: This key format is permitted when the flags value (offset 012) is set to either: 1. Generate a new key 2. Export an existing key X'01' Return a CCA DES key-token encrypted under a transport key. Note: 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>
</tr>
<tr>
<td>019</td>
<td>001</td>
<td>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 Meaning 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.</td>
</tr>
</tbody>
</table>
Table 118. 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>020</td>
<td>yyy</td>
<td>Rule section subsections (tag-length-value objects). A series of zero through five objects in TLV format.</td>
</tr>
</tbody>
</table>

**Note:** See "Number representation in trusted blocks" on page 446.

**Trusted block section X'12' subsections**

Section X'12' has five rule subsections (tag-length-value objects) defined. These subsections are summarized in the following table:

Table 119. Summary of trusted block rule subsection

<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>
<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 446.

**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 the following table:

Table 120. 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>
</tbody>
</table>
Table 120. Transport key variant subsection (X'0001') 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>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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>Transport key variant.</td>
</tr>
<tr>
<td></td>
<td></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'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 446.

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 the following table:

Table 121. 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 justified 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>

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)
Key token formats

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 the following table:

Table 122. 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. 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 (yyy). Length must be 0, 8, 16, or 24. Also applies to the source key. Not applicable for key generation.</td>
</tr>
<tr>
<td>010</td>
<td>001</td>
<td>Output key variant length in bytes (xxx). Valid values are 0 or 8 through 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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If the length is not 0, 8, or 16, return an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If a nonzero length is less than the length of the key identified by the source_key_identifier parameter, return an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. Ignore all CV bit definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. If CCA DES key-token format, set the flag byte of the symmetric encrypted output key to indicate a CV value is present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. If the source key is eight bytes in length, do not replicate the key to 16 bytes</td>
</tr>
</tbody>
</table>
Table 122. 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>012 + xxx</td>
<td>yyy</td>
<td>CV. (See “Control vector table” on page 461.)</td>
</tr>
</tbody>
</table>

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 Replicate Key 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 through 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 1). See “Key Form Bits, ‘fff’” on page 466.

Note: A transport key is not used when the symmetric encrypted output key is in RKX key-token format.

Note: See “Number representation in trusted blocks” on page 446.

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 the following table:

Table 123. 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: X'0004' 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>
</tbody>
</table>
### Key token formats

Table 123. Source key rule reference subsection (X'0004') 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>006</td>
<td>008</td>
<td>Rule ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rule identifier for the rule that must have been used to create the source key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Rule ID is an 8-byte string of ASCII characters, left justified 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 446.

**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 the following table:

Table 124. 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0005' 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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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><strong>Example:</strong> An export key minimum length of 16 and an export key CV limit mask length of 8 returns an error.</td>
</tr>
<tr>
<td>009</td>
<td>yyy</td>
<td>Export key CV limit mask (does not exist if yyy=0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See &quot;Control-vector-base bit maps&quot; on page 463</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indicates which CV bits to check against the source key CV limit template (offset 009 + yyy).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples:</strong> A mask of X'FF' means check all bits in a byte. A mask of X'FE' ignores the parity bit in a byte.</td>
</tr>
</tbody>
</table>
Table 124. 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>
<tbody>
<tr>
<td>009 + yyy</td>
<td>yyy</td>
<td>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 &quot;Control-vector-base bit maps&quot; on page 463). 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 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.</td>
</tr>
<tr>
<td>009 + yyy + yyy</td>
<td>001</td>
<td>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.</td>
</tr>
<tr>
<td>010 + yyy + yyy</td>
<td>zzz</td>
<td>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: 1. The key label template must be 64 bytes in length 2. The first character cannot be in the range X'00' - X'1F', nor can it be X'FF' 3. The first character cannot be numeric (X'30' - X'39') 4. 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 5. The only special characters permitted are #, $, @, and * (X'23', X'24', X'40', and X'2A') 6. The wildcard X'2A'(*) is only permitted as the first character, the last character, or the only character in the template 7. Only alphanumeric characters (a through z, A through Z, 0 through 9), the four special characters (X'23', X'24', X'40', and X'2A'), and the space character (X'20') are allowed</td>
</tr>
</tbody>
</table>

Note: See "Number representation in trusted blocks" on page 446.

Trusted block section X'13'
Trusted block section X'13' contains the name (key label). 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 the following table:

<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 446.

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 the following table:

<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>X'000000000' Trusted block is in the inactive state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X'0000000001' Trusted block is in the active state</td>
</tr>
<tr>
<td>010</td>
<td>xxx</td>
<td>Information section subsections (tag-length-value objects).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>One or two objects in TLV format.</td>
</tr>
</tbody>
</table>

Note: See “Number representation in trusted blocks” on page 446.

Trusted block section X'14' subsections

Section X'14' has two information subsections (tag-length-value objects) defined. These subsections are summarized in the following table:
Table 127. 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 446.

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 the following table:

Table 128. 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></td>
<td></td>
<td>Offset</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>
<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 446.

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 the following table:

Table 129. 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: X’0000’ The coprocessor does not check dates. X’0001’ The coprocessor checks dates. Compare the activation date (offset 008) and the expiration date (offset 012) to the coprocessor’s internal real-time clock. Return an error if the coprocessor date is before the activation date or after the expiration date.</td>
</tr>
<tr>
<td>008</td>
<td>004</td>
<td>Activation date. Contains the first date that the trusted block can be used for generating or exporting keys. Format of the date is YYMD, where: YY Big-endian year (return an error if greater than 9999) MM Month (return an error if any value other than X’01’ - X’0C’) DD Day of month (return an error if any value other than X’01’ - X’1F’. Day must be valid for given month and year, including leap years). Return an error if the activation date is after the expiration date or is not valid.</td>
</tr>
<tr>
<td>012</td>
<td>004</td>
<td>Expiration date. 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 446.

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 the following table:

Table 130. 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>
</tbody>
</table>
Table 130. Trusted block application-defined data section (X’15’) (continued)

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 can 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 446.
Key token formats
Appendix C. Control vectors and changing control vectors
with the Control Vector Translate verb

This section contains a control vector table, which displays the default value of
the control vector associated with each type of key. It also describes how to change
control vectors with the Control Vector Translate verb.

Control vector table

Note: The control vectors described here are exactly the same as documented in
the IBM 4758, 4764, and TSS documents.

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 131 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</td>
<td>00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>AESTOKEN</td>
<td>00 00 00 00 00 00 00 00 00</td>
<td>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</td>
<td>00 03 71 00 03 21 00 00 00</td>
</tr>
<tr>
<td>CIPHER (double length)</td>
<td>00 03 71 00 03 41 00 00</td>
<td>00 03 71 00 03 21 00 00 00</td>
</tr>
<tr>
<td>CVARDEC</td>
<td>00 3F 42 00 03 00 00 00</td>
<td>00 03 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>CVARENC</td>
<td>00 3F 48 00 03 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2007, 2010
<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>CVARPINE</td>
<td>00 3F 41 00 03 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CVARXCVL</td>
<td>00 3F 44 00 03 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CVARXCVR</td>
<td>00 3F 47 00 03 00 00 00</td>
<td></td>
</tr>
<tr>
<td>*DATA</td>
<td>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</td>
<td>00 00 71 00 03 21 00 00</td>
</tr>
<tr>
<td>*DATAM generation key (external)</td>
<td>00 00 4D 00 03 41 00 00</td>
<td>00 00 4D 00 03 21 00 00</td>
</tr>
<tr>
<td>*DATAM key (internal)</td>
<td>00 05 4D 00 03 00 00 00</td>
<td>00 05 4D 00 03 00 00 00</td>
</tr>
<tr>
<td>*DATAMV MAC verification key (external)</td>
<td>00 00 44 00 03 41 00 00</td>
<td>00 00 44 00 03 21 00 00</td>
</tr>
<tr>
<td>*DATAMV MAC verification key (internal)</td>
<td>00 05 44 00 03 00 00 00</td>
<td>00 05 44 00 03 00 00 00</td>
</tr>
<tr>
<td>*DATA XLAT</td>
<td>00 06 71 00 03 00 00 00</td>
<td></td>
</tr>
<tr>
<td>DECIPHER</td>
<td>00 03 50 00 03 00 00 00</td>
<td></td>
</tr>
<tr>
<td>DECIPHER (double-length)</td>
<td>00 03 50 00 03 41 00 00</td>
<td>00 03 50 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYGENKY</td>
<td>00 71 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYL0</td>
<td>This control vector has the DKEYL set by default.</td>
<td></td>
</tr>
<tr>
<td>DKEYL1</td>
<td>00 72 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYL2</td>
<td>00 74 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYL3</td>
<td>00 77 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYL4</td>
<td>00 78 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYL5</td>
<td>00 7B 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYL6</td>
<td>00 7D 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>DKEYL7</td>
<td>00 7E 44 00 00 03 41 00 00</td>
<td>00 71 44 00 03 21 00 00</td>
</tr>
<tr>
<td>ENCRYPT</td>
<td>00 03 60 00 03 00 00 00</td>
<td></td>
</tr>
<tr>
<td>ENCRYPT (double-length)</td>
<td>00 03 60 00 03 41 00 00</td>
<td>00 03 60 00 03 21 00 00</td>
</tr>
<tr>
<td>*EXPORTER</td>
<td>00 41 7D 00 03 41 00 00 00</td>
<td>00 41 7D 00 03 21 00 00</td>
</tr>
<tr>
<td>IKEYXLAT</td>
<td>00 42 42 00 03 41 00 00 00</td>
<td>00 42 42 00 03 21 00 00</td>
</tr>
<tr>
<td>*IMP-PKA</td>
<td>00 42 05 00 03 41 00 00 00</td>
<td>00 42 05 00 03 21 00 00</td>
</tr>
<tr>
<td>*IMPORTER</td>
<td>00 42 7D 00 03 41 00 00 00</td>
<td>00 42 7D 00 03 21 00 00</td>
</tr>
<tr>
<td>*IPINENC</td>
<td>00 21 5F 00 03 41 00 00 00</td>
<td>00 21 5F 00 03 21 00 00</td>
</tr>
<tr>
<td>*MAC</td>
<td>00 05 4D 00 03 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>MAC (double-length)</td>
<td>00 05 4D 00 03 41 00 00</td>
<td>00 05 4D 00 03 21 00 00</td>
</tr>
<tr>
<td>*MACVER</td>
<td>00 05 44 00 03 00 00 00 00</td>
<td></td>
</tr>
<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>OKRYXLAT</td>
<td>00 41 42 00 03 41 00 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 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 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 00</td>
<td>00 22 42 00 03 21 00 00</td>
</tr>
</tbody>
</table>
Table 131. 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>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

![Control vector base bit map (common bits and key-encrypting keys)](image-url)

Figure 5. Control vector base bit map (common bits and key-encrypting keys)
**Figure 6. Control vector base bit map (data operation keys)**

<table>
<thead>
<tr>
<th>Control-Vector Base Bits</th>
<th>0 0 0 0</th>
<th>0 1 1 1</th>
<th>1 1 2 2</th>
<th>2 2 2 3</th>
<th>3 3 3 3</th>
<th>4 4 4 4</th>
<th>4 5 5 5</th>
<th>5 5 6 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 2 4 6</td>
<td>8 0 2 4</td>
<td>6 8 0 2</td>
<td>4 6 8 0</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>
</tr>
</tbody>
</table>

(Visual representation of binary bit map)

**Data Operation Keys**

- **DATA**: 00000000 00000000 00000000 00000000
- **DATAC**: 00000000 00000000 00000000 00000000
- **DATAM**: 00000000 00000000 00000000 00000000
- **DATAMV**: 00000000 00000000 00000000 00000000
- **CIPHER**: 00000000 00000000 00000000 00000000
- **DECIPHER**: 00000000 00000000 00000000 00000000
- **ENCIPHER**: 00000000 00000000 00000000 00000000
- **SECMSG**: 00000000 00000000 00000000 00000000
- **MAC**: 00000000 00000000 00000000 00000000
- **MACVER**: 00000000 00000000 00000000 00000000

(Visual representation of key forms and encryption types)

- 0000 ANY
- 0001 ANSI X9.9
- 0010 CVK KEY-A
- 0011 CVK KEY-B

---

464 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer’s Guide
Figure 7. Control vector base bit map (PIN processing keys and cryptographic variable-encrypting keys)
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:

000 Single length key

010 Double length key, left half

001 Double length key, right half

The following values could exist in some CCA implementations:

110 Double-length key, left half, halves guaranteed unique

101 Double-length key, right half, halves guaranteed unique

### Specifying a control-vector-base value

You can determine the value of a control vector by working through the following series of questions:

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 to 14) as follows:
   - The main key type bits (bits 8 to 11). Set bits 8 to 11 to one of the following values:

<table>
<thead>
<tr>
<th>Bits 8 to 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtype Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>DDATA</td>
</tr>
<tr>
<td>0010</td>
<td>DMAC</td>
</tr>
<tr>
<td>0011</td>
<td>DMV</td>
</tr>
<tr>
<td>0100</td>
<td>DIMP</td>
</tr>
<tr>
<td>0101</td>
<td>DEXP</td>
</tr>
<tr>
<td>0110</td>
<td>DPVR</td>
</tr>
<tr>
<td>1000</td>
<td>DMKEY</td>
</tr>
<tr>
<td>1001</td>
<td>DMPIN</td>
</tr>
<tr>
<td>1111</td>
<td>DALL</td>
</tr>
</tbody>
</table>

Figure 8. Control vector base bit map (key generating keys)
<table>
<thead>
<tr>
<th>Bits 8 to 11</th>
<th>Main Key Type</th>
</tr>
</thead>
<tbody>
<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 to 14). Set bits 12 to 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>
<thead>
<tr>
<th>Bits 12 to 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>
<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 to 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). Without any of the gks bits set to 1, the Key Generate
verb cannot use the associated key-encrypting key. The Key Token Build verb can set the gks bits to 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 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 1, the associated key-encrypting key can be used in the Key Translate verb.

- The key-form bits (fff, bits 40 to 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 469.

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 to 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 to 3). Set these bits to one of the following values:

     | Bits 0 to 3 | Calculation Method Keyword | Description |
     |------------|-----------------------------|-------------|
     | 0000       | NO-SPEC                     | A key with this control vector can be used with any PIN calculation method. |
     | 0001       | IBM-PIN or IBM-PINO         | A key with this control vector can be used only with the IBM PIN or PIN Offset calculation method. |
     | 0010       | VISA-PVV                    | A key with this control vector can be used only with the VISA-PVV calculation method. |
     | 0100       | GBP-PIN or GBP-PINO         | A key with this control vector can be used only with the German Banking Pool PIN or PIN Offset calculation method. |
     | 0011       | INBK-PIN                    | A key with this control vector can be used only with the Interbank PIN calculation method. |

   - The prohibit-offset bit (o, bit 37) to restrict operations to the PIN value. If set to 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 through 22 to indicate whether the key can be used with the following verbs:

<pre><code> | Service Allowed | Bit Name   | Bit |
 |-----------------|------------|-----|
 | Clear PIN Generate | CPINGEN | 18  |
 | Encrypted PIN Generate Alternate | EPINGENA** | 19 |
</code></pre>
<table>
<thead>
<tr>
<th>Service Allowed</th>
<th>Bit Name</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypted PIN Generate</td>
<td>EPINGEN</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Clear PIN Generate Alternate</td>
<td>CPINGENA</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</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 1 to permit the key to be used in the PIN Translate verb. The Control Vector Generate verb can set the TRANSLAT bit to 1 when you supply the TRANSLAT keyword.
   - Set the REFORMAT bit (r, bit 22) to 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 1 when you supply the REFORMAT keyword.

7. For the cryptographic variable-encrypting keys (bits 18 to 22), set the variable-type bits (bits 18 to 22) to one of the following values:

<table>
<thead>
<tr>
<th>Bits 18 to 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 to 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 1 if the key will be used in the secure messaging for PINs service. Set bit 19 to 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 0, the export bit prevents a key from being exported. By setting this bit to 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 0, it cannot be set to 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 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 0. The Control Vector Generate verb can set the key-part bit to 1 when you supply the KEY-PART keyword.
   - The anti-variant bits (bit 30 and bit 38). Set bit 30 to 0 and bit 38 to 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.
   - Control vector bits 64 to 127. If bits 40 to 42 are B'000' (single-length key), set bits 64 to 127 to 0. Otherwise, copy bits 0 to 63 into bits 64 to 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.
Changing control vectors with the Control Vector Translate verb

Do the following when using the Control Vector Translate 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. 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: A_1, B_1, A_2, B_2, A_3, B_3, and B_4. 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 473.) Set the A bits to the value you require for the corresponding control vector bits. In expressions 1 through 3, set the B bits to select the control vector bits to be evaluated. In expression 4, set the B bits to select the source and target control vector bits to be evaluated. Also, use the following control vector information:

- C_1 is the control vector associated with the left half of the KEK.
- C_2 is the control vector associated with the source key or selected source-key half/halves.
- C_3 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 474.
Keyword Meaning

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.

Figure 10. Control Vector Translate verb. In this figure, CHANGE-CV means the requested control vector translation change; LEFT and RIGHT mean the left and right halves of a key and its control vector.
vector to obtain the actual key value. The key value is then
ciphered 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:

- **A**₁ 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. **B**₁ is set to eight bytes of
  X'FF' so all bits of the KEK's control vector will be tested.
- **A**₂ is set to eight bytes of X'00', the (null) value of the source key control vector.
  **B**₂ is set to eight bytes of X'FF' so all bits of the source-key “control vector” will
  be tested.
- **A**₃ is set to the value of the target key's left-half control vector. **B**₃ 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.
- **B**₄ is set to eight bytes of X'00' so no comparison is made between the source
  and target control vectors.
Appendix D. PIN formats and algorithms

This appendix describes the personal identification number (PIN) formats and 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. The value ranges from 4 to 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.

\[
P1 = CLPPPPffffffffFF\\nP2 = ZZZZAAAAAA\AA\
\]

\[
\text{PIN Block} = P1 \oplus R P2
\]

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

This format is also named ECI format 4.

PIN Block = CLPPPPrrrrrrrrRR

where C = X'1'
L = X'4' to X'C'

ISO Format 2

PIN Block = CLPPPPffffffFFF

where C = X'2'
L = X'4' to X'C'

ISO Format 3

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.

The following are the formats of the intermediate PIN-block, the PAN block, and the ISO-3 PIN-block:

```
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
  3  L  P  P  P  P  P/R P/R P/R P/R P/R P/R P/R R R
```

Intermediate PIN-Block = IPB

```
  0  0  0  0  PAN  PAN  PAN  PAN  PAN  PAN  PAN  PAN  PAN  PAN  PAN  PAN
```

PAN Block

```
  3  L  P  P  P/R  P/R  P/R  P/R  P/R  P/R  P/R  P/R  P/R  R  R
```

PIN Block = IPB XOR PAN Block

Figure 11. ISO-3 PIN-block format

where:

- 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
The 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 two through ten.

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 = 11122333444555
36123456AFBECDDC : IPB
0000222334445555 : 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**

PIN Block = LPPPPzzDDDDDDDDDD

where L = X'4' to X'6'

**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 = PPPPPFPXXXXXXXX

**IBM 4700 Encrypting PINPAD Format**

This format uses the value X'F' as the delimiter for the PIN.

PIN Block = LPPPPffffffffFSS

where L = X'4' to X'C'

**IBM 3624 Format**

This format requires the program to specify the delimiter, X, for determining the PIN length.

PIN Block = PPPxxxxxxxxxxxx

**IBM 3621 Format**

This format requires the program to specify the delimiter, X, for determining the PIN length.

PIN Block = SSSSSSSPxxxxxxx

**ECI Format 2**

This format defines the PIN to be 4 digits.

PIN Block = PPPPPPPPPPPPPPPPP
PIN extraction rules

This section describes the PIN extraction rules for the Encrypted PIN Verify and Encrypted PIN Translate verbs.

Encrypted PIN Verify verb

The verb extracts the customer-entered 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, 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

The verb extracts the customer-entered 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 rule_array (if specified) indicates the PIN extraction method. Refer to the "Clear PIN Generate Alternate (CSNBCPA)" on page 295 for an explanation of PIN extraction method keywords.

Encrypted PIN Translate verb

The verb extracts the customer-entered 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.

IBM PIN algorithms

This section describes the IBM PIN generation algorithms, IBM PIN offset generation algorithm, and IBM PIN verification algorithms.

3624 PIN Generation algorithm

This algorithm generates an n-digit PIN based on account-related data or person-related data, namely the validation data. The assigned PIN length parameter specifies the length of the generated PIN.

The algorithm requires the following input parameters:

- A 64-bit validation data
- A 64-bit decimalization table
- A 4-bit assigned PIN length
- A 128-bit PIN-generation key

The service uses the PIN generation key to encipher the validation data. 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 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 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 illustrates the PIN offset generation algorithm.

If $A = 0$, then $Z = 1$; otherwise, $Z = A$.

*Figure 13. GBP PIN 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.

---

**Figure 15. PIN verification algorithm**
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 illustrates the GBP PIN verification algorithm.
CE PIN: Customer-entered PIN

Figure 16. 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.

Figure 17
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 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

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' to 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 E. Cryptographic algorithms and processes

This section provides processing details for the following aspects of the CCA design:

- "Cryptographic key-verification techniques"
- "Modification Detection Code calculation" on page 495
- "Ciphering methods" on page 497
- "MAC calculation methods" on page 504
- "RSA key-pair generation" on page 506
- "Master-key-splitting algorithm" on page 507
- "Multiple decipherment and encipherment" on page 507
- "PKA92 key format and encryption process" on page 514
- "Formatting hashes and keys in public-key cryptography" on page 516

Cryptographic key-verification techniques

The key-verification implementations described in this document employ several mechanisms for assuring the integrity and value of the key. These topics are discussed:

- Master-key verification algorithms
- CCA DES-key and key-part verification algorithm
- Encrypt zeros algorithm

Master-key verification algorithms

The CEX3C and CEX2C 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). 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 CEX3C and CEX2C 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 verb 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_{first-part} ⊕ eC(MK_{first-part})
- MKVP = MK_{second-part} ⊕ eIR(MK_{second-part})

where:
• $e_x(Y)$ is the DES encoding of $Y$ using $x$ as a key
• $\oplus$ represents the bitwise XOR function

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.

The 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 verb has 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:
• AES key tokens: 8-byte 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': 8-byte SHA-1
  – Triple-length master key, key token version X'03': 2-byte SHA-1
  – Double-length master key, key token version X'00': 8-byte z/OS
  – Double-length master key, key token version X'03': 2-byte SHA-1
• RSA key tokens:
  – Private-key section types X'06' and X'08': MDC-based
  – Private-key section types X'02' and X'05': two 20-byte SHA-1

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 (CSNBKYN)" on page 146.

The CCA DES key verification algorithm does the following:
1. Sets $KKR' = KKR \oplus RN$
2. Sets $K1 = X'4545454545454545'$
3. Sets $X1 = \text{DES encoding of } KKL \text{ using key } K1$
4. Sets $K2 = X1 \oplus KKL$
5. Sets $X2 = \text{DES encoding of } KKR' \text{ using key } K2$
6. Sets $VP = X2 \oplus KKR'$

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 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 verb.

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.

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 output into input (a digest) but very difficult to compute the output into input. MDC uses DES encryption only and a default key of $X'5252 5252 5252 5252 5252 5252 5252 5252'$.

The MDC Generate verb supports four versions of the MDC calculation method that you specify by using one of the keywords shown in Table 132 on page 496. All versions use the MDC-1 calculation.
Table 132. Versions of the MDC calculation method

<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 133.</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 133.</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' to X'10').

Use the resulting pad text in the **Table 133**. The MDC Generate verb uses these MDC calculation methods. See **MDC Generate (CSNBMDG)** on page 352 for more information.

Table 133. MDC calculation procedures

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDC-1</td>
<td>MDC-1(KD1, KD2, IN1, IN2, OUT1, OUT2);</td>
</tr>
<tr>
<td></td>
<td>Set KD1mod := set KD1 bit 1 to B'1' and bit 2 to B'0' (bits 0-7)</td>
</tr>
<tr>
<td></td>
<td>Set KD2mod := set KD2 bit 1 to B'0' and bit 2 to B'1' (bits 0-7)</td>
</tr>
<tr>
<td></td>
<td>Set F1 := IN1 XOR eKD1mod(IN1)</td>
</tr>
<tr>
<td></td>
<td>Set F2 := IN2 XOR eKD2mod(IN2)</td>
</tr>
<tr>
<td></td>
<td>Set OUT1 := (bits 0..31 of F1)</td>
</tr>
<tr>
<td></td>
<td>Set OUT2 := (bits 0..31 of F2)</td>
</tr>
<tr>
<td></td>
<td>End procedure</td>
</tr>
<tr>
<td>MDC-2</td>
<td>MDC-2(n, text, KEY1, KEY2, MDC);</td>
</tr>
<tr>
<td></td>
<td>For i := 1, 2, ..., n do</td>
</tr>
<tr>
<td></td>
<td>Call MDC-1(KEY1, KEY2, T8&lt;i&gt;, T8&lt;i&gt;, OUT1, OUT2)</td>
</tr>
<tr>
<td></td>
<td>Set KEY1 := OUT1</td>
</tr>
<tr>
<td></td>
<td>Set KEY2 := OUT2</td>
</tr>
<tr>
<td></td>
<td>End do</td>
</tr>
<tr>
<td></td>
<td>Set output MDC := (KEY1</td>
</tr>
<tr>
<td></td>
<td>End procedure</td>
</tr>
<tr>
<td>MDC-4</td>
<td>MDC-4(n, text, KEY1, KEY2, MDC);</td>
</tr>
<tr>
<td></td>
<td>For i := 1, 2, ..., n do</td>
</tr>
<tr>
<td></td>
<td>Call MDC-1(KEY1, KEY2, T8&lt;i&gt;, T8&lt;i&gt;, OUT1, OUT2)</td>
</tr>
<tr>
<td></td>
<td>Set KEY1int := OUT1</td>
</tr>
<tr>
<td></td>
<td>Set KEY2int := OUT2</td>
</tr>
<tr>
<td></td>
<td>Call MDC-1(KEY1int, KEY2int, KEY2, KEY1, OUT1, OUT2)</td>
</tr>
<tr>
<td></td>
<td>Set KEY1 := OUT1</td>
</tr>
<tr>
<td></td>
<td>Set KEY2 := OUT2</td>
</tr>
<tr>
<td></td>
<td>End do</td>
</tr>
<tr>
<td></td>
<td>Set output MDC := (KEY1</td>
</tr>
<tr>
<td></td>
<td>End procedure</td>
</tr>
</tbody>
</table>
Table 133. MDC calculation procedures (continued)

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation:</td>
<td></td>
</tr>
<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 501 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 CEX3C and CEX2C, 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.

**ANSI X3.106 Cipher Block Chaining (CBC) method**

ANSI standard X3.106 defines four modes of operation for ciphering. One of these modes, Cipher Block Chaining (CBC), defines the basic method for ciphering multiple 8-byte data strings. Figure 19 on page 499 and Figure 20 on page 499 show CBC using the Encipher and the 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 through 7 of the chaining_vector variable.

An ICV is XORed with the first block of eight bytes. When you call the Encipher verb or the 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.

*Figure 18. Triple-DES data encryption and decryption*
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 one through eight 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' through X'08'. The standard defines that any other added bytes, or pad characters, be random.

Figure 19. Enciphering using the ANSI X3.106 CBC method

Figure 20. Deciphering using the CBC method

ANSI X9.23
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 one through eight 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' through 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 one through eight 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 501.

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.

![Figure 21. Enciphering using the ANSI X9.23 method](image)
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 502 and Figure 24 on page 502. This is often referred to as “outer CBC mode.”

This CCA supports double-length DES keys for triple-DES data encryption using the Decipher and Encipher 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 503 and Figure 26 on page 504. K1, K2, and K3 are true keys while “K4” and “K5” are initialization vectors. See Figure 25 on page 503 and Figure 26 on page 504.
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:
- ANSI X9.9
- ANSI X9.19 Optional Procedure 1
- EMV post-padding of X'80'
- ISO 16609 TDES MAC

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. This process is called the message authentication code (MAC) calculation method.  

Figure 27 on page 505 shows the MAC calculation for binary data. In this figure, KEY is a 64-bit key, and T<sub>1</sub> through T<sub>n</sub> are 64-bit data blocks of text. If T<sub>n</sub> is less
than 64 bits long, binary zeros are appended to the right of $T_n$. Data blocks $T_1...T_n$ are DES CBC-encrypted with all output discarded except for the final output block, $O_n$.

**ANSI X9.19 Optional Procedure 1 MAC**

The Financial Institution (Retail) Message Authentication Standard, ANSI X9.19 Optional Procedure 1 (ISO/IEC 9797-1, Algorithm 3) specifies additional processing of the 64-bit $O_n$ MAC value. 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 X9.9 and X9.19 Optional Procedure 1 standards, the leftmost 32 bits (4 bytes) of $O_n$ 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.

![Figure 27. MAC calculation method](image)

**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. 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.

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 = \(((\text{modulus bit length} + 1)/2)\)
- prime \( q \) bit length = \(\text{modulus bit length} - p\_\text{bit length}\)
- \( p \) 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, \(|p-q| > 2^{(\text{prime bit length} - 100)}\). 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 \(|p-q| > 2^{412}\).

1. For each key generation, and for any size of key, the PKA manager seeds an internal FIPS-approved, SHA-1 based psuedo 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.
Master-key-splitting algorithm

This section describes the mathematical and cryptographic basis for the $m$-of-$n$ key shares scheme.

The key splitting is based on Shamir's secret sharing algorithm:

The value to be shared is the master key, $K_m$, which is a triple-DES key and 168 bits long. Let $P$ be the first prime number larger than $2^{168}$. All operations are carried out modulo $P$.

Shamir's secret sharing allows the sharing of $K_m$ among $n$ trustees in a way that no set of $t$ or less of trustees has any information about $K_m$, while $t+1$ trustees can reconstruct $K_m$.

Sharing phase:

1. Randomly choose $a_t,...,a_1$ in $[0..P-1]$.
2. Consider the polynomial $f(x) = a_t x^t + ... + a_1 x + a_0$, where $a_0=K_m$.
   Compute $mk_i = f(i) \mod P$ for all $i = 1,...,n$.
3. Proceed to distribute the values $mk_i$ as described above.

Reconstruction phase:

After generating the set of authentic values proceed as follows:

1. Take $t+1$ values and interpolate the polynomial $f(x)$ of degree $t$. Pass through these values using Lagrange interpolation. This defines a polynomial $f(x)$ such that: $f(i)=mk_i$, and further more $f(0) = MK$. Use the mathematical formula to reconstruct the free term of the polynomial $f(x)$. Let $k_1,...,k_{t+1}$ be the indices of the $mk_i$'s used for reconstruction. Then
   
   $a_0 = \text{SUM}_j(b_{k_j}) \text{PROD}_h (x_{(k_h)} / (x_{(k_h)} - x_{(k_j)})) \mod P$

2. Install key $K_m = a_0 = f(0) \mod P$.

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:

- \( eK(x) \), where \( x \) is enciphered under \( K \)
- \( dK(y) \) represents plaintext, where \( K \) is the key and \( y \) is the ciphertext

Therefore, \( dK(eK(x)) \) 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**

The multiple encipherment of a single-length key (\( K \)) using a double-length *KEK is defined as follows:

\[
e^*KEK(K) = eKEKL(dKEKR(eKEKL(K)))
\]

where KEKL is the left 64 bits of *KEK and KEKR is the right 64 bits of *KEK.

![Figure 28](image_url) illustrates the definition.
Multiple decipherment of single-length keys

The multiple encipherment of an encrypted single-length key (\( Y = e^*\text{KEK}(K) \)) using a double-length *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 KEKL is the left 64 bits of *KEK and KEKR is the right 64 bits of *KEK.

Figure 29 illustrates the definition.
Multiple encipherment of double-length keys

The multiple encipherment of a double-length key (*K) using two double-length *KEKs, *KEKa, and *KEKb is defined as follows:

\[ e^{*KEKa}(KL) \| e^{*KEKb}(KR) = eKEKaL(dKEKaR(eKEKaL(KL))) \| eKEKbL(dKEKbR(eKEKbL(KR))) \]

where:

- KL is the left 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 30 illustrates the definition.
The multiple decipherment of an encrypted double-length key, \( Y = e^{KEK_a}(KL) \| e^{KEK_b}(KR) \), using two double-length \( KEK_a \) and \( KEK_b \), is defined as follows:

\[
D^{KEK_a}(YL) \| d^{KEK_b}(YR) = d^{KEK_a}(e^{KEK_a}(KL)) \| d^{KEK_b}(e^{KEK_b}(KR))
\]

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 illustrates the definition.
Multiple encipherment of triple-length keys

The multiple encipherment of a triple-length key (**K) using two double-length *KEKs, *KEKa, and *KEKb is defined as follows:

\[
e^{*\text{KEKa}}(\text{KL}) || e^{*\text{KEKb}}(\text{KM}) || e^{*\text{KEKa}}(\text{KR}) = \\
e^{\text{KEKa}}(d^{\text{KEKa}}(e^{\text{KEKa}}(\text{KL}))) || \\
e^{\text{KEKb}}(d^{\text{KEKb}}(e^{\text{KEKb}}(\text{KM}))) || \\
e^{\text{KEKa}}(d^{\text{KEKa}}(e^{\text{KEKa}}(\text{KR})))
\]

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 32 on page 513 illustrates the definition.
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:

\[
\begin{align*}
\text{d*KEKa}(Y_L) & \quad \text{d*KEKb}(Y_M) \quad \text{d*KEKa}(Y_R) \\
& = \text{dKEKaL}(\text{eKEKaR}(\text{dKEKaL}(Y_L))) \\
& \quad \text{dKEKbL}(\text{eKEKbR}(\text{dKEKbL}(Y_M))) \\
& \quad \text{dKEKaL}(\text{eKEKaR}(\text{dKEKaL}(Y_R))) \\
& = \text{dKEKa}(e*KEKa(KL)) \\
& \quad \text{dKEKb}(e*KEKb(KM)) \\
& \quad \text{dKEKa}(e*KEKa(KR)) \\
& = **K
\end{align*}
\]

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 on page 514 illustrates the definition.
PKA92 key format and encryption process

The PKA Symmetric Key Generate and the PKA 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 134 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 134. 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>
<tr>
<td>021</td>
<td>008</td>
<td>Control vector base for the DES key.</td>
</tr>
</tbody>
</table>
Table 134. 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>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 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 134 on page 514

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 to 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 to 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 discusses 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:

---

5. PKCS standards can be retrieved from http://www.rsasecurity.com/rsalabs/pkcs.
For formatting keys for secured transport (CSNDSYX, CSNDSYG, CSNDSYI):

- RSAES-OAEP, the preferred method for key-encipherment when exchanging DATA keys between systems. Keyword PKCSOAEP 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. Keyword PKCS-1.2 is used to invoke this formatting technique.

For formatting hashes for digital signatures (CSNDDSG and CSNDDSV):

- 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 discusses 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):

\[ \text{X'00'} \| \text{BT} \| \text{PS} \| \text{X'00'} \| \text{D} \]

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, and is bytes of X'00' 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 BER encoding of an MD5 or SHA-1 value by prepending these strings to the 16-byte or 20-byte hash values, respectively:

**MD5**

\[ \text{X'3020300C 06082A86 4886F70D 02050500 0410'} \]

**SHA-1**

\[ \text{X'30213009 0605280E 03021A05 000414'} \]

---

6. 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 F. Access control points and verbs

**Important:** By default, you should disable commands. Do not enable an access control point unless you know why you are enabling it.

For instructions on how to enable and disable these access control points 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 access control points (current and new) are enabled in the DEFAULT role with the appropriate licensed internal code on the CEX3C.

Note that each domain in the CEX3C (with hardware enforced access permissions) starts out with its own DEFAULT role with the default access control point values as shown. However, it is possible to use the TKE to change access control point values in the DEFAULT role or to define other roles. 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 135 lists the CCA access control commands (access control points, ACPs). The role to which a user is assigned determines the commands available to that user. The table includes the following columns:

- **Offset**: The hexadecimal offset, or access-control-point code, for the command. Offsets between X'0000' and X'FFFF' not listed in this table are reserved.
- **Command name**: The name of the command required by the following verbs. These command names are defined in the csuap.def file used by the CNM utility.
- **Verb name**: The names of the verbs that require that command to be enabled; for example, the Encipher (CSNBENC) verb fails without permission to use the Encipher command.
- **Entry**: The entry-point name of the verb.
- **Usage**: Usage recommendations for the command. The abbreviations in this column are explained at the end of the table.

These access control points are OFF by default:

- X'0020'
- X'0033'
- X'0061'
- X'008A'
- X'010E'
- X'0120'
- X'0231'
- X'0232'
- X'0236'
- X'0290'
- X'030C'
- X'0313'

All others are ON by default.

See the **Restrictions**, **Required commands**, or **Usage notes** sections at the end of each verb description for access control information.

### Table 135. Access Control Points and corresponding CCA commands

<table>
<thead>
<tr>
<th>Offset</th>
<th>Command name</th>
<th>Verb name</th>
<th>Entry</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'000E'</td>
<td>Encipher</td>
<td>Encipher</td>
<td>CSNBENC</td>
<td>O</td>
</tr>
<tr>
<td>Offset</td>
<td>Command name</td>
<td>Verb name</td>
<td>Entry</td>
<td>Usage</td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>X'000F'</td>
<td>Decipher</td>
<td>Decipher</td>
<td>CSNBDEC</td>
<td>O</td>
</tr>
<tr>
<td>X'0010'</td>
<td>Generate MAC</td>
<td>MAC Generate</td>
<td>CSNBMGN</td>
<td>O</td>
</tr>
<tr>
<td>X'0011'</td>
<td>Verify MAC</td>
<td>MAC Verify</td>
<td>CSNBMVR</td>
<td>O</td>
</tr>
<tr>
<td>X'0012'</td>
<td>Reencipher to Master Key</td>
<td>Key Import</td>
<td>CSNBKIM</td>
<td>O</td>
</tr>
<tr>
<td>X'0013'</td>
<td>Reencipher from Master Key</td>
<td>Key Export</td>
<td>CSNBKEK</td>
<td>O</td>
</tr>
<tr>
<td>X'0018'</td>
<td>Load First Master Key Part</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0019'</td>
<td>Combine Master Key Parts</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'001A'</td>
<td>Set Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'001B'</td>
<td>Load First Key Part</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'001C'</td>
<td>Combine Key Parts</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'001D'</td>
<td>Compute Verification Pattern</td>
<td>Key Test</td>
<td>CSNBKTYT</td>
<td>R</td>
</tr>
<tr>
<td>X'001E'</td>
<td>Translate Key</td>
<td>Key Translate</td>
<td>CSNBKTR</td>
<td>O</td>
</tr>
<tr>
<td>X'0020'</td>
<td>Generate Random Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0022'</td>
<td>Clear New Master Key Register</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0023'</td>
<td>Clear Old Master Key Register</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0040'</td>
<td>Generate Diversified Key (CLR8-ENC)</td>
<td>Diversified Key Generate</td>
<td>CSNBKDG</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0041'</td>
<td>Generate Diversified Key (TDES-ENC)</td>
<td>Diversified Key Generate</td>
<td>CSNBKDG</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0042'</td>
<td>Generate Diversified Key (TDES-DEC)</td>
<td>Diversified Key Generate</td>
<td>CSNBKDG</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0043'</td>
<td>Generate Diversified Key (SESS-XOR)</td>
<td>Diversified Key Generate</td>
<td>CSNBKDG</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0044'</td>
<td>Enable DKG Single Length Keys and Equal Halves for TDES-ENC, TDES-DEC</td>
<td>Diversified Key Generate</td>
<td>CSNBKDG</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0045'</td>
<td>Generate Diversified Key (TDES-XOR)</td>
<td>Diversified Key Generate</td>
<td>CSNBKDG</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0046'</td>
<td>Generate Diversified Key (TDESEMV)</td>
<td>Diversified Key Generate</td>
<td>CSNBKDG</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0053'</td>
<td>Load First Asymmetric Master Key Part</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0054'</td>
<td>Combine Asymmetric Master Key Parts</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0057'</td>
<td>Set Asymmetric Master Key</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0060'</td>
<td>Clear New Asymmetric Master Key Buffer</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0061'</td>
<td>Clear Old Asymmetric Master Key Buffer</td>
<td>Master Key Process</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'008A'</td>
<td>Generate MDC</td>
<td>MDC Generate</td>
<td>CSNBMGD</td>
<td>R</td>
</tr>
<tr>
<td>X'008C'</td>
<td>Generate Key Set</td>
<td>Key Generate</td>
<td>CSNBKGN</td>
<td>O</td>
</tr>
<tr>
<td>X'008E'</td>
<td>Generate Key</td>
<td>Key Generate</td>
<td>CSNBKGN</td>
<td>O</td>
</tr>
<tr>
<td>X'0090'</td>
<td>Re-encipher to Current Master Key</td>
<td>Key Token Change</td>
<td>CSNBKTC</td>
<td>R</td>
</tr>
<tr>
<td>X'00A0'</td>
<td>Generate Clear 3624 PIN</td>
<td>Clear PIN Generate</td>
<td>CSNBPGN</td>
<td>O</td>
</tr>
<tr>
<td>X'00A4'</td>
<td>Generate Clear 3624 PIN Offset</td>
<td>Clear PIN Generate Alternate</td>
<td>CSNBPVA</td>
<td>O</td>
</tr>
<tr>
<td>X'00AB'</td>
<td>Verify Encrypted 3624 PIN</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVY</td>
<td>O</td>
</tr>
<tr>
<td>X'00AC'</td>
<td>Verify Encrypted German Bank Pool PIN</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVY</td>
<td>O</td>
</tr>
<tr>
<td>X'00AD'</td>
<td>Verify Encrypted Visa PVV</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVY</td>
<td>O</td>
</tr>
<tr>
<td>X'00AE'</td>
<td>Verify Encrypted InterBank PIN</td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVY</td>
<td>O</td>
</tr>
<tr>
<td>Offset</td>
<td>Command name</td>
<td>Verb name</td>
<td>Entry</td>
<td>Usage</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------</td>
<td>------------------------------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>X'00AF'</td>
<td>Format and Encrypt PIN</td>
<td>Clear PIN Encrypt</td>
<td>CSNBCPE</td>
<td>O</td>
</tr>
<tr>
<td>X'00B0'</td>
<td>Generate Formatted and Encrypted 3624 PIN</td>
<td>Encrypted PIN Generate†</td>
<td>CSNBEPG</td>
<td>O</td>
</tr>
<tr>
<td>X'00B1'</td>
<td>Generate Formatted and Encrypted German Bank Pool PIN</td>
<td>Encrypted PIN Generate†</td>
<td>CSNBEPG</td>
<td>O</td>
</tr>
<tr>
<td>X'00B2'</td>
<td>Generate Formatted and Encrypted InterBank PIN</td>
<td>Encrypted PIN Generate†</td>
<td>CSNBEPG</td>
<td>O</td>
</tr>
<tr>
<td>X'00B3'</td>
<td>Translate PIN with No Format-Control to No Format-Control</td>
<td>Encrypted PIN Translate†</td>
<td>CSNBPTR</td>
<td>O</td>
</tr>
<tr>
<td>X'00B7'</td>
<td>Reformat PIN with No Format-Control to No Format-Control</td>
<td>Encrypted PIN Translate†</td>
<td>CSNBPTR</td>
<td>O</td>
</tr>
<tr>
<td>X'00BB'</td>
<td>Generate Clear Visa PVV Alternate</td>
<td>Clear PIN Generate Alternate†</td>
<td>CSNBCPA</td>
<td>O</td>
</tr>
<tr>
<td>X'00BC'</td>
<td>Generate PIN Change Using OPINENC</td>
<td>PIN Change/Unblock†</td>
<td>CSNBPCU</td>
<td>O</td>
</tr>
<tr>
<td>X'00BD'</td>
<td>Generate PIN Change Using IPINENC</td>
<td>PIN Change/Unblock†</td>
<td>CSNBPCU</td>
<td>O</td>
</tr>
<tr>
<td>X'00C3'</td>
<td>Encrypt Under Master Key</td>
<td>Clear Key Import</td>
<td>CSNBCKI</td>
<td>SC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiple Clear Key Import</td>
<td>CSNBCKM</td>
<td></td>
</tr>
<tr>
<td>X'00CD'</td>
<td>Lower Export Authority</td>
<td>Prohibit Export</td>
<td>CSNBPEX</td>
<td>O</td>
</tr>
<tr>
<td>X'00D6'</td>
<td>Translate Control Vector</td>
<td>Control Vector Translate</td>
<td>CSNBCTV</td>
<td>SC</td>
</tr>
<tr>
<td>X'00D7'</td>
<td>Generate Key Set Extended</td>
<td>Key Generate†</td>
<td>CSNBKGN</td>
<td>SC, SUP</td>
</tr>
<tr>
<td>X'00DA'</td>
<td>Encrypt Cryptovariable</td>
<td>Cryptographic Variable Encryptor</td>
<td>CSNBCVE</td>
<td>NRP, SUP</td>
</tr>
<tr>
<td>X'00DB'</td>
<td>Replicate Key</td>
<td>Key Generate†</td>
<td>CSNBKGN</td>
<td>NR, SUP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remote Key Export†</td>
<td>CSNDRKX</td>
<td></td>
</tr>
<tr>
<td>X'00DF'</td>
<td>Verify CVV</td>
<td>CVV Generate</td>
<td>CSNBCSG</td>
<td>O</td>
</tr>
<tr>
<td>X'00E0'</td>
<td>Verify CVV</td>
<td>CVV Verify</td>
<td>CSNBCSV</td>
<td>O</td>
</tr>
<tr>
<td>X'00E1'</td>
<td>Unique Key Per Transaction, ANSI X9.24</td>
<td>Encrypted PIN Translate†</td>
<td>CSNBPTR</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encrypted PIN Verify†</td>
<td>CSNBVPVR</td>
<td></td>
</tr>
<tr>
<td>X'0100'</td>
<td>PKA96 Digital Signature Generate</td>
<td>Digital Signature Generate</td>
<td>CSNDDS</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0101'</td>
<td>PKA96 Digital Signature Verify</td>
<td>Digital Signature Verify</td>
<td>CSNDDS</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0102'</td>
<td>PKA96 Key Token Change</td>
<td>PKA Key Token Change</td>
<td>CSNDKTC</td>
<td>O</td>
</tr>
<tr>
<td>X'0103'</td>
<td>PKA96 PKA Key Generate</td>
<td>PKA Key Generate†</td>
<td>CSNDPKG</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'0104'</td>
<td>PKA96 PKA Key Import</td>
<td>PKA Key Import</td>
<td>CSNPKI</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'0105'</td>
<td>Symmetric Key Export PKCS-1.2/OAEP</td>
<td>PKA Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>SC</td>
</tr>
<tr>
<td>X'0106'</td>
<td>Symmetric Key Import PKCS-1.2/OAEP</td>
<td>PKA Symmetric Key Import†</td>
<td>CSNDSY</td>
<td>O</td>
</tr>
<tr>
<td>X'0109'</td>
<td>Data Key Import</td>
<td>Data Key Import</td>
<td>CSNBDKM</td>
<td>O</td>
</tr>
<tr>
<td>X'010A'</td>
<td>Data Key Export</td>
<td>Data Key Export</td>
<td>CSNBDKX</td>
<td>O</td>
</tr>
<tr>
<td>X'010D'</td>
<td>PKA92 Symmetric Key Generate</td>
<td>PKA Symmetric Key Generate†</td>
<td>CSNDSYG</td>
<td>SC</td>
</tr>
<tr>
<td>X'011E'</td>
<td>PKA Encipher Clear Key</td>
<td>PKA Encrypt</td>
<td>CSNDPKE</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'011F'</td>
<td>PKA Decipher Key Data</td>
<td>PKA Decrypt</td>
<td>CSNDPKD</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0120'</td>
<td>Generate Random Asymmetric Master Key</td>
<td>Master Key Process†</td>
<td>CSNBMKP</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0124'</td>
<td>Clear AES NMK Register (CLEAR)</td>
<td>Master Key Process†</td>
<td>CSNBMKP</td>
<td>O, SUP (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0125'</td>
<td>Load First AES Master Key Part</td>
<td>Master Key Process†</td>
<td>CSNBMKP</td>
<td>O, SUP (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0126'</td>
<td>Combine AES Master Key Parts</td>
<td>Master Key Process†</td>
<td>CSNBMKP</td>
<td>O, SUP (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0128'</td>
<td>Activate New AES Master Key (SET)</td>
<td>Master Key Process†</td>
<td>CSNBMKP</td>
<td>O, SUP (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0129'</td>
<td>Encipher Under AES Master Key</td>
<td>Multiple Clear Key Import</td>
<td>CSNBCKM</td>
<td>SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'012A'</td>
<td>Encipher Data Using AES</td>
<td>Symmetric Algorithm Encipher†</td>
<td>CSNBSAE</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'012B'</td>
<td>Decipher Data Using AES</td>
<td>Symmetric Algorithm Decipher†</td>
<td>CSNBSAD</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'012C'</td>
<td>Generate AES DATA Key (PKCSOAEP or PKCS-1.2)</td>
<td>PKA Symmetric Key Generate</td>
<td>CSNDSYG</td>
<td>SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'012D'</td>
<td>Generate AES DATA Key (ZERO-PAD)</td>
<td>PKA Symmetric Key Generate</td>
<td>CSNDSYG</td>
<td>SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>Offset</td>
<td>Command name</td>
<td>Verb name</td>
<td>Entry</td>
<td>Usage</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
<td>--------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>X'012E'</td>
<td>Import AES Key (PKCSOAEP or PKCS-1.2)</td>
<td>PKA Symmetric Key Import</td>
<td>CSNDSYI</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'012F'</td>
<td>Import AES Key (ZERO-PAD)</td>
<td>PKA Symmetric Key Import</td>
<td>CSNDSYI</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0130'</td>
<td>Export AES Key (PKCSOAEP or PKCS-1.2)</td>
<td>PKA Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0131'</td>
<td>Export AES Key (ZERO-PAD)</td>
<td>PKA Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0203'</td>
<td>Delete Retained Key</td>
<td>Retained Key Delete</td>
<td>CSNDRKD</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0205'</td>
<td>PKA Clear Key Generate</td>
<td>PKA Key Generate†</td>
<td>CSNDPKG</td>
<td>O, SUP</td>
</tr>
<tr>
<td>X'0230'</td>
<td>List Retained Key</td>
<td>Retained Key List</td>
<td>CSNDRKL</td>
<td>O</td>
</tr>
<tr>
<td>X'0235'</td>
<td>PKA92 Symmetric Key Import</td>
<td>PKA Symmetric Key Import†</td>
<td>CSNDSYI</td>
<td>O</td>
</tr>
<tr>
<td>X'0236'</td>
<td>PKA92 Symmetric Key Import with PIN Keys</td>
<td>PKA Symmetric Key Import†</td>
<td>CSNDSYI</td>
<td>O</td>
</tr>
<tr>
<td>X'023C'</td>
<td>ZERO-PAD Symmetric Key Generate</td>
<td>PKA Symmetric Key Generate†</td>
<td>CSNDSYG</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'023D'</td>
<td>ZERO-PAD Symmetric Key Import</td>
<td>PKA Symmetric Key Generate†</td>
<td>CSNDSYI</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'023E'</td>
<td>ZERO-PAD Symmetric Key Export</td>
<td>PKA Symmetric Key Export</td>
<td>CSNDSYX</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'023F'</td>
<td>Symmetric Key Generate PKCS-1.2/OAEP</td>
<td>PKA Symmetric Key Generate†</td>
<td>CSNDSYG</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0273'</td>
<td>Secure Messaging for Keys</td>
<td>Secure Messaging for Keys</td>
<td>CSNBSKY</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>O</td>
</tr>
<tr>
<td>X'0276'</td>
<td>Unrestrict Reencipher from Master Key</td>
<td>Key Export</td>
<td>CSNBKEX</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0277'</td>
<td>Unrestrict Data Key Export</td>
<td>Data Key Export</td>
<td>CSNBDKX</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0278'</td>
<td>Add Key Part</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'0279'</td>
<td>Complete Key Part</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>SC, SEL</td>
</tr>
<tr>
<td>X'027A'</td>
<td>Unrestrict Combine Key Parts</td>
<td>Key Part Import</td>
<td>CSNBKPI</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'027B'</td>
<td>Unrestrict Reencipher to Master Key</td>
<td>Key Import</td>
<td>CSNBKIM</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'027C'</td>
<td>Unrestrict Data Key Import</td>
<td>Data Key Import</td>
<td>CSNBDKM</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'027D'</td>
<td>Permit Regeneration Data</td>
<td>PKA Key Generate‡</td>
<td>CSNDPKG</td>
<td>O, NRP, SC</td>
</tr>
<tr>
<td>X'027E'</td>
<td>Permit Regeneration Data for Retained Keys</td>
<td>PKA Key Generate‡</td>
<td>CSNDPKG</td>
<td>O, NRP, SC</td>
</tr>
<tr>
<td>X'0290'</td>
<td>Generate Diversified Key (DALL with DKYGENKY Key Type)</td>
<td>Diversified Key Generate‡</td>
<td>CSNBDKG</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'0291'</td>
<td>Generate CSC 3, 4 and 5 Values</td>
<td>Transaction Validation‡</td>
<td>CSNBTIV</td>
<td>O, SEL</td>
</tr>
<tr>
<td>X'0292'</td>
<td>Verify CSC 3 Values</td>
<td>Transaction Validation‡</td>
<td>CSNBTIV</td>
<td>O</td>
</tr>
<tr>
<td>X'0293'</td>
<td>Verify CSC 4 Values</td>
<td>Transaction Validation‡</td>
<td>CSNBTIV</td>
<td>O</td>
</tr>
<tr>
<td>X'0294'</td>
<td>Verify CSC 5 Values</td>
<td>Transaction Validation‡</td>
<td>CSNBTIV</td>
<td>O</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>O</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>O</td>
</tr>
<tr>
<td>X'0301'</td>
<td>Lower Export Authority, Extended</td>
<td>Prohibit Export Extended</td>
<td>CSNBPEXX</td>
<td>O</td>
</tr>
<tr>
<td>X'030C'</td>
<td>Override DSG ZERO-PAD Length Restriction</td>
<td>Digital Signature Generate</td>
<td>CSNDDSX</td>
<td>O, SC</td>
</tr>
<tr>
<td>X'030F'</td>
<td>Create a Trusted Block in Inactive Form</td>
<td>Trusted Block Create</td>
<td>CSNDTBC</td>
<td>O, SUP (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0310'</td>
<td>Activate an Inactive Trusted Block</td>
<td>Trusted Block Create</td>
<td>CSNDTBC</td>
<td>O, SUP (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0311'</td>
<td>Convert Trusted Block from External to Internal Form</td>
<td>PKA Key Import</td>
<td>CSNDRKI</td>
<td>O, SEL (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0312'</td>
<td>Generate or Export a Key for Use by a Non-CCA Node</td>
<td>Remote Key Export</td>
<td>CSNDRKX</td>
<td>O, SEL (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0313'</td>
<td>PTR Enhanced PIN Security Mode</td>
<td>Clear PIN Generate Alternate</td>
<td>CSNBCPA</td>
<td>O, SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clear PIN Encrypt</td>
<td>CSNBCPE</td>
<td>O, SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encrypted PIN Generate</td>
<td>CSNBEPG</td>
<td>O, SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encrypted PIN Translate</td>
<td>CSNPTR</td>
<td>O, SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encrypted PIN Verify</td>
<td>CSNBPVR</td>
<td>O, SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIN Change/Unblock</td>
<td>CSNBPCU</td>
<td>O, SC (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'0318'</td>
<td>Translate from CCA RSA to SC VISA Format</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
</tbody>
</table>
Table 135. Access Control Points and corresponding CCA commands (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Command name</th>
<th>Verb name</th>
<th>Entry</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0319'</td>
<td>Translate from CCA RSA to SC ME Format</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'031A'</td>
<td>Translate from CCA RSA to SC CRT Format</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'031B'</td>
<td>XLATE from Encryption Under Source EXP KEK to Target EXP KEK</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'031C'</td>
<td>XLATE from Encryption Under Source IMP KEK to Target EXP KEK</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>O (Rel. 4.0 or later)</td>
</tr>
<tr>
<td>X'031D'</td>
<td>XLATE from Encryption Under Source IMP KEK to Target IMP KEK</td>
<td>PKA Key Translate</td>
<td>CSNDPKT</td>
<td>O (Rel. 4.0 or later)</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.
- †: 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.
- ‡: 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 CEX3C. 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 workstation in order to see supported CEX3C 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 CEX3C are controlled through access control points. Most of these are enabled in the DEFAULT role.

New TKE users and non-TKE users have almost all access control points enabled.

Note: Access control points DKYGENKY-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.

For CCA Version 4.0.0, these are new access control points for these verbs:
- Cryptographic Resource Allocate and Cryptographic Resource Deallocate - Enables translating DES keys for use with the CPACF (offset X'295')
- Cryptographic Resource Allocate and Cryptographic Resource Deallocate - Enables translating AES keys for use with the CPACF (offset X'296')
- PKA Key Translate. See [Usage notes of "PKA Key Translate (CSNDPKT)" on page 380 for the type of transport keys that can be used.
- PKA Symmetric Key Export - Export AES Key (PKCSOAEP or PKCS-1.2) (offset X'0130')
- PKA Symmetric Key Export - Export AES Key (ZERO-PAD) (offset X'0131')
- PKA Symmetric Key Generate - Generate AES DATA Key (PKCSOAEP or PKCS-1.2) (offset X'012C')
- PKA Symmetric Key Generate - Generate AES DATA Key (ZERO-PAD) (offset X'012D')
- PKA Symmetric Key Import - Import AES Key (PKCSOAEP or PKCS-1.2) (offset X'012E')
- PKA Symmetric Key Import - Import AES Key (ZERO-PAD) (offset X'012F')
- Symmetric Algorithm Decipher - Decipher Data Using AES (offset X'012B')
- Symmetric Algorithm Encipher - Encipher Data Using AES (offset X'012A')
Appendix G. Sample verb call routines

**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 on page 526 and Figure 35 on page 531 refer to the chapters in this book for descriptions of the called verbs and their parameters. These verbs are listed in Table 136.

<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/CEX3C/samples directory. For reference, a copy of the sample routine is shown in Figure 34 on page 526.
Figure 34. Syntax, sample routine in C (Part 1 of 4)
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.

```c
#include <studio.h>
#include <string.h>
#ifdef _AIX
  #include <csufincl.h>
#elif __WINDOWS__
  #include "csunincl.h"
#else
  #include "csulincl.h" /* else linux */
#endif

#define KEY_FORM "OPOP"
#define KEY_LENGTH "SINGLE"
#define KEY_TYPE_1 "MAC"
#define KEY_TYPE_2 "MACVER"
#define INPUT_TEXT "abcdefhijklmnop0987654321"
#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");

  Figure 34. Syntax, sample routine in C (Part 2 of 4)
```
/* 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 */
CSNBKGN(&return_code,
&reason_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 */
    printf("return_code = %ld, ", return_code); /* Print return code */
    printf("reason_code = %ld.
", reason_code); /* Print reason code */
    return;
} else
    printf("Key_Generate successful.
");

/* 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, '\x00', 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 */

Figure 34. Syntax, sample routine in C (Part 3 of 4)
/* 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.
", reason_code); /* Print reason code */
    return;
} else {
    printf ("MAC_Generate successful.
" );
    printf ("MAC_value = %s
", 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, '\0', sizeof(rule_array));/* Clear rule array */
memcpy (rule_array[0], MAC_LENGTH, 8); /* Rule array element */
    /* (use default Ciphering */
    /* Method 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, /* Same as for MAC_Generate */
    text,
    &rule_array_count,
    &rule_array[0][0],
    chaining_vector,
    mac_value); /* Output from MAC_Generate */

/* Check the return/reason codes. Terminate if there is an error. */
if (return_code != 0 || reason_code != 0) {
    printf ("MAC_Verify failed: "); /* Print failing verb */
    printf ("return_code = %ld, ", return_code); /* Print return code */
    printf ("reason_code = %ld.
", reason_code); /* Print reason code */
    return;
} else /* No error occurred */
    printf ("MAC_Verify successful.
");

Figure 34. Syntax, sample routine in C (Part 4 of 4)
Sample program in Java

Before running this program, review the information about the JNI interface in "Building Java applications to use with the CCA JNI" on page 18.

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 531. 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:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Default distribution location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novell SUSE Linux</td>
<td>/opt/IBM/CEX3C/samples</td>
</tr>
<tr>
<td>Red Hat Linux</td>
<td>/opt/IBM/CEX3C/samples</td>
</tr>
</tbody>
</table>

Invoke the following command from the directory that contains the sample source code to compile the program:

```bash
javac -classpath /opt/IBM/CEX3C/cnm/HIKM.zip mac.java
```

Notes:
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:

```bash
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64
/opt/ibm/java-1386-60/jre/bin/java -classpath /opt/IBM/CEX3C/cnm/HIKM.zip:. mac
```

For a SUSE Linux system:

```bash
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64
java -classpath /opt/IBM/CEX3C/cnm/HIKM.zip:. mac
```

Notes:
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 (Part 1 of 4)
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 = "abcdefhgijklmnopqrstuvwx";
    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.");

            Figure 35. Syntax, sample routine in Java (Part 2 of 4)
/* 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,
Byte_key_length,
Byte_mac_key_type,
Byte_macver_key_type,
Byte_kek_key_id_1,
Byte_kek_key_id_2,
Byte_mac_key_id,
Byte_macver_key_id);

if (0 != IntReturncode.getValue() || 0 != IntReasoncode.getValue())
{
    System.out.println ("Key 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_text = 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);

Figure 35. Syntax, sample routine in Java (Part 3 of 4)
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 ("\nMAC_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.");
}
}

/* Print ending banner */
System.out.println("\nCryptographic Coprocessor Support Program JAVA example program finished.\n");
//end main
//end mac class

Figure 35. Syntax, sample routine in Java (Part 4 of 4)
Appendix H. 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 RPM is: csulcca-4.0.0-10.s390x.rpm.

In order to use the full set of CCA V 4.0.0 functions, a CEX3C feature is required. This feature is available with IBM System z10 model GA3 and higher models. A limited set of CCA V 4.0.0 functions can be used with a CEX2C feature.

Consult the README.linz file in the /opt/IBM/CEX3C/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 ‘zcrypt’ is already included in the regular kernel package shipped with your Linux distribution. The cryptographic device driver is provided as a single kernel module named z90crypt. For information on how to load and configure the cryptographic device driver refer to the documentation provided with your Linux distribution, and Device Drivers, Features, and Commands, SC33-8411. Because there are some versions of this book that are for a specific Linux kernel, make sure that you refer to the book that goes with your specific Linux kernel level.

zcrypt device driver usage

The zcrypt driver supports kernel parameter:

```
domain
```

An integer argument that sets the domain index for AP devices. If not set or set to -1, the domain index with the maximum number of devices will be used. You have to make sure that the selected domain contains at least one CEX3C adapter.

The `poll_thread` parameter is no longer used. If you are running Linux in an LPAR on an IBM System z10 EC or later, AP interrupts are used instead of the polling thread. The polling thread is disabled when AP interrupts are available.

Depending on the Linux distribution being used, it is possible to display zcrypt information using the lszcrypt command. Also, the chzcrypt command enables and disables the cryptographic cards. See Device Drivers, Features, and Commands, SC33-8411. Because there are some versions of this book that are for a specific Linux kernel, make sure that you refer to the book that goes with your specific Linux kernel level.

High resolution polling timer

The zcrypt driver can run with or without the polling thread. When running with the polling thread, one CPU without an outstanding workload is constantly polling the cryptographic cards for finished cryptographic requests. The polling thread sleeps when no cryptographic requests are being processed. This mode utilizes the cryptographic cards as much as possible at the cost of blocking one CPU. Without
the polling thread, the cryptographic cards are polled at a much lower rate. This could result in higher latency and reduced throughput for cryptographic requests.

The high resolution polling timer is available for some Linux distributions. This timer can be used to set the polling frequency to be larger than 100 Hz. See the chzcrypt command in *Device Drivers, Features, and Commands*, SC33-8411. Be aware that setting a value larger than 100 Hz is not valid if AP interrupts are used.

### The sysfs interface

The zcrypt cryptographic driver utilizes the device model introduced with the Linux 2.6 kernel series. It introduces a new bus named "AP" which can be found under `/sys/bus/ap`.

The following attributes are defined at AP bus level:

- `/sys/bus/ap/ap_domain` Read-only attribute representing the domain index used for all AP devices.
- `/sys/bus/ap/ap_interrupts` Read-only attribute indicating whether AP adapter interrupts are used. A value of 1 means that AP adapter interrupts are used. A value of 0 means that they are not used.
- `/sys/bus/ap/config_time` Read-write attribute representing the interval in seconds for re-scanning the AP bus for new or gone devices.
- `/sys/bus/ap/drivers_autoprobe` This attribute controls whether a bus can bind devices by default, indicated by a value of 1. If not (value of 0), the bus initializes the device and does nothing else.
- `/sys/bus/ap/drivers_probe` This attribute controls whether a bus (its ID is given as input) can attempt to bind a driver to this device. A value of 1 is ‘yes’ and a value of 0 is ‘no’.
- `/sys/bus/ap/poll_thread` Read-write attribute indicating whether a polling thread is to be used to increase cryptographic performance. By writing 0 or 1 to this attribute the poll thread can be disabled or enabled.
- `/sys/bus/ap/poll_timeout` Read-write attribute representing the interval (in nanoseconds) for polling timeout on the AP bus.
- `/sys/bus/ap/uevent` File for storing uevents. Every time that the AP bus detects a device addition or removal, a new uevent is generated. If there is a udev rule, it can catch this event and perform appropriate actions.

For each cryptographic adapter a new directory in `/sys/bus/ap/devices` is created using the following naming convention: cardxx where xx is the device index for each device. The valid device index range is hex X'00' to X'3f'. For example, device X'1a' can be found under `/sys/bus/ap/devices/card1a`. Within each device directory the following attributes can be found:

- `depth` Read-only attribute representing the input queue length for this device.
hwtype  Read-only attribute representing the hardware type for this device. The following values are defined:

3  PCICC cards
4  PCICA cards
5  PCIXCC cards
6  CEX2A cards
7  CEX2C cards
8  CEX3A cards
9  CEX3C cards

modalias  Read-only attribute representing an internally used device bus-ID.

online  Read-write attribute representing the online status for this AP device. Writing 0 or 1 to this attribute sets this device offline or online.

request_count  Read-only attribute representing the number of requests already processed by this device.

type  Read-only attribute representing the type of this device. The following types are defined:

- PCICC
- PCICA
- PCIXCC_MCL2
- PCIXCC_MCL3
- CEX2A
- CEX2C
- CEX3A
- CEX3C

Running secure key under a z/VM guest

In order to use the CEX3C feature under z/VM versions 6.1, 5.3 and 5.4, you need to apply these APAR fixes:

VM64656  Introduces CEX3C support.
VM64793  Introduces protected key CPACF support.

To get secure key running under a z/VM guest, a directory control statement (CRYPTO APDED) for a given VM guest needs to be used. This require 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.

For clear key, z/VM does a virtualization for the AP’s, which is done by the user statement:
The domain must have one or more AP's for this. If available, an accelerator is used. If not, a coprocessor is taken. The guest always sees only one card regardless of how many cards are owned by the LPAR.

This requires that the cards are made available to the LPAR in which VM is running. The domain and APs must be defined in the LPAR profile.

Note that the CCA does not use and cannot handle the clear key settings.
Appendix I. CCA installation instructions

This appendix contains CAA V 4.0.0 installation, configuration, and uninstallation instructions.

Before you begin

Before you begin the CCA installation, review these points:

- Ensure that you are using supported hardware. See “Hardware requirements” on page xx.
- Ensure that your Linux distribution has the zcrypt device driver support. For details, see “Installing and loading the cryptographic device driver” on page 535.
- If you are going to use the Java Native Interface (JNI), see “Building Java applications to use with the CCA JNI” on page 18 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 new CEX3C CCA 4.0 RPM will support CEX2C access, including AES (for up-to-date CEX2C firmware).
  - The new TKE catcher daemon supports managing CEX2C as well as CEX3C.
  - You are advised to pay special attention to Appendix J, “Coexistence,” on page 547 and particular the section “Dual Support: Key storage interactions” on page 549.

Download and install the RPM

The name of the CCA RPM is: csulcca-4.0.0-10.s390x.rpm. The CCA RPM contains files, samples, and groups.

To download the CCA RPM, complete these steps:

1. Point your Web browser at this location:
   http://www.ibm.com/security/cryptocards
2. Locate the box on the left side labeled Cryptocards.
3. Click the PCle Cryptographic Coprocessor link from the Cryptocards box.
4. The Cryptocards box will be updated with a submenu underneath the PCle 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 Obtaining CCA software for 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 234.

/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 234.

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
/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/CEX3C/bin/TKECM.dat
/opt/IBM/CEX3C/bin/acpoints.dat
/opt/IBM/CEX3C/bin/catcher.exe
/opt/IBM/CEX3C/bin/panel.exe
Utility: run with no arguments for usage
/opt/IBM/CEX3C/bin/ivp.e
Utility: run with no arguments for install verification
/opt/IBM/CEX3C/bin/profile.perl
/opt/IBM/CEX3C/doc/README.linz
/opt/IBM/CEX3C/include/csulincl.h
/opt/IBM/CEX3C/include/HIKM.h
/opt/IBM/CEX3C/doc/hikmNativeInteger.html
/opt/IBM/CEX3C/cnm/HIKM.zip
/opt/IBM/CEX3C/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.4.0.0
/usr/lib64/libcsulcca.so.4.0.0
/usr/lib64/libcsulccamk.so
Link to: /usr/lib64/libcsulccamk.so.4
/usr/lib64/libcsulccamk.so.4
Link to: /usr/lib64/libcsulccamk.so.4.0.0
/usr/lib64/libcsulccamk.so.4.0.0
Samples in the RPM

These samples are included in the RPM:

- /opt/IBM/CEX3C/samples/mac.c
  - C code sample
- /opt/IBM/CEX3C/samples/makefile.lnx
  - Used to build mac.c
- /opt/IBM/CEX3C/samples/mac.java
  - JAVA code sample
- /opt/IBM/CEX3C/samples/RNG.java
  - JAVA code sample

Groups in the RPM

These groups are created for the purpose of loading master keys. See Table 137 on page 545.

- 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 V 4.0.0 RPM.

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:
   - rpm -i <rpm name>

Notes:

a. If this is an upgrade you can use rpm -Uvh
b. If you are installing the RPM on a Novell 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/CEX3C/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:
      lsmod
      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

     Notes:
     1) This works if you have only one domain assigned to the LPAR (or z/VM
guest using a dedicated CEX3C 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).
     2) SUSE Linux uses its own start script, 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
      command:
      /opt/IBM/CEX3C/bin/ivp.e
      This will health check all active cards.
      /opt/IBM/CEX3C/bin/panel.exe -x
      This will show 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 will also be reported.

     Notes:
     1) To be able to use /opt/IBM/CEX3C/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.

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
   Redbook 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 545
   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

542   Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
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/CEX3C/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 get 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 '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

   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 calls for Red Hat, Fedora, and SUSE)
   a) Create user cca_user, which will own default key storage by issuing the following:

   useradd -g cca_admin -d /home/cca_user -m cca_user

   This creates the user with primary group cca_admin and a new home directory.
passwd cca_user

This sets the new user's password.

b) Create user cca_lfmkp by issuing the following:

useradd -g cca_admin -d /home/cca_lfmkp -G cca_admin,cca_lfmkp -m cca_lfmkp

This creates the user with primary group cca_admin, secondary group cca_lfmkp, and a new home directory.

passwd cca_lfmkp

This sets the new user's password.

c) Create user cca_cmkp by issuing the following:

useradd -g cca_admin -d /home/cca_cmkp -G cca_admin,cca_cmkp -m cca_cmkp

This creates the user with primary group cca_admin, secondary group cca_cmkp, and a new home directory.

passwd cca_cmkp

This sets the new user's password.

d) Create user cca_clrmk by issuing the following:

useradd -g cca_admin -d /home/cca_clrmk -G cca_admin,cca_clrmk -m cca_clrmk

This creates the user with primary group cca_admin, secondary group cca_clrmk, and a new home directory.

passwd cca_clrmk

This sets the new user's password.

e) Create user cca_setmk

useradd -g cca_admin -d /home/cca_setmk -G cca_admin,cca_setmk -m cca_setmk

This creates the user with primary group cca_admin, secondary group cca_setmk, and a new home directory.

passwd cca_setmk

This sets the new user's password.

b. Add group membership privileges to users based on their required function.
Table 137. CCA groups

<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>

- Load FIRST, MIDDLE (optional), and LAST key parts for the AES, SYM and ASYM master keys and then call SET for each master key. This can be accomplished 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.

For details about panel.exe, see [The panel.exe utility](#) on page 551.

**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 553.](#) See also ["Dual Support: Key storage interactions" on page 549.](#)

9. **Key storage re-encipher when changing the master key** - To perform this step, see ["Using panel.exe for key storage re-encipher when changing the master key" on page 555.](#)

10. If you are going to be using Central Processor Assist for Cryptographic Functions (CPACF), it must be configured. See ["CPACF support" on page 8.](#)

## Uninstall the RPM

Use the following steps to uninstall CCA V 4.0.0 RPM:

1. Uninstall any RPMs that depend on the csulcca-4.0.0-10.s390x.rpm. If you try to uninstall the csulcca-4.0.0-10.s390x.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 csulcca-4.0.0-10.s390x.rpm.
   - You have to use the full name. You can find that out by issuing the following:
rpm -qa |grep csulcca

b. Login as root. You have to be root to uninstall the RPM.

c. Uninstall the RPM with the following command:

    rpm -e <rpm name>

**Notes:**

a. 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) then the un-install process will not be able to remove those groups. Just 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 544 of "Install and configure the RPM" on page 541).
Appendix J. Coexistence

This section discusses trade-offs when configuring CEX2C features together with CEX3C features available to the same Linux instance.

Legacy support

These are legacy support considerations.

- Running this release in parallel with the prior release RPM and an IBM 4764 Crypto Express2 (CEX2C) feature in coprocessor mode configured to the same system image is a supported configuration. See "Concurrent installations."

- This release of host library and follow-on releases are supported for interaction with the (CEX2C) feature in coprocessor mode. See "Dual Support: TKE catcher can run in only one instance" on page 550.

Concurrent installations

These are background considerations for installation of the CEX3C and CEX2C RPM alongside the previously released RPM for the CEX2C.

1. The libcsulcca.so (Release 4.0.0) and libcsulsapi.so (release 3.24) CCA libraries 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/CEX3C/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 549.

3. See "Interaction between the ‘default card’ and use of Protected Key CPACF" on page 13 for a concurrency and CPACF.

CEX3C and CEX2C co-install toleration

The new RPM supports accessing the CEX2C as well as the CEX3C, with the first CEX3C becoming the ‘default’ adapter. This can be changed with the elsewhere noted environment variable method. Using the new RPM is your best option for accessing a CEX2C as well as a CEX3C 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 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 234.

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.4.0.0` to `libcsulsapi.so`, `libcsulsecy.so`, `libds30.so` and `libcsulcall`
3. Create a soft link of `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:**

**IMPORTANT**

This is not a supported operation. The csulcca RPM cannot detect this scenario to try to recover the csulcca package function.

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 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. Re-install 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 548.
   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 234](#) for a discussion 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/CEX3C/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:
   
   ```
   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 adapters. The new TKE catcher daemon can work with both CEX2C and CEX3C adapters, so it is preferred for all systems with co-install of the xcryptolinzGA RPM and the csulcca RPM.

**Caveats:**

- 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, 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**

The following factors together should be considered and managed carefully for user installations:

- The environment variables described in ["Environment variables for the key storage file" on page 234](#) have the same names for the 4.0 release of the CCA RPM as for the earlier release, in order to ease application porting efforts.
- Installation of the CCA 4.0 RPM on top of an xcryptolinz RPM installation (the recommended method to accomplish coexistence of the two packages for porting or debug environments) will change how these variables are defined for any user performing login after the CCA 4.0 RPM install, such that these 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 4.0 RPM, the next time that the legacy application starts. These are likely consequences:

- 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 4.0 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 install (look for the LINZCRYPT section). They were defined as follows:
  - CSUDESLD=/opt/IBM/4764/keys/deslist
  - CSUDESDS=/opt/IBM/4764/keys/des.key
  - CSUPKALD=/opt/IBM/4764/keys/pkalist
  - CSUPKADS=/opt/IBM/4764/keys/pka.key
- 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.

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 release. Therefore, the new daemon supports TKE management communication to both the CEX2C and CEX3C adapters. Special steps are taken in the install/uninstall and daemon management for the CEX3C release to ensure that the new daemon is running when it is available.

You must have a TKE V6.0 workstation in order to see supported CEX3Cs. They are not seen when using TKE V5 workstations. For more information about the TKE workstation, see z/OS Cryptographic Services ICSF: Trusted Key Entry PCIX Workstation User’s Guide.
Appendix K. Utilities

These two utilities are used in this document:

ivp.e

This utility is used to verify installation, when run without arguments.

This utility can also be used to tell you whether your cards are CEX3C or CEX2C, 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 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/CEX3C/bin/panel.exe

The panel.exe utility

The panel.exe utility is installed by the Linux for System z Cryptographic Coprocessor install package or RPM to this path in the Linux system:

/opt/IBM/CEX3C/bin/panel.exe

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:

Usage
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---

-?, -h: Usage
   -x: List crypto resources (and basic status)
   -m: List CPACF (local CPU crypto) resources

---MASTER KEY (MK)---

To LOAD a Master Key (MK) PART:
   -l (for interactive)
OR===>
   -l -t [A|S|E] -p [F|M|L] KEYPART
where: -t [A|S|E] is which MK: A=ASYM, S=SYM, E=AES
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]
where: -t [A|S|E] is which MK: A=ASYM, S=SYM, E=AES

To CLEAR a Master Key 'New' Register:
  -c (for interactive)
OR===>
  -c -t [A|S|E]
where: -t [A|S|E] is which MK: A=ASYM, S=SYM, E=AES

To QUERY a Master Key Verification Pattern:
  -q (for interactive)
OR===>
  -q -t [A|S|E] -r [N|C|O]
where: -t [A|S|E] is which MK: A=ASYM, S=SYM, E=AES
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

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.

panel.exe functions

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.
- 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 re-encipher when changing the master key” on page 555.

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 42 for that functionality.

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 542) 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 discussions elsewhere in this manual, including the verb “Key Storage Initialization (CSNBKSI)” on page 95). 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 551.

If you are planning to use both the CEX2C and CEX3C in the same environment, see the information about the key storage environment variable in “Concurrent installations” on page 547.

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/CEX3C/keys/ path, where before the path contained /*/4764/*. There 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 543 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 542.

RECALL: To be able to use /opt/IBM/CEX3C/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 (default)
   a. Ensure permissions to the default location (/opt/IBM/CEX3C/keys/) allow your user to perform this operation.
   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).

   /opt/IBM/CEX3C/bin/panel.exe -t AES -i
   /opt/IBM/CEX3C/bin/panel.exe -t DES -i
   /opt/IBM/CEX3C/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/CEX3C/bin/panel.exe -t AES -f <AES file name> -i
   /opt/IBM/CEX3C/bin/panel.exe -t DES -f <DES file name> -i
   /opt/IBM/CEX3C/bin/panel.exe -t PKA -f <PKA file name> -i

   For example, if you entered the following commands:

   /opt/IBM/CEX3C/bin/panel.exe -t AES -f /tmp/a -i
   /opt/IBM/CEX3C/bin/panel.exe -t DES -f /tmp/d -i
   /opt/IBM/CEX3C/bin/panel.exe -t PKA -f /tmp/p -i

   These files would be created:

   /tmp/a
   /tmp/a.NDX
   /tmp/d
Using panel.exe for key storage re-encipher 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-enciphered when the master key is changed. 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 re-encipher 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 log in and drive the re-encipher. This can be done programmatically (using several verbs) or with /opt/IBM/CEX3C/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 re-encipher when changing the master key.

1. To re-encipher default key storage with panel.exe use:

   /opt/IBM/CEX3C/bin/panel.exe -t AES -r
   /opt/IBM/CEX3C/bin/panel.exe -t DES -r
   /opt/IBM/CEX3C/bin/panel.exe -t PKA -r

2. To re-encipher non-default key storage with panel.exe use:

   /opt/IBM/CEX3C/bin/panel.exe -t AES -f <AES file name> -r
   /opt/IBM/CEX3C/bin/panel.exe -t DES -f <DES file name> -r
   /opt/IBM/CEX3C/bin/panel.exe -t PKA -f <PKA file name> -r
Notices

This information was developed for products and services offered in the USA.

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
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**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**

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- AIX®
- AS/400
- Enterprise Storage Server®
- eServer™
- FICON®
- HiperSockets™
- IBM
- IBMLink
- Processor Resource/Systems Manager™
- RACF
- S/390
- System i
- System p
- System x
- System z
- System z10™
- System/390®
- TotalStorage®
- z/Architecture®
- z9®
- z10
- z/OS
- z/VM
- zSeries®
Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a 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.
Index

Numerics
3621 PIN block format 284, 479
3624 PIN block format 284, 479
4700-PAD 499
4704-EPP PIN block format 284

A
about this document xix
access control points 519
CPACF 9
access control verbs 519
AES encryption 192
AES encryption algorithm 205, 212
AES key 10
AES Key Record Create (CSNBAKRC) 239
format 239
JNI version 240
overview 35
parameters 239
related information 240
required commands 240
restrictions 240
AES Key Record Delete (CSNBAKRD) 242
format 242
JNI version 243
overview 35
parameters 242
related information 243
required commands 243
restrictions 243
AES Key Record List (CSNBAKRL) format 244
JNI version 246
overview 35
parameters 244
related information 246
required commands 246
AES Key Record Read (CSNBAKRR) 247
JNI version 248
overview 35
parameters 247
related information 248
required commands 248
restrictions 248
AES Key Record Write (CSNBAKRW) 249
JNI version 250
overview 35
parameters 249
related information 250
required commands 250
restrictions 250
AES key storage 233
AES master key 25
AES-MK 25

algorithm 37
3624 PIN generation 481
3624 PIN verification 484
DES 21, 37
GBP PIN generation 482
GBP PIN verification 486
GBP-PIN 321
IBM-PIN 321
IBM-PINO 321
Interbank PIN generation 490
PIN offset generation 483
PIN, detailed 481
PIN, general 40
PVV generation 489
PVV verification 490
RSA 53
VISA PIN 489
VISA-PVV 297, 321
VISAPVV4 321
ANSI 9.9-1 algorithm 217
ANSI X3.106 (CBC) 498
ANSI X9.19 optional double MAC procedure 217
ANSI X9.19 Optional Procedure 1 MAC 505
ANSI X9.23 499
ANSI X9.23 processing rule 192, 196, 201
ANSI X9.31 hash format 516
ANSI X9.8 318
ANSI X9.8 PIN block format 477
ANSI X9.9 MAC 504
APAR fixes
z/VM xxi, 537
application programs
service request 7
array_key_left_identifier parameter
Control Vector Translate verb 112
array_key_right_identifier parameter
Control Vector Translate verb 112
asym_encrypted_key parameter
Remote Key Export verb 404
asym_encrypted_key_length parameter
Remote Key Export verb 404
ASYM-MK 25, 53
asymmetric keys master key 25
authenticating messages 217
authentication 38
authentication_issuer_master_key_identifier parameter
PIN Change/Unblock verb 326
authentication_issuer_master_key_length parameter
PIN Change/Unblock verb 325

B
Bellare-Rogaway 517
block_size parameter
Symmetric Algorithm Decipher verb 207
Symmetric Algorithm Encipher verb 213
Byte *
description 18
C

C

c-variable_encrypting_key_identifier parameter
Cryptographic Variable Encipher verb 116

calculation method
MAC padding method 504
message authentication code (MAC) 504
Modification Detection Code (MDC) 495
X9.19 method 504

CBC processing rule 191, 192, 195, 201, 205, 212

CBC processing rules
Decipher 193
Symmetric Algorithm Decipher 203
Symmetric Algorithm Encipher 210

CCA
functions 21
overview 21

CCA access control 5

CCA AES, DES, and hashing verbs 46

CCA API 3

CCA application
compile 17
JAVA 18
link 17

CCA DES-key verification 494

CCA description 5

CCA functional overview 3

CCA installation 539

CCA library 7

CCA master key 6
establishing 6

CCA nodes and access control verbs 46, 65

CCA programming 3, 13

CCA RPM
configure 541
download 539
files 539
groups 541
samples 541
uninstall 545

CCA sample program 525
C 525
Java 530

CCA software support 4

CCA system setup 535

CCA verbs 3, 63

CCA verbs description 7

Central Processor Assist for Cryptographic Functions (CPACF) xxi, 537

certificate parameter
Remote Key Export verb 401
certificate_length parameter
Remote Key Export verb 401
certificate_parms parameter
Remote Key Export verb 402
certificate_parms_length parameter
Remote Key Export verb 401

CEX2C xix, xx, xxi, 30, 65, 547
CEX2C coprocessors 5
CEX3C xix, xx, xxi, 3, 5, 30, 54, 65, 537, 547

data protection 21

chain_data parameter
Symmetric Algorithm Decipher verb 207
Symmetric Algorithm Encipher verb 214

chain_data_length parameter
Symmetric Algorithm Decipher verb 207
Symmetric Algorithm Encipher verb 214

chain_data_length parameter
Symmetric Algorithm Decipher verb 207
Symmetric Algorithm Encipher verb 214

chaining_vector parameter
Decipher verb 196
Encipher verb 201
MAC Generate verb 221
MAC Verify verb 226
MDC Generate verb 354
One-Way Hash verb 231

chaining_vector_length parameter
One-Way Hash verb 230

changing control vectors 471
cipher block chaining (CBC) 191

CIPHER key type 26

CIPHER keys
definition 25

cipher_text parameter
Decipher verb 195
Encipher verb 202
ciphering method 499
ciphering methods 497
ciphertext
deciphering 191

ciphertext parameter
Cryptographic Variable Encipher verb 116
Symmetric Algorithm Decipher verb 207
Symmetric Algorithm Encipher verb 214


ciphertext_length parameter
Symmetric Algorithm Decipher verb 207
Symmetric Algorithm Encipher verb 214

clear key
definition 29
protecting 191

Clear Key Import (CSNBCKI) 105
format 106
JNI version 107
overview 32
parameters 106
required commands 106

Clear PIN Encrypt (CSNBCPE) 287
format 288
JNI version 290
overview 41
parameters 288
required commands 289
restrictions 289

Clear PIN Generate (CSNBPGN) 291
format 291
JNI version 293
overview 41
parameters 291
related information 293
required commands 293
usage notes 293

Clear PIN Generate Alternate (CSNBCPA) 295

extraction rules 480
format 295
Clear PIN Generate Alternate (CSNBCPA) (continued)
   JNI version 298
   overview 41
   parameters 295
   required commands 298
clear_key parameter
   Clear Key Import verb 106
   Multiple Clear Key Import verb 172
clear_key_length parameter
   Multiple Clear Key Import verb 172
clear_master_key parameter
   Key Storage Initialization verb 96
clear_PIN parameter
   Clear PIN Encrypt verb 289
clear_text parameter
   Decipher verb 196
   Encipher verb 200
   Secure Messaging for Keys verb 331
   Secure Messaging for PINs verb 336
cleartext parameter
   Symmetric Algorithm Decipher verb 208
   Symmetric Algorithm Encipher verb 214
cleartext_length parameter
   Symmetric Algorithm Decipher verb 208
   Symmetric Algorithm Encipher verb 214
coeexistence 547
commands
   ivp.e 30
commands and access control, roles, profiles
description 7
Common Cryptographic Architecture (CCA)
description xix, xx
common parameters 15
concurrent installations 547
contact IBM xxiii
continue processing rule 196, 201, 206, 213
control information (continued)
   for PKA key generate 364
   for PKA Key Record Delete 266
   for PKA Key Token Build 371
   for PKA Symmetric Key Export 387
   for PKA Symmetric Key Generate 390
   for PKA Symmetric Key Import 395
   for Secure Messaging for Keys 331
   for Secure Messaging for PINs 335
   for Symmetric Algorithm Decipher 205
   for Symmetric Algorithm Encipher 212
   for Transaction Validation 339
control vector
definition 21, 24
description 461
value 461
Control Vector Generate (CSNBCVG) 108
   format 108
   JNI version 109
   overview 32
   parameters 108
   usage notes 109
control vector keyword combinations 28
control vector table 461
Control Vector Translate (CSNBCVT) 111
   format 111
   JNI version 113
   overview 32
   parameters 111
   required commands 113
   usage notes 113
control vectors, changing 471
control_vector parameter
   Control Vector Generate verb 109
   Key Token Build verb 159
   Key Token Parse verb 167
   Control-vector-base bit maps 463
coprocessor certification 6
coprocessor resource selection 89, 92
CPACF xxi, 8, 194, 199, 219, 224, 537
   access control points 9
   clear key 10, 12
   default card 13
   environment variables 9
   illustration 10
   preparation at startup 12
   protected key 10
   using protected key 12
Crypto Express2 feature xix, xx
Crypto Express3 feature xix
cryptographic device driver
   installing 535
cryptographic engine 5
Cryptographic Facility Query (CSUACFQ) 65
   format 66
   JNI version 85
   overview 31
   parameters 66
   required commands 85
   restrictions 85
Cryptographic Facility Version (CSUACFV) 87
data confidentiality 3
data integrity 3
  managing 38
  verifying 217
data integrity and authentication verbs 219
data key
  exporting 118
  importing 105
  re-enciphering 118
Data Key Export (CSNBDKX) 118
  format 118
  JNI version 119
  overview 33
  parameters 118
  required commands 119
  restrictions 119
Data Key Import (CSNBDKM) 120
  format 120
  JNI version 121
  overview 33
  parameters 120
  required commands 121
  restrictions 121
  usage notes 121
data parameter
  Diversified Key Generate verb 124
data_array parameter
  Clear PIN Generate Alternate verb 297
  Clear PIN Generate verb 292
  Encrypted PIN Generate verb 310
  Encrypted PIN Verify verb 321
data_length parameter
  Diversified Key Generate verb 124
data_structure parameter
  PKA Decrypt verb 175
  PKA Encrypt verb 179
data_structure_length parameter
  PKA Decrypt verb 175
  PKA Encrypt verb 179
data-encrypting key
  definition 25
  generating 43
  length 25
DATAC 25, 44
DATAC key type 26
DATAM 25, 44
DATAM key type 26
DATAMV key type 26
dataset_name parameter
  AES Key Record List verb 245
  DES Key Record List verb 257
  PKA Key Record List verb 268
dataset_name_length parameter
  AES Key Record List verb 245
  DES Key Record List verb 257
  PKA Key Record List verb 268
de-allocating a coprocessor resource 92
Decipher (CSNBDEC) (continued)
  parameters 194
  required commands 196
  restrictions 196
  DECIPHER key type 26
  Decipher processing rules 193, 203
deciphering
  data 191, 193
  multiple 507
default card 13
DES algorithm 21, 37, 191
DES cryptographic key verbs 105
DES cryptography 21
DES encryption
  triple 191
DES encryption algorithm 196
DES encryption algorithm processing rule 201
DES external key token format 432
DES internal key token format 429
DES key 9
DES key flow 23
DES Key Record Create (CSNBKRC) 252
  format 252
  JNI version 253
  overview 35
  parameters 252
  related information 253
  required commands 253
  restrictions 252
DES Key Record Delete (CSNBKRD) 254
  format 254
  JNI version 255
  overview 36
  parameters 254
  related information 255
  required commands 255
  restrictions 255
DES Key Record List (CSNBKRL) 256
  format 256
  JNI version 258
  overview 36
  parameters 256
  related information 258
  required commands 257
DES Key Record Read (CSNBKRR) 259
  format 259
  JNI version 260
  overview 36
  parameters 259
  related information 260
  required commands 260
  restrictions 260
DES Key Record Write (CSNBKRW) 261
  format 261
  JNI version 262
  overview 36
  parameters 261
  related information 262
  required commands 262
  restrictions 262
  DES key storage 233
DES key-storage initialization 95
DES keys 32
  managing 105
device key 6
digital signature 3
  using 343
Digital Signature Generate (CSNDDSG) 343
  format 343
  JNI version 346
  overview 54
  parameters 344
  required commands 346
  restrictions 346
digital signature verbs 54, 343
Digital Signature Verify (CSNDDS V) 348
  format 348
  JNI version 350
  overview 54
  parameters 348
  related information 350
  required commands 350
  restrictions 350
Diversified Key Generate (CSNBDKG) 122
  format 122
  JNI version 125
  overview 33
  parameters 122
  required commands 125
  usage notes 125
DKYGENKY key type 26
double-length key
  multiple decipherment 511
  multiple encipherment 510
  using 26

E
ECB processing rule 205, 212
ECI-1 318
ECI-2 PIN block format 284, 479
ECI-3 PIN block format 284, 480
ECI-4 318
edition notice ii
electronic code book (ECB) 507
EMV MAC 505
Encipher (CSNBENC) 198
  format 199
  JNI version 202
  overview 37
  parameters 199
  required commands 202
  restrictions 202
ENCIPHER key type 26
Encipher processing rules 198, 210
enciphered
  key 130, 191
  under master key 138
enciphered_text parameter
  Secure Messaging for Keys verb 332
  Secure Messaging for PINs verb 336
enciphering
  data 191, 198
  multiple 507
encrypt zeros DES-key verification 495
Encrypted PIN Generate (CSNBE PG) 308
  format 308
  JNI version 311
  overview 41
  parameters 308
  required commands 311
  restrictions 311
Encrypted PIN Translate (CSNBP TR) 313
  extraction rules 480
  format 313
  JNI version 318
  overview 41
  parameters 313
  related information 317
  required commands 317
  usage notes 318
Encrypted PIN Verify (CSNBPV R) 319
  extraction rules 480
  format 319
  JNI version 323
  overview 41
  parameters 319
  related information 323
  required commands 322
encrypted_PIN_block parameter
  Clear PIN Encrypt verb 289
  Clear PIN Generate Alternate verb 296
  Encrypted PIN Generate verb 311
  Encrypted PIN Verify verb 320
encryption algorithm processing rule
  AES 205, 212
  DES 196, 201
encryptionIssuer_master_key_identifier parameter
  PIN Change/Unblock verb 326
cryptographic issuer_master_key_length parameter
  PIN Change/Unblock verb 326
tentry-point names 13
  prefixes 13
environment variables
  CSU_HCPUACLR 9
  CSU_HCPUAPRT 9
establishing a CCA master key 6
EX key form 44
EXEX key form 46
exit_data parameter 15
AES Key Record Create verb 240
AES Key Record Delete verb 242
AES Key Record List verb 245
AES Key Record Read verb 247
AES Key Record Write verb 249
Clear Key Import verb 106
Clear PIN Encrypt verb 288
Clear PIN Generate Alternate verb 296
Control Vector Generate verb 108
Control Vector Translate verb 111
Cryptographic Facility Query verb 66
Cryptographic Facility Version verb 87
exit_data parameter (continued)
Cryptographic Resource Allocate verb 89
Cryptographic Resource Deallocate verb 92
Cryptographic Variable Encipher verb 116
CVV Generate verb 300
CVV Verify verb 304
Data Key Export verb 118
Data Key Import verb 120
Decipher verb 194
DES Key Record Create verb 252
DES Key Record Delete verb 254
DES Key Record List verb 257
DES Key Record Read verb 259
DES Key Record Write verb 261
Digital Signature Generate verb 344
Digital Signature Verify verb 348
Diversified Key Generate verb 123
Encipher verb 200
Encrypted PIN Generate verb 309
Encrypted PIN Translate verb 313
Encrypted PIN Verify verb 319
Key Export verb 127
Key Generate verb 130
Key Import verb 138
Key Part Import verb 143
Key Storage Initialization verb 95
Key Test Extended verb 153
Key Test verb 148
Key Token Build verb 158
Key Token Change verb 161
Key Token Parse verb 165
Key Translate verb 169
MAC Generate verb 220
Master Key Process verb 99
MCA Verify verb 225
MDC Generate verb 353
Multiple Clear Key Import verb 171
One-Way Hash verb 229
PIN Change/Unblock verb 325
PKA Decrypt verb 174
PKA Encrypt verb 178
PKA Key Generate verb 364
PKA Key Import verb 367
PKA Key Record Create verb 263
PKA Key Record Delete verb 265
PKA Key Record List verb 268
PKA Key Record Read verb 270
PKA Key Record Write verb 272
PKA Key Token Build verb 371
PKA Key Token Change verb 377
PKA Key Translate verb 381
PKA Public Key Extract verb 384
PKA Symmetric Key Export verb 386
PKA Symmetric Key Generate verb 389
PKA Symmetric Key Import verb 394
Prohibit Export Extended verb 184
Prohibit Export verb 182
Random Number Generate Long verb 188
Random Number Generate verb 186
Random Number Tests verb 103
Remote Key Export verb 400
exit_data parameter (continued)
Retained Key Delete verb 275
Retained Key List verb 279
Secure Messaging for Keys verb 330
Secure Messaging for PINs verb 334
Symmetric Algorithm Decipher verb 205
Symmetric Algorithm Encipher verb 212
Transaction Validation verb 338
Trusted Block Create verb 410
exit_data_length parameter 15
AES Key Record Create verb 240
AES Key Record Delete verb 242
AES Key Record List verb 245
AES Key Record Read verb 247
AES Key Record Write verb 249
Clear Key Import verb 106
Clear PIN Encrypt verb 288
Clear PIN Generate Alternate verb 295
Clear PIN Generate verb 291
Control Vector Generate verb 108
Control Vector Translate verb 111
Cryptographic Facility Query verb 66
Cryptographic Facility Version verb 87
Cryptographic Resource Allocate verb 89
Cryptographic Resource Deallocate verb 92
Cryptographic Variable Encipher verb 115
CVV Generate verb 300
CVV Verify verb 304
Data Key Export verb 118
Data Key Import verb 120
Decipher verb 194
DES Key Record Create verb 252
DES Key Record Delete verb 254
DES Key Record List verb 256
DES Key Record Read verb 259
DES Key Record Write verb 261
Digital Signature Generate verb 344
Digital Signature Verify verb 348
Diversified Key Generate verb 123
Encipher verb 200
Encrypted PIN Generate verb 309
Encrypted PIN Translate verb 313
Encrypted PIN Verify verb 319
Key Export verb 127
Key Generate verb 130
Key Import verb 138
Key Part Import verb 143
Key Storage Initialization verb 95
Key Test Extended verb 153
Key Test verb 148
Key Token Build verb 158
Key Token Change verb 161
Key Token Parse verb 165
Key Translate verb 169
MAC Generate verb 220
Master Key Process verb 99
MCA Verify verb 224
MDC Generate verb 353
Multiple Clear Key Import verb 171
One-Way Hash verb 229
PIN Change/Unblock verb 325
PKA Decrypt verb 174
PKA Encrypt verb 178
PKA Key Generate verb 364
PKA Key Import verb 367
PKA Key Record Create verb 263
PKA Key Record Delete verb 265
PKA Key Record List verb 268
PKA Key Record Read verb 270
PKA Key Record Write verb 272
PKA Key Token Build verb 371
PKA Key Token Change verb 377
PKA Key Translate verb 381
PKA Public Key Extract verb 384
PKA Symmetric Key Export verb 386
PKA Symmetric Key Generate verb 389
PKA Symmetric Key Import verb 394
Prohibit Export Extended verb 184
Prohibit Export verb 182
Random Number Generate Long verb 188
Random Number Generate verb 186
Random Number Tests verb 103
Remote Key Export verb 400
ICV selection processing rule
  continue 196, 201, 206, 213
  initial 196, 201, 206, 213
IEKYXLAT key type 26
IKEYXLAT 26
IM key form 44
IMEX key form 45
importable key
  generating 44
importable key form
definition 22
  value 131
IMPORTER key type 26
importer key-encrypting key 26
importer_key_identifier parameter
  Data Key Import verb 120
  Key Import verb 139
  PKA Key Import verb 368
  Remote Key Export verb 403
importer_key_identifier_length parameter
  Remote Key Export verb 403
INBK PIN 291
INBK-PIN 319
initial processing rule 196, 201, 206, 213
initialization_vector parameter
  Cryptographic Variable Encipher verb 116
  Decipher verb 195
  Encipher verb 200
  Secure Messaging for Keys verb 331
  Secure Messaging for PINs verb 336
  Symmetric Algorithm Decipher verb 207
  Symmetric Algorithm Encipher verb 213
initialization_vector_length parameter
  Symmetric Algorithm Decipher verb 207
  Symmetric Algorithm Encipher verb 213
initializing key storage 95
input_block_identifier parameter
  Trusted Block Create verb 410
input_block_identifier_length parameter
  Trusted Block Create verb 410
input_KEY_key_identifier parameter
  Key Translate verb 169
input_key_identifier parameter
  Secure Messaging for Keys verb 331
input_key_token parameter
  Key Translate verb 169
input_PAN_data parameter
  Secure Messaging for PINs verb 335
input_PIN block parameter
  Secure Messaging for PINs verb 335
input_PIN_encrypting_key_identifier parameter
  Encrypted PIN Translate verb 314
  Encrypted PIN Verify verb 320
input_PIN_profile parameter
  Encrypted PIN Translate verb 314
  Encrypted PIN Verify verb 320
  Secure Messaging for PINs verb 335
input/output (I/O) parameters 14
Interbank PIN 49, 283, 291, 319
intermediate PIN-block (IPB) 479
internal AES key-token
  flag byte 430
internal key token 16, 58, 59
  AES 429
  clear 430
  definition 23
  DES 429, 431
  PKA
    RSA private 438, 439, 440
  IPB (intermediate PIN-block) 479
  IPINENC key type 26, 314
  IPS processing rule 196, 201
ISO 16609 TDES MAC 505
ISO format 0 477
ISO Format 1 478
ISO format 2 478
ISO format 3 478
ISO-0 PIN block format 284, 477
ISO-1 PIN block format 284, 478
ISO-2 PIN block format 284, 478
ISO-3 PIN block format 284
ISO-3 PIN-block format 478
ivp.e 30
  command 30, 551
  utility 551
J
JAVA
  data types 18
  entry point names 18
  supported versions 18
JAVA Native Interface (JNI) 18
K
kek_key_identifier parameter
  Key Test Extended verb 155
KEK_key_identifier parameter
  Control Vector Translate verb 111
  Prohibit Export Extended verb 184
KEK_key_identifier_1 parameter
  Key Generate verb 134
KEK_key_identifier_2 parameter
  Key Generate verb 134
key cache, host side 8
Key Export (CSNBKEX) 127
  format 127
  JNI version 129
  overview 33
  parameters 127
  required commands 128
  restrictions 128
  usage notes 129
key form 43
  combinations for a key pair 136
  combinations with key type 136
  definition 22
  exportable 22
  importable 22
  operational 22
key form (continued)
  value 131
key form bits 466
key formats 429
key formatting 516
key forms 33
Key Generate (CSNBKGN) 130
  format 130
  JNI version 137
  overview 33
  parameters 130
  required commands 135
  usage notes 136
  using 43
key identifier 16, 17
  definition 58
  PKA 58
key identifier parameter
  Clear Key Import verb 106
Key Import (CSNBKIM) 138
  format 138
  JNI version 140
  overview 33
  parameters 138
  required commands 140
  restrictions 140
  usage notes 140
key label 16, 17, 58, 233, 234
key management
  PKA 57
key pair 136
key pair generation 506
Key Part Import (CSNBKPI) 142
  format 142
  JNI version 145
  overview 34
  parameters 142
  required commands 144
  restrictions 144
key record 35, 36
key storage
  environment variables 234
  Linux on IBM System z 235
key storage file 35, 238, 551
Key Storage Initialization (CSNBKSI) 95
  format 95
  JNI version 97
  overview 32
  parameters 95
  required commands 97
key storage verbs 50, 233, 239
Key Test (CSNBKYT) 146
  format 147
  JNI version 150
  overview 34
  parameters 147
  required commands 150
  usage notes 150
Key Test Extended (CSNBKYTX) 151
  format 152
  JNI version 155
Key Test Extended (CSNBKYTX) (continued)
  overview 34
  parameters 152
  required commands 155
  restrictions 155
  usage notes 155
key token 16, 17
  AES 429
  definition 23
  DES
    external 432
    internal 429
    null 433
  DES internal 430, 431
    external 16, 23
    internal 16, 23, 59
    null 24
    operational 16
  PKA 56
    null 443
    RSA 1024-bit modulus-exponent private
      external 435
    RSA 1024-bit private internal 439, 440
    RSA 2048-bit Chinese remainder theorem private
      external 436
    RSA 2048-bit Chinese remainder theorem private
      internal 441
    RSA private 434, 435, 436
    RSA private external 434
    RSA private internal 438
    RSA public 433
    variable modulus-exponent token 443
  PKA external 59
  verbs 23
Key Token Build (CSNBKTB) 157
  format 157
  JNI version 160
  overview 34
  parameters 157
  usage notes 160
Key Token Change (CSNBKTC) 161
  format 161
  JNI version 163
  overview 34
  parameters 161
  required commands 163
Key Token Parse (CSNBKTP) 164
  format 164
  JNI version 167
  overview 34
  parameters 164
  usage notes 167
key token sections 56
Key Translate (CSNBKTR) 169
  format 169
  JNI version 170
  overview 34
  parameters 169
  required commands 170
  restrictions 170
  key type 21, 43
key type 1 44, 45, 46
key type 2 44, 45, 46
key_check_parameters parameter
  Remote Key Export verb 405
key_check_parameters_length parameter
  Remote Key Export verb 405
key_check_value parameter
  Remote Key Export verb 405
key_check_value_length parameter
  Remote Key Export verb 405
key_encrypting_key_identifier parameter
  PKA Symmetric Key Generate verb 391
  Secure Messaging for Keys verb 331
key_form parameter
  Key Generate verb 130
key_generation_data parameter
  PIN Change/Unblock verb 326
key_generation_data_length parameter
  PIN Change/Unblock verb 326
key_identifier parameter
  Clear Key Import verb 106
  Decipher verb 194
  Diversified Key Generate verb 125
  Encipher verb 200
  Key Part Import verb 144
  Key Test Extended verb 154
  Key Test verb 149
  Key Token Change verb 163
  MAC Generate verb 220
  MAC Verify verb 225
  Multiple Clear Key Import verb 172
  PKA Key Token Change verb 378
  Prohibit Export verb 182
  Symmetric Algorithm Decipher verb 206
  Symmetric Algorithm Encipher verb 213
key_identifier_length parameter
  Multiple Clear Key Import verb 172
  PKA Key Token Change verb 378
  Symmetric Algorithm Decipher verb 206
  Symmetric Algorithm Encipher verb 213
key_label parameter
  AES Key Record Create verb 240
  AES Key Record Delete verb 243
  AES Key Record List verb 245
  AES Key Record Read verb 248
  AES Key Record Write verb 250
  DES Key Record Create verb 252
  DES Key Record Delete verb 255
  DES Key Record List verb 257
  DES Key Record Read verb 259
  DES Key Record Write verb 261
  PKA Key Record List verb 268
  Retained Key Delete verb 276
key_label_mask parameter
  Retained Key List verb 279
key_labels parameter
  Retained Key List verb 279
key_labels_count parameter
  Retained Key List verb 279
key_length parameter
  Key Generate verb 132
key_offset parameter
  Secure Messaging for Keys verb 331
key_offset_field_length parameter
  Secure Messaging for Keys verb 332
key_parms parameter
  Symmetric Algorithm Decipher verb 206
  Symmetric Algorithm Encipher verb 213
key_parms_length parameter
  Symmetric Algorithm Decipher verb 206
  Symmetric Algorithm Encipher verb 213
key_part parameter
  Key Part Import verb 144
  Master Key Process verb 100
key_storage_description parameter
  Key Storage Initialization verb 96
key_storage_description_length parameter
  Key Storage Initialization verb 96
key_storage_file_name parameter
  Key Storage Initialization verb 96
key_storage_file_name_length parameter
  Key Storage Initialization verb 96
key_token parameter
  AES Key Record Create verb 240
  AES Key Record Read verb 248
  AES Key Record Write verb 250
  DES Key Record Read verb 259
  DES Key Record Write verb 261
  Key Token Build verb 158
  Key Token Parse verb 165
  PKA Key Token Build verb 376
key_token_length parameter
  AES Key Record Create verb 240
  AES Key Record Read verb 248
  AES Key Record Write verb 250
  PKA Key Token Build verb 375
key_type parameter
  Control Vector Generate verb 108
  Key Export verb 127
  Key Import verb 139
  Key Token Build verb 158
  Key Token Parse verb 165
key_type_1 parameter
  Key Generate verb 133
key_type_2 parameter
  Key Generate verb 133
key_value parameter
  Key Token Build verb 159
  Key Token Parse verb 166
key_value_structure parameter
  PKA Key Token Build verb 372
key_value_structure_length parameter
  PKA Key Token Build verb 372
key_encrypting_key 25
description 25
exporter 118, 127
key_encrypting_key_variant
definition 22
key_generating_keys
definition 26
key-half processing 473
key-storage initialization 95
key-token verification patterns 494
key-verification 493
KEYGENKY key type 26
keys
  AES master key 25
  asymmetric master key 25
  CIPHER 25
clear 29
control vector 21, 24
data key
  exporting 118
  importing 105
  re-enciphering 118
data-encrypting 25
DECIPHER 25
double-length 45, 46
ENCIPHER 25
exporter key-encrypting 26
forms 22
generating
  encrypted 130
  values for keys 35
importer key-encrypting 26
key-encrypting 25
MAC 25
multiple decipherment/encipherment 507
NOCV Importers and Exporters 26
pair 44, 45, 46
parity 105
PIN 25
PIN-encrypting key 313
protecting 191
re-enciphered 138
re-enciphering 127
separation 21
single-length 44
symmetric master key 24
transport 25
types of 24
VISA PVV
  generating 295
keys type
  list 26
  specified by rule_array 28
keyvalue parameter
  PKA Encrypt verb 179
keyvalue_length parameter
  PKA Encrypt verb 179

L
label parameter
  PKA Key Record Create verb 264
  PKA Key Record Delete verb 266
  PKA Key Record Read verb 271
  PKA Key Record Write verb 273
legacy support 547
Linux
  mixed configurations 551
Linux distribution xx
loading a master key 98
local_enciphered_key_identifier parameter
  PKA Symmetric Key Generate verb 391
local_enciphered_key_identifier_length parameter
  PKA Symmetric Key Generate verb 391

M
MAC 25, 38, 44
  length keywords 220, 225
  managing 38
MAC Generate (CSNBMGN) 219
  format 219
  JNI version 222
  overview 39
  parameters 219
  related information 222
  required commands 222
  restrictions 222
MAC key type 26
MAC keys
  definition 25
mac parameter
  MAC Generate verb 221
  MAC Verify verb 226
MAC Verify
  methods 39
  overview 39
MAC Verify (CSNBMVR) 224
  format 224
  JNI version 227
  parameters 224
  related information 227
  required commands 227
  restrictions 227
  usage notes 227
MACVER 25, 38
MACVER key type 26
managing DES keys 105
mask array preparation 471
mask_array_left parameter
  Control Vector Translate verb 112
mask_array_right parameter
  Control Vector Translate verb 112
master key 6, 237, 551
  changing
    possible effect on internal key tokens 23
    enciphered key 138
master key management 235
Master Key Process (CSNBMKP) 98
  format 98
  JNI version 102
  overview 32
  parameters 99
  Questionable DES keys 101
  required commands 100
  restrictions 100
master key variant
  definition 21
master_key_verification_pattern parameter
  Key Token Parse verb 167
master-key loading 95, 98
master-key verification 493
master-key-splitting 507
MasterCard card-verification code (CVC) 40, 281
masterkey_verify_parm parameter
  Key Token Build verb 159
MD5 38, 39
MDC Generate (CSNBMDG) 352
  format 353
  JNI version 355
  overview 54
  parameters 353
  related information 355
  required commands 355
  restrictions 355
MDC keyed hash 355
MDC parameter
  MDC Generate verb 354
message authentication
  definition 38, 39
message authentication code (MAC) 38
  description 217
  generating 217, 219
  verifying 217, 224
message authentication code (MAC) calculation
  method 504
messages
  authenticating 217
MKVP parameter
  Key Token Parse verb 166
  modes of operation 191
modification detection code (MDC) 38, 54
  generating 218
  verifying 218
Modification Detection Code (MDC) 495
multi-coprocessor functions 30
multiple
  decipherment 507
  encipherment 507
Multiple Clear Key Import (CSNBCKM) 171
  format 171
  JNI version 172
  overview 34
  parameters 171
  required commands 172
  usage notes 172

O
OAEP 517
object protection key (OPK) 456
OCV (output chaining value) 498
OKEYXLAT 26
OKEYXLAT key type 26
One-Way Hash (CSNBOWH) 229
  format 229
  JNI version 231
  overview 39
  parameters 229
  usage notes 231
OP key form 44
operational key
  generating 43
operational key form
  definition 22
  value 131
operational key token 16
operational private keys 53
OPEX key form 45
OPIM key form 44
OPINENC key type 26, 314
OPK, object protection key 456
OPOP key form 44
optional_data_parameter
  Symmetric Algorithm Decipher verb 208
  Symmetric Algorithm Encipher verb 215
optional_data_length parameter
  Symmetric Algorithm Decipher verb 208
  Symmetric Algorithm Encipher verb 215
outbound_PIN_encrypting_key_identifier parameter
  Encrypted PIN Generate verb 309
output chaining value (OCV) 498
output chaining vector (OCV)
  description 192
output_chaining_vector_parameter
  Secure Messaging for Keys verb 332
  Secure Messaging for PINs verb 337
output_KEYK_key_identifier parameter
  Key Translate verb 170
output_key_token parameter
  Key Translate verb 170
output_PAN_data parameter
  Secure Messaging for PINs verb 336
output_PIN_data parameter
  PIN Change/Unblock verb 328
output_PIN_data_length parameter
  PIN Change/Unblock verb 328
output_PIN_encryption_key_identifier parameter
  Encrypted PIN Translate verb 314
output_PIN_message parameter
  PIN Change/Unblock verb 328

N
new master-key register 17
new_reference_PIN_block parameter
  PIN Change/Unblock verb 326
new_reference_PIN_key_identifier parameter
  PIN Change/Unblock verb 326
new_reference_PIN_key_length parameter
  PIN Change/Unblock verb 326
new_reference_PIN_PAN_data parameter
  PIN Change/Unblock verb 327
new_reference_PIN_profile parameter
  PIN Change/Unblock verb 327
NOCV 22
NOCV Importers and Exporters 26
non-repudiation 3
Notices 557
null key token 58
  definition 24
  format 433, 443
number, generated 186, 188, 398

574 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
output_PIN_message_length parameter  
PIN Change/Unblock verb 328

output_PIN_profile parameter  
Encrypted PIN Translate verb 316
PIN Change/Unblock verb 328
Secure Messaging for PINs verb 336

overlapped processing restrictions 8

P

pad digit 286
format 286
pad_character parameter
Encipher verb 201
PADDIGIT PIN extraction method keyword 285
padding schemes 199
PADEXIST PIN extraction method keyword 285
pair of keys 44, 45, 46
PAN_data parameter
  Clear PIN Encrypt verb 289
  Clear PIN Generate Alternate verb 296
  CVV Generate verb 301
  CVV Verify verb 305
  Encrypted PIN Generate verb 310
  Encrypted PIN Verify verb 320
PAN_data_in parameter
Encrypted PIN Translate verb 314
PAN_data_out parameter
Encrypted PIN Translate verb 317
pan.exe
authorization 552
functions 551
functions not supported 553
utility 12, 24, 25, 30, 31, 65, 238, 540, 542, 545, 551, 552, 553, 554, 555
parameter
AES Key Record list verb 245
parity of key 105
EVEN
  form parameter 186
ODD
  form parameter 186
personal account number (PAN) 40
  for Encrypted PIN Translate 314
  for Encrypted PIN Verify 320
personal identification number (PIN)  
(continued)
extraction rules 480
formats 40
GBP PIN verification algorithm 486
generating 282, 291
  from encrypted PIN block 282
German Banking Pool PIN algorithm 482
Interbank PIN generation algorithm 490
keys 25
managing 40
PIN offset generation algorithm 483
PVV generation algorithm 489
PVV verification algorithm 490
translating 282, 283
translation of, in networks 282
translation verb 41, 313
using 281
verification verb 319
verifying 282, 319
VISA PIN algorithm 489
PIN block 282
PIN block format
3621 479
3624 479
additional names 318
ANSI X9.8 477
detail 477
ECI-2 479
ECI-3 480
IBM-4700 479
ISO-1 478
ISO-2 478
PIN extraction method keywords 284
values 284
VISA-2 479
VISA-3 479
PIN Change/Unblock (CSNBPCU) 324
format 324
JNI version 329
overview 41
parameters 324
required commands 328
usage notes 329
PIN keys 25
PIN notation 477
PIN profile 284
description 314, 320
PIN validation value (PVV) 282, 291
PIN verbs 282
PIN_encrypting_key_identifier parameter
Clear PIN Encrypt verb 288
Secure Messaging for PINs verb 335

Index 575
PIN_encryption_key_identifier parameter
Clear PIN Generate Alternate verb 295
PIN_generating_key_identifier parameter
Clear PIN Generate verb 291
Encrypted PIN Generate verb 309
PIN_generation_key_identifier parameter
Clear PIN Generate Alternate verb 296
PIN_length parameter
Clear PIN Generate verb 292
Encrypted PIN Generate verb 309
PIN_offset parameter
Secure Messaging for PINs verb 336
PIN_offset_field_length parameter
Secure Messaging for PINs verb 336
PIN_profile parameter
Clear PIN Encrypt verb 289
Clear PIN Generate Alternate verb 296
Encrypted PIN Generate verb 310
PIN_verifying_key_identifier parameter
Encrypted PIN Verify verb 320
PINBLOCK PIN extraction method keyword 285
PINGEN 44
PINGEN key 44
PINGEN key type 26
PINLEN04 PIN extraction method keyword 285
PINLEN12 PIN extraction method keyword 285
PINVER key 44
PINVER key type 26
PKA cryptographic key verbs 363
PKA cryptography 53
PKA Decrypt (CSNDPKD) 174
format 174
JNI version 176
overview 37
parameters 174
required commands 176
restrictions 176
usage notes 176
PKA Encrypt (CSNDPKE) 178
format 178
JNI version 180
overview 37
parameters 178
required commands 180
restrictions 180
usage notes 180
PKA external key token 59
PKA Key Generate (CSNDPKG) 363
format 363
JNI version 366
overview 54
parameters 364
required commands 366
PKA Key Import (CSNDPKI) 367
format 367
JNI version 369
overview 55
parameters 367
required commands 368
restrictions 368
usage notes 368
PKA key management 57
PKA key management verbs 54
PKA Key Record Create
related information 264
PKA Key Record Create (CSNDKRC) 263
format 263
JNI version 264
overview 36
parameters 263
required commands 264
PKA Key Record Delete (CSNDKRD) 265
format 265
JNI version 266
overview 36
parameters 265
related information 266
required commands 266
PKA Key Record List (CSNDKRL) 267
format 267
JNI version 269
overview 36
parameters 267
related information 269
required commands 269
PKA Key Record Read (CSNDKRR) 270
format 270
JNI version 271
overview 36
parameters 270
related information 271
required commands 271
PKA Key Record Write (CSNDKRW) 272
format 272
JNI version 273
overview 37
parameters 272
related information 273
required commands 273
PKA key storage 233
PKA key storage file 36
PKA key token 56
external 59
record format
PKA Key Token Build (CSNDPKB) 370
format 370
JNI version 376
overview 55
parameters 370
PKA Key Token Change (CSNDKTC) 377
PKA Key Token Change (CSNDKTC)  (continued)
format 377
JNI version 378
overview 55
parameters 377
required commands 378
PKA Key Translate (CSNDPKT) 380
format 380
JNI version 383
overview 55
parameters 380
required commands 382
restrictions 382
usage notes 382
PKA master key 53
PKA Public Key Extract (CSNDPKX) 384
format 384
JNI version 385
overview 55
parameters 384
usage notes 385
PKA Symmetric Key Export (CSNDSYX) 386
format 386
JNI version 388
overview 55
parameters 386
required commands 388
usage notes 388
PKA Symmetric Key Generate (CSNDSYG) 389
format 389
JNI version 392
overview 55
parameters 389
required commands 392
usage notes 392
PKA Symmetric Key Import (CSNDSYI) 394
format 394
JNI version 396
overview 56
parameters 394
required commands 396
restrictions 396
usage notes 396
PKA verbs 54, 60
PKA_enciphered_keyvalue parameter
    PKA Decrypt verb 175
    PKA Encrypt verb 180
PKA_enciphered_keyvalue_length parameter
    PKA Decrypt verb 175
    PKA Encrypt verb 179
PKA_key_identifier parameter
    PKA Decrypt verb 175
    PKA Encrypt verb 179
PKA_key_identifier_length parameter
    PKA Decrypt verb 175
    PKA Encrypt verb 179
PKA_private_key_identifier parameter
    Digital Signature Generate verb 345
PKA_public_key_identifier parameter
    Digital Signature Verify verb 349
PKA_public_key_identifier_length parameter
    Digital Signature Verify verb 349
PKA92 key format and encryption process 514
PKCS #1 formats 516
PKCS-PAD processing rule 206, 212
plaintext
    enciphering 191
plaintext parameter
    Cryptographic Variable Encipher verb 116
privacy 37
private external key token
    RSA 434
private internal key token
    RSA 438, 439, 440
private key token
    RSA 434, 435, 436
private_key_name parameter
    PKA Key Token Build verb 374
private_key_name_length parameter
    PKA Key Token Build verb 374
problems, reporting xxiii
procedure calls 13
processing a master key 98
processing overlap 8
processing rule
    ANSI X9.23 192, 196, 201
    CBC 191, 192, 195, 201, 205, 212
    CUSP 195, 201
    Decipher 195
description 191, 192
    ECB 205, 212
    Encipher 201
    GBP-PIN 292
    IBM-PIN 292
    IBM-PINO 292
    INBK-PIN 292
    IPS 196, 201
    PKCS-PAD 206, 212
    recommendations for Encipher 201
    Symmetric Algorithm Decipher 205
    Symmetric Algorithm Encipher 212
    VISA-PVV 292
processing rules
    Decipher 193
    Encipher 198
    Symmetric Algorithm Decipher 203
    Symmetric Algorithm Encipher 210
Prohibit Export (CSNBPEX) 182
format 182
JNI version 182
overview 34
parameters 182
required commands 182
Prohibit Export Extended (CSNBPEX)
overview 34
Prohibit Export Extended (CSNBPEXX) 184
format 184
JNI version 185
parameters 184
Prohibit Export Extended (CSNBPEXX) *(continued)*
required commands 185
restrictions 185
protecting data and keys 191
pseudonyms 13
public key cryptography 53
public key token
RSA 433

R
Random Number Generate (CSNBRNG) 186
format 186
JNI version 186
overview 35
parameters 186
required commands 187
Random Number Generate Long (CSNBRNGL) 188
format 188
JNI version 189
parameters 188
Random Number Generate Long (CSNBRNG) overview 35
Random Number Generate Long (CSNBRNGL) overview 35
Random Number Tests (CSUARNT) 103
format 103
JNI version 104
overview 32
parameters 103
random_number parameter
Key Test Extended verb 154
Key Test verb 149
Random Number Generate Long verb 189
Random Number Generate verb 187
random_number_length parameter
Random Number Generate Long verb 189
re-enciphered
key 138
re-enciphering
data-encrypting key 118
PIN block 313
reason code 413
reason codes
with return code 0 414
with return code 12 425
with return code 16 426
with return code 4 414
with return code 8 415
reason_code parameter *(continued)*
Cryptographic Resource Allocate verb 89
Cryptographic Resource Deallocate verb 92
Cryptographic Variable Encipher verb 115
CVV Generate verb 300
CVV Verify verb 304
Data Key Export verb 118
Data Key Import verb 120
Decipher verb 194
DES Key Record Create verb 252
DES Key Record Delete verb 254
DES Key Record List verb 256
DES Key Record Read verb 259
DES Key Record Write verb 261
Digital Signature Generate verb 344
Digital Signature Verify verb 348
Diversified Key Generate verb 123
Encipher verb 199
Encrypted PIN Generate verb 308
Encrypted PIN Translate verb 313
Encrypted PIN Verify verb 319
Key Export verb 127
Key Generate verb 130
Key Import verb 138
Key Part Import verb 142
Key Storage Initialization verb 95
Key Test Extended verb 152
Key Test verb 147
Key Token Build verb 157
Key Token Change verb 161
Key Token Parse verb 164
Key Translate verb 169
MAC Generate verb 219
Master Key Process verb 99
MCA Verify verb 224
MDC Generate verb 353
Multiple Clear Key Import verb 171
One-Way Hash verb 229
PIN Change/Unblock verb 325
PKA Decrypt verb 174
PKA Encrypt verb 178
PKA Key Generate verb 364
PKA Key Import verb 367
PKA Key Record Create verb 263
PKA Key Record Delete verb 265
PKA Key Record List verb 267
PKA Key Record Read verb 270
PKA Key Record Write verb 272
PKA Key Token Build verb 370
PKA Key Token Change verb 377
PKA Key Translate verb 380
PKA Key Token Read verb 384
PKA Symmetric Key Export verb 386
PKA Symmetric Key Generate verb 389
PKA Symmetric Key Import verb 394
Prohibit Export Extended verb 184
Prohibit Export verb 182
Random Number Generate Long verb 188
Random Number Generate verb 186
Random Number Tests verb 103
Remote Key Export verb 400

578 Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
reason_code parameter (continued)
Retained Key Delete verb 275
Retained Key List verb 278
Secure Messaging for Keys verb 330
Secure Messaging for PINs verb 334
Symmetric Algorithm Decipher verb 204
Symmetric Algorithm Encipher verb 211
Transaction Validation verb 338
Trusted Block Create verb 409
recommendations for Encipher processing rules 201
record chaining 192
regeneration_data parameter
PKA Key Generate verb 365
regeneration_data_length parameter
PKA Key Generate verb 365
related publications xxiii
Remote Key Export (CSNDRKX) 398
format 400
JNI version 406
overview 56
parameters 400
restrictions 406
remote key-loading 445
remote keys
export 56
reserved parameter
Control Vector Generate verb 109
reserved_1 parameter
PKA Key Token Build verb 375
reserved_1_length parameter
PKA Key Token Build verb 375
reserved_2 parameter
Key Token Parse verb 166
PKA Key Token Build verb 375
reserved_2_length parameter
PKA Key Token Build verb 375
reserved_3 parameter
Key Token Parse verb 167
PKA Key Token Build verb 375
reserved_3_length parameter
PKA Key Token Build verb 375
reserved_4 parameter
Key Token Parse verb 167
PKA Key Token Build verb 375
reserved_4_length parameter
PKA Key Token Build verb 375
reserved_5 parameter
Key Token Parse verb 166
PKA Key Token Build verb 375
reserved_5_length parameter
PKA Key Token Build verb 375
reserved_6 parameter
Key Token Parse verb 167
resource_name parameter
Cryptographic Resource Allocate verb 90
Cryptographic Resource Deallocate verb 93
resource_name_length parameter
Cryptographic Resource Allocate verb 90
Cryptographic Resource Deallocate verb 93
Retained Key Delete (CSNDRKD) 275
Retained Key Delete (CSNDRKD) (continued)
format 275
JNI version 277
overview 36
parameters 275
related information 277
required commands 276
Retained Key List (CSNDRKL) 278
format 278
JNI version 279
overview 36
parameters 278
related information 279
required commands 279
retained keys 276
retained_keys_count parameter
Retained Key List verb 279
return code 413
return_code parameter 15, 16
AES Key Record Create verb 239
AES Key Record Delete verb 242
AES Key Record List verb 244
AES Key Record Read verb 247
AES Key Record Write verb 249
Clear Key Import verb 106
Clear PIN Encrypt verb 288
Clear PIN Generate Alternate verb 295
Clear PIN Generate verb 291
Control Vector Generate verb 108
Control Vector Translate verb 111
Cryptographic Facility Query verb 66
Cryptographic Facility Version verb 87
Cryptographic Resource Allocate verb 89
Cryptographic Resource Deallocate verb 92
Cryptographic Variable Encipher verb 115
CVV Generate verb 300
CVV Verify verb 304
Data Key Export verb 118
Data Key Import verb 120
Decipher verb 194
DES Key Record Create verb 252
DES Key Record Delete verb 254
DES Key Record List verb 256
DES Key Record Read verb 259
DES Key Record Write verb 261
Digital Signature Generate verb 344
Digital Signature Verify verb 348
Diversified Key Generate verb 122
Encipher verb 199
Encrypted PIN Generate verb 308
Encrypted PIN Translate verb 313
Encrypted PIN Verify verb 319
Key Export verb 127
Key Generate verb 130
Key Import verb 138
Key Part Import verb 142
Key Storage Initialization verb 95
Key Test Extended verb 152
Key Test verb 147
Key Token Build verb 157
Key Token Change verb 161
return_code parameter (continued)
  Key Token Parse verb 164
  Key Translate verb 169
  MAC Generate verb 219
  Master Key Process verb 99
  MCA Verify verb 224
  MDC Generate verb 353
  Multiple Clear Key Import verb 171
  One-Way Hash verb 229
  PIN Change/Unblock verb 324
  PKA Decrypt verb 174
  PKA Encryp verb 178
  PKA Key Generate verb 364
  PKA Key Import verb 367
  PKA Key Record Create verb 263
  PKA Key Record Delete verb 265
  PKA Key Record List verb 267
  PKA Key Record Read verb 270
  PKA Key Record Write verb 272
  PKA Key Token Build verb 370
  PKA Key Token Change verb 377
  PKA Key Translate verb 380
  PKA Public Key Extract verb 384
  PKA Symmetric Key Export verb 386
  PKA Symmetric Key Generate verb 389
  PKA Symmetric Key Import verb 394
  Prohibit Export Extended verb 184
  Prohibit Export verb 182
  Random Number Generate Long verb 188
  Random Number Generate verb 186
  Random Number Tests verb 103
  Remote Key Export verb 400
  Retained Key Delete verb 275
  Retained Key List verb 278
  Secure Messaging for Keys verb 330
  Secure Messaging for PINs verb 334
  Symmetric Algorithm Encipher verb 204
  Symmetric Algorithm Decipher verb 211
  Transaction Validation verb 338
  Trusted Block Create verb 409

RSA_enciphered_key parameter
  PKA Symmetric Key Export verb 387
  PKA Symmetric Key Generate verb 392
  PKA Symmetric Key Import verb 395

RSA_enciphered_key_length parameter
  PKA Symmetric Key Export verb 387
  PKA Symmetric Key Generate verb 391
  PKA Symmetric Key Import verb 395

RSA_private_key_identifier parameter
  PKA Symmetric Key Export verb 387
  PKA Symmetric Key Generate verb 391

RSA_private_key_identifier_length parameter
  PKA Symmetric Key Export verb 387

rule_array parameter 16
  AES Key Record Create verb 240
  AES Key Record Delete verb 243
  AES Key Record list verb 245
  AES Key Record Read verb 247
  AES Key Record Write verb 250
  Clear PIN Encrypt verb 288
  Clear PIN Generate Alternate verb 296
  Clear PIN Generate verb 292
  Control Vector Generate verb 109
  Control Vector Translate verb 112
  Cryptographic Facility Query verb 67
  Cryptographic Resource Allocate verb 90
  Cryptographic Resource Deallocate verb 93
  CVV Generate verb 301
  CVV Verify verb 305
  Decipher verb 195
  DES Key Record Delete verb 254
  Digital Signature Generate verb 344
  Digital Signature Verify verb 349
  Diversified Key Generate verb 123
  Encipher verb 200
  Encrypted PIN Generate verb 309
  Encrypted PIN Translate verb 315
  Encrypted PIN Verify verb 321
  Key Part Import verb 143
  Key Storage Initialization verb 96
  Key Test Extended verb 153
  Key Test verb 148
  Key Token Build verb 158
  Key Token Change verb 162
  Key Token Create verb 165
  MAC Generate verb 220
  MAC Verify verb 225
  Master Key Process verb 99
  MDC Generate verb 354
  Multiple Clear Key Import verb 171
  One-Way Hash Import verb 171
  PIN Change/Unblock verb 325
  PKA Decrypt verb 175
  PKA Encrypt verb 178

580  Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
security_server_name parameter  (continued)
   DES Key Record List verb  257
   PKA Key Record List verb  269
seed parameter
   Random Number Generate Long verb  189
seed_length parameter
   Random Number Generate Long verb  189
segmenting
   control keywords  220, 225
selecting a coprocessor resource  89, 92
sequence_number parameter
   Clear PIN Encrypt  289
   Encrypted PIN Generate verb  311
   Encrypted PIN Translate verb  317
sequences of verbs  43
service_code parameter
   CVV Generate verb  302
   CVV Verify verb  305
SHA-1  38, 39
short blocks  198
signature_bit_length parameter
   Digital Signature Generate verb  346
signature_field parameter
   Digital Signature Generate verb  346
   Digital Signature Verify verb  350
signature_field_length parameter
   Digital Signature Generate verb  345
   Digital Signature Verify verb  350
single-length key
   multiple decipherment  509
   multiple encipherment  508
   purpose  44
   using  26
skeleton_key_identifier parameter
   PKA Key Generate verb  365
skeleton_key_identifier_length parameter
   PKA Key Generate verb  365
SNA-SLE  499
source_key_identifier parameter
   Data Key Export verb  118
   Key Export verb  128
   Key Import verb  139
   PKA Key Import verb  368
   PKA Key Translate verb  381
   PKA Public Key Extract verb  385
   PKA Symmetric Key Export verb  387
   Remote Key Export verb  403
source_key_identifier_length parameter
   PKA Key Import verb  368
   PKA Key Translate verb  381
   PKA Public Key Extract verb  385
   PKA Symmetric Key Export verb  387
   Remote Key Export verb  403
source_key_token parameter
   Control Vector Translate verb  111
   Data Key Import verb  120
   Prohibit Export Extended verb  184
source_transport_key_identifier parameter
   PKA Key Translate verb  381
source_transport_key_identifier_length parameter
   PKA Key Translate verb  381
SSL support  37
STATCARD  30
STATICS A  79
STATICSE  81
STATICSX  83
STATKPR  78
STATKPR L  78
sym_encrypted_key_identifier parameter
   Remote Key Export verb  404
sym_encrypted_key_identifier_length parameter
   Remote Key Export verb  404
SYM-MK  24
Symmetric Algorithm Decipher (CSNBSAD)  203
   format  204
   JNI version  208
   overview  37
   parameters  204
   required commands  208
   restrictions  208
Symmetric Algorithm Encipher (CSNBSAE)  210
   format  211
   JNI version  215
   overview  38
   parameters  211
   required commands  215
   restrictions  215
Symmetric Key Encipher/Decipher - Encrypted AES keys  10
Symmetric Key Encipher/Decipher - Encrypted DES keys  9
symmetric keys master key  24
sysfs interface  30, 31, 536

T

table of contents  ii
target_key_identifier parameter
   Data Key Export verb  119
   Data Key Import verb  121
   Key Export verb  128
   Key Import verb  139
   PKA Key Import verb  368
   PKA Symmetric Key Import verb  396
target_key_identifier_length parameter
   PKA Key Import verb  368
   PKA Symmetric Key Import verb  395
target_key_token parameter
   Control Vector Translate verb  113
   PKA Key Translate verb  382
target_key_token_length parameter
   PKA Key Translate verb  382
target_keyvalue parameter
   PKA Decrypt verb  176
target_keyvalue_length parameter
   PKA Decrypt verb  176
target_public_key_token parameter
   PKA Public Key Extract verb  385
target_public_key_token_length parameter
   PKA Public Key Extract verb  385
target_transport_key_identifier parameter
   PKA Key Translate verb  382

582  Linux for System z: Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide
target_transport_key_identifier_length parameter

PKA Key Translate verb 381
TDES 501
terminology xx
text parameter
MAC Generate verb 220
MAC Verify verb 225
MDC Generate verb 353
One-Way Hash verb 230
text_length parameter
Cryptographic Variable Encipher verb 116
Decipher verb 195
Encipher verb 200
MAC Generate verb 220
MAC Verify verb 225
MDC Generate verb 353
One-Way Hash verb 230
Secure Messaging for Keys verb 331
Secure Messaging for PINs verb 336
TKE workstation 24, 25
token parameter
PKA Key Record Create verb 264
PKA Key Record Read verb 271
PKA Key Record Write verb 273
token validation value (TVV) 430
token_length parameter
PKA Key Record Create verb 264
PKA Key Record Read verb 271
PKA key record write verb 273
trailing short blocks 198
Transaction Validation (CSNBTRV) 338
format 338
JNI version 340
overview 42
parameters 338
required commands 340
usage notes 340
transaction_info parameter
Transaction Validation verb 339
transaction_info_length parameter
Transaction Validation verb 339
transaction_key_identifier parameter
Transaction Validation verb 339
transaction_key_identifier_length parameter
Transaction Validation verb 339
transport key 22, 25
transport key variant
definition 22
transport_key_identifier parameter
PKA Key Generate verb 365
Remote Key Export verb 402
Trusted Block Create verb 410
transport_key_identifier_length parameter
Remote Key Export verb 402
triple DES encryption 191
triple-DES 501
triple-length keys
multiple encipherment 512
multiple decipherment 513
trusted block 444
create 56
trusted block (continued)
number representation 446
section format 446
Trusted Block Create (CSNDTBC) 408
format 409
JNI version 411
overview 56
parameters 409
required commands 411
restrictions 411
trusted block integrity 446
trusted block sections 445
Trusted Key Entry (TKE) 53, 54, 523, 551
overview 42
trusted_block_identifier parameter
Remote Key Export verb 401
Trusted Block Create verb 411
trusted_block_identifier_length parameter
Remote Key Export verb 401
Trusted Block Create verb 411
types of keys 24

U
UKPT

format 287
ule_id_length parameter
Remote Key Export verb 403
using retained keys 276
utilities
ivp.e 551
Key Token Build 157
panel.exe 12, 24, 25, 30, 31, 65, 238, 540, 542, 545, 551, 552, 553, 554, 555
PKA Key Token Build 370

V
validation_values parameter
Transaction Validation verb 340
validation_values_length parameter
Transaction Validation verb 339
variable modulus-exponent token
RSA 443
variable types 15
verb
AES Key Record Create (CSNBAKRC) 239
AES Key Record Delete (CSNBAKRD) 242
AES Key Record List (CSNBAKRL) 244
AES Key Record Read (CSNBAKRR) 247
AES Key Record Write (CSNBAKRW) 249
Clear Key Import (CSNBCKI) 105
Clear PIN Encrypt (CSNCP) 287
Clear PIN Generate (CSNPIN) 291
Clear PIN Generate Alternate (CSNPIN) 295
Control Vector Generate (CSNBCVG) 108
Control Vector Translate (CSNBCVT) 111
Cryptographic Facility Query (CSUACFQ) 65
Cryptographic Facility Version (CSUACFV) 87
Cryptographic Resource Allocate (CSUACRA) 89
Cryptographic Resource Deallocate (CSUACRD) 92

Index 583
verb (continued)
Cryptographic Variable Encipher (CSNBCVE) 115
CVV Generate (CSNBCSG) 300
CVV Verify (CSNBCSV) 304
Data Key Export (CSNBDDX) 118
Data Key Import (CSNBDDKM) 120
Decipher (CSNBDEC) 193
definition 21
DES Key Record Create (CSNBKRC) 252
DES Key Record Delete (CSNBKRD) 254
DES Key Record List (CSNBKRL) 256
DES Key Record Read (CSNBKRR) 259
DES Key Record Write (CSNBKRW) 261
Digital Signature Generate (CSNDSSG) 343
Digital Signature Verify (CSNDSSV) 348
Diversified Key Generate (CSNBDKG) 122
Encipher (CSNBENC) 198
Encrypted PIN Generate (CSNBEPG) 308
Encrypted PIN Translate (CSNBPTR) 313
Encrypted PIN Verify (CSNBVPTR) 319
Key Export (CSNBEKEX) 127
Key Generate (CSNBKGN) 130
Key Import (CSNBKIM) 138
Key Part Import (CSNBKPI) 142
Key Storage Initialization (CSNBKSI) 95
Key Test (CSNBKTYT) 146
Key Test Extended (CSNBKTYTX) 151
Key Token Build (CSNBKTBM) 157
Key Token Change (CSNBKTC) 161
Key Token Parse (CSNBKTP) 164
Key Translate (CSNBKTR) 169
MAC Generate (CSNBMGN) 219
MAC Verify (CSNBMVR) 224
Master Key Process (CSNBMPK) 98
MDC Generate (CSNBMDG) 352
Multiple Clear Key Import (CSNBCKIM) 171
One-Way Hash (CSNBOWH) 229
PIN Change/Unblock (CSNBPCU) 324
PKA Decrypt (CSNPKDPK) 174
PKA Encrypt (CSNPKDPE) 178
PKA Key Generate (CSNPDPKG) 363
PKA Key Import (CSNPDPKI) 367
PKA Key Record Create (CSNPKDKC) 263
PKA Key Record Delete (CSNPKDRC) 265
PKA Key Record List (CSNPKDRL) 267
PKA Key Record Read (CSNPKDRR) 270
PKA Key Record Write (CSNPKDWR) 272
PKA Key Token Build (CSNPKDBK) 370
PKA Key Token Change (CSNPKDTC) 377
PKA Key Translate (CSNPKDPT) 380
PKA Public Key Export (CSNPKPE) 384
PKA Symmetric Key Export (CSNPSYX) 386
PKA Symmetric Key Generate (CSNPSYG) 389
PKA Symmetric Key Import (CSNPSYI) 394
Prohibit Export (CSNPBPEX) 182
Prohibit Export Extended (CSNPBPEXX) 184
Random Number Generate (CSNBRNG) 186
Random Number Generate Long (CSNBRNGL) 188
Random Number Tests (CSURSNT) 103
Remote Key Export (CSNPDRKX) 398
Retained Key Delete (CSNPDRKD) 275
verb (continued)
Retained Key List (CSNPDRKL) 278
Secure Messaging for Keys (CSNBTSKY) 330
Secure Messaging for PINs (CSNBTSPI) 334
sequences 43
Symmetric Algorithm Decipher (CSNBDSE) 203
Symmetric Algorithm Encipher (CSNBSAE) 210
Transaction Validation (CSNTRV) 338
Trusted Block Create (CSNTRBC) 408
verb_data parameter
Cryptographic Facility Query verb 77, 78
verb_data_length parameter
Cryptographic Facility Query verb 77
verbs 13
access control 519
CCA 63
CCA AES, DES, and hashing 46
CCA nodes and access control 46, 65
data integrity and authentication 219
DES cryptographic key 105
digital signature 54, 343
encipher and decipher 193
financial services 281, 287
input/output (I/O) parameters 14
key storage 50, 233, 239
parameters 14
PIN 282
PKA 53, 54, 60
PKA cryptographic key 363
PKA key management 54
prefixes 13
variable types 15
Verbs to encipher and decipher 193
verification pattern 493
verification pattern, generating and verifying 146, 151
verification_pattern parameter
Key Test Extended verb 154
Key Test verb 149
verifying data integrity and authenticity 217
version_data parameter
Cryptographic Facility Version verb 88
version_data_length parameter
Cryptographic Facility Version verb 87
VISA card-verification value (CVV) 40, 281
VISA PVV 291
generating 295
VISA-1 318
VISA-2 PIN block format 284, 479
VISA-3 PIN block format 284, 479
VISA-4 PIN block format 284
VISA-PVV algorithm 297, 321
VISAPV4 algorithm 321

W
who should use this document xx
X
X3.106 (CBC) method  498
X9.23 method  499
X9.23 padding scheme  194
X9.23 processing rules
  Decipher  193
  Encipher  198
X9.31 hash format  516
X9.9-1 keyword  220, 225

Z
z/OS
  mixed configurations  551
z/VM  xxi
z/VM guest  537
zcrypt
  installing  535
Readers’ Comments — We'd Like to Hear from You

Linux for System z
Secure Key Solution with the Common Cryptographic Architecture Application Programmer’s Guide

Publication No. SC33-8294-01

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.
Submit your comments using one of these channels:
• Send your comments to the address on the reverse side of this form.
• Send your comments via e-mail to: eservdoc@de.ibm.com

If you would like a response from IBM, please fill in the following information:

Name

Address

Company or Organization

Phone No.

E-mail address
IBM

Program Number:

Printed in USA