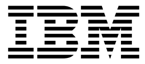

IBM SecureWay Cryptographic Products

SC31-8609-01

IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference And Guide

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xiii.

Second Edition (August, 1998)

This is the second edition of *IBM 4758 CCA Basic Services Reference and Guide*, SC31-8609-01.

This manual describes the IBM Common Cryptographic Architecture (CCA) Basic Services API that is implemented for the IBM 4758 Model 001 PCI Cryptographic Coprocessor and it's CCA Support Program feature. This manual revision describes the API provided with Release 1.3 of the CCA Support Program.

Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult your IBM representative to be sure you have the latest edition and any Technical Newsletter.

IBM does not stock publications at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office that serves your location. This and other publications related to the IBM 4758 Coprocessor can be obtained in PDF format from the Library page at http://www.ibm.com/security/cryptocards.

Reader’s comments can be communicated to IBM in these ways:

- On the form for reader’s comments provided at the back of this publication
- Comments can be addressed to the IBM Corporation, Department 57QC, MG81/204, 8501 IBM Drive, Charlotte, NC 28262-8563, U.S.A.
- On the question and suggestion form provided via the product Support web page that you can locate from http://www.ibm.com/security/cryptocards.

© **Copyright International Business Machines Corporation 1997-98. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . xiii
Trademarks . . . xiii

About This Publication . . . xv
| Revision History . . . xv
Organization . . . xix
Related Publications . . . xx

Chapter 1. Introduction to Programming for the IBM CCA . . . 1-1
What CCA Services Are Available with the IBM 4758 . . . 1-1
An Overview of the CCA Environment . . . 1-2
How Application Programs Obtain Service . . . 1-5
The Security API, Programming Fundamentals . . . 1-7
Verbs, Variables, and Parameters . . . 1-7
Commonly-Encountered Parameters . . . 1-9
Parameters Common to All Verbs . . . 1-10
Rule_Array and Other Keyword Parameters . . . 1-11
Key_Identifiers, Key_Labels, and Key_Tokens . . . 1-11
How the Verbs Are Organized in the Remainder of the Book . . . 1-12

Chapter 2. CCA Node Management and Access Control . . . 2-1
CCA Access Control . . . 2-2
Understanding Access Control . . . 2-2
Role-based Access Control . . . 2-2
Understanding Roles . . . 2-2
Understanding Profiles . . . 2-3
Initializing and Managing the Access Control System . . . 2-5
The Access Control Management and Initialization Verbs . . . 2-5
Permitting Changes to the Configuration . . . 2-6
Configuration and Greenwich Mean Time (GMT) . . . 2-6
Logging On and Logging Off . . . 2-7
Protecting Your Transaction Information . . . 2-7
| Understanding and Managing Master Keys . . . 2-8
Access_Control_Initialization(CSUAACI) . . . 2-13
Access_Control_Maintenance (CSUAACM) . . . 2-16
Cryptographic_Facility_Control (CSUACFC) . . . 2-22
Cryptographic_Facility_Query (CSUACFQ) . . . 2-26
| Key_Storage_Initialization (CSNBKSI) . . . 2-36
Logon_Control (CSUALCT) . . . 2-38
The use of Logon Context information . . . 2-40
| Master_Key_Distribution (CSUAMKD) . . . 2-42
Master_Key_Process (CSNBMKP) . . . 2-46

Chapter 3. RSA Key Administration . . . 3-1
RSA Key Management . . . 3-1
Key Generation . . . 3-2
Key Import . . . 3-4
Re-enciphering a Private Key Under an Updated Master Key . . . 3-4
Using the RSA Keys . . . 3-5
Using the Private Key at Multiple Nodes . . . 3-5

© Copyright IBM Corp. 1997-98

Registering and Retaining a Public Key . . . 3-5
PKA_Key_Generate (CSNDPKG) . . . 3-6
PKA_Key_Import (CSNDPKI) . . . 3-10
PKA_Key_Token_Build (CSNDPKB) . . . 3-12
PKA_Key_Token_Change (CSNDKTC) . . . 3-18
PKA_Public_Key_Extract (CSNDPKX) . . . 3-20
PKA_Public_Key_Hash_Register (CSNDPKH) . . . 3-22
PKA_Public_Key_Register (CSNDPKR) . . . 3-24

Chapter 4. Hashing and Digital Signatures . . . 4-1
Hashing . . . 4-1
Digital Signatures . . . 4-2
Digital_Signature_Generate (CSNDDSG) . . . 4-4
Digital_Signature_Verify (CSNDDSV) . . . 4-7
One_Way_Hash (CSNBOWH) . . . 4-10

Chapter 5. Basic CCA DES Key Management . . . 5-1
Understanding CCA DES Key Management . . . 5-2
Control Vectors . . . 5-3
Checking a Control Vector Before Processing a Cryptographic Command . 5-4
Key Types . . . 5-5
Key Usage Restrictions . . . 5-7
Key Tokens, Key Labels, and Key Identifiers . . . 5-8
Key Tokens . . . 5-8
Key Labels . . . 5-10
Key Identifiers . . . 5-10
Using the Key Processing and Key Storage Verbs . . . 5-11
Installing and Verifying Keys . . . 5-11
Generating Keys . . . 5-12
Exporting and Importing Keys . . . 5-14
Storing Keys in Key Storage . . . 5-15
Security Precautions . . . 5-15
Clear_Key_Import (CSNBCKI) . . . 5-16
Data_Key_Export (CSNBDKX) . . . 5-17
Data_Key_Import (CSNBDKM) . . . 5-18
Diversified_Key_Generate (CSNBDKG) . . . 5-20
Key_Export (CSNBKEX) . . . 5-23
Key_Generate (CSNBKGN) . . . 5-25
Key Type Specifications . . . 5-28
Key Length Specification . . . 5-29
Key_Import (CSNBKIM) . . . 5-31
Key_Part_Import (CSNBKPI) . . . 5-33
Key_Test (CSNBKYT) . . . 5-35
Key_Token_Build (CSNBKTB) . . . 5-38
Key_Token_Change (CSNBKTC) . . . 5-41
Key_Translate (CSNBKTR) . . . 5-43
Random_Number_Generate (CSNBRNG) . . . 5-45
PKA_Symmetric_Key_Export (CSNDSYX) . . . 5-47
PKA_Symmetric_Key_Generate (CSNDSYG) . . . 5-49
PKA_Symmetric_Key_Import (CSNDSYI) . . . 5-52

Chapter 6. Data Confidentiality and Data Integrity . . . 6-1
Encryption and Message Authentication Codes . . . 6-1
Ensuring Data Confidentiality . . . 6-1

Ensuring Data Integrity . . . 6-2
MACing Segmented Data . . . 6-3
Decipher (CSNBDEC) . . . 6-4
Encipher (CSNBENC) . . . 6-7
MAC_Generate (CSNBMGN) . . . 6-10
MAC_Verify (CSNBMVR) . . . 6-13

Chapter 7. Key Storage Verbs . . . 7-1
Key Labels and Key Storage Management . . . 7-1
Key Label Content . . . 7-2
DES_Key_Record_Create (CSNBKRC) . . . 7-4
DES_Key_Record_Delete (CSNBKRD) . . . 7-5
DES_Key_Record_List (CSNBKRL) . . . 7-7
DES_Key_Record_Read (CSNBKRR) . . . 7-9
DES_Key_Record_Write (CSNBKRW) . . . 7-10
PKA_Key_Record_Create (CSNDKRC) . . . 7-11
PKA_Key_Record_Delete (CSNDKRD) . . . 7-13
PKA_Key_Record_List (CSNDKRL) . . . 7-15
PKA_Key_Record_Read (CSNDKRR) . . . 7-17
PKA_Key_Record_Write (CSNDKRW) . . . 7-19
| Retained_Key_Delete (CSNDRKD) . . . 7-21
| Retained_Key_List (CSNDRKL) . . . 7-22

Chapter 8. Financial Services Support Verbs . . . 8-1
| Processing Financial PINs . . . 8-1
| PIN Verb Summary . . . 8-4
| PIN Calculation Method and PIN Block Format Summary . . . 8-5
| Providing Security for PINs . . . 8-5
| Using Specific Key Types and Key-Usage Bits to Help Ensure PIN Security . . . 8-6
| Supporting Multiple PIN Calculation Methods . . . 8-7
| PIN Calculation Methods . . . 8-7
| Data_Array . . . 8-7
| Supporting Multiple PIN-Block Formats and PIN Extraction Methods . . . 8-9
| PIN Profile . . . 8-9
| PIN Extraction Methods . . . 8-10
| Personal Account Number (PAN) . . . 8-11
| Clear_PIN_Encrypt (CSNBCPE) . . . 8-12
| Clear_PIN_Generate (CSNBPGN) . . . 8-15
| Clear_PIN_Generate_Alternate (CSNBCPA) . . . 8-18
| Encrypted_PIN_Generate (CSNBEPG) . . . 8-24
| Encrypted_PIN_Translate (CSNBPTR) . . . 8-29
| Encrypted_PIN_Verify (CSNBPVR) . . . 8-34
SET_Block_Compose (CSNDSBC) . . . 8-40
SET_Block_Decompose (CSNDSBD) . . . 8-44

Appendix A. Return Codes and Reason Codes . . . A-1
Return Codes . . . A-1
Reason Codes . . . A-1
Return Code 0 . . . A-2
Return Code 4 . . . A-3
Return Code 8 . . . A-5
Return Code 12 . . . A-13
Return Code 16 . . . A-14

Return Code 24 . . . . . A-15

Appendix B. Data Structures B-1
Key Tokens B-1
Master Key Verification Pattern B-1
Token-Validation Value and Record-Validation Value B-2
Null Key Token B-2
Internal DES Key Token B-3
External DES Key Token B-4
DES Key Token Flag Byte 1 . . . . . B-4
DES Key Token Flag Byte 2 . . . . . B-4
RSA Key Token Formats B-5
RSA Key Token Sections B-6
Chaining Vector Records B-13
Key Storage Records B-14
Key Record List Data Set B-16
Access Control Data Structures B-18
Role Structure B-18
Basic Structure of a Role B-18
Aggregate Role Structure B-19
The Access Control Point List B-19
Contents of the Default Role B-20
Profile Structure B-21
Basic Structure of a Profile B-21
Aggregate Profile Structure B-22
The Authentication Data Structure B-22
| Examples of the data structures B-25
| Passphrase authentication data B-25
| User profile B-26
| Aggregate profile structure B-27
| Access control point list B-27
| Role data structure B-28
| Aggregate role data structure B-29
| Master Key Shares Data Formats B-30
| Function Control Vector B-31

Appendix C. CCA Control Vector Definitions and Key Encryption C-1
DES Control Vector Values C-1
| Key Form Bits, 'fff' C-5
Specifying a Control Vector Base Value C-5
CCA Key Encryption and Decryption Process C-8
| CCA DES Key Encryption and Decryption Process C-8
CCA RSA Private Key Encryption and Decryption Process C-10
Changing Control Vectors C-11

Appendix D. Algorithms and Processes D-1
Cryptographic Key Verification Techniques D-1
Master Key Verification Algorithm D-1
DES Key Verification Algorithm D-1
| Encrypt Zeros DES Key Verification Algorithm D-2
Ciphering Methods D-3
ANSI X3.106 Cipher Block Chaining (CBC) Method D-3
ANSI X9.23 . . . . . D-5
MAC Calculation Method D-7

| PKA92 Key Format and Encryption Process . . . D-8
| Encrypting a Key_Encrypting Key in the NL-EPP-5 Format . . . D-10
Triple-DES Ciphering Algorithms . . . D-11
| RSA Key-Pair Generation . . . D-15
Access Control Algorithms . . . D-16
Passphrase Verification Protocol . . . D-16
Design Criteria . . . D-16
Description of the Protocol . . . D-16
| Master Key Splitting Algorithm . . . D-18

| **Appendix E. Financial PIN Calculation Methods and PIN Blocks** . . . E-1
| PIN Calculation Methods . . . E-1
| IBM 3624 PIN Calculation Method . . . E-2
| IBM 3624 PIN Offset Calculation Method . . . E-3
| Netherlands PIN-1 Calculation Method . . . E-4
| IBM German Bank Pool Institution PIN Calculation Method . . . E-5
| VISA PIN Validation Value (PVV) Calculation Method . . . E-6
| Interbank PIN Calculation Method . . . E-7
| PIN Block Formats . . . E-8
| 3624 PIN Block Format . . . E-8
| ISO-0 PIN Block Format . . . E-9
| ISO-1 PIN Block Format . . . E-10
| ISO-2 PIN Block Format . . . E-11

Appendix F. Verb List . . . F-1

Appendix G. Access Control Request Function Codes . . . G-1

List of Abbreviations . . . X-1

Glossary . . . X-3

Index . . . X-15

Figures

| 0-1. New and Modified Verbs for Support of Master Key Loading and Coprocessor-Retained Keys . . . xvi
| 0-2. New Verbs for Support of Finance Industry PIN Processing . . . xvii
| 0-3. CCA RSA-Based Key Management Extended Verbs . . . xviii
| 0-4. Miscellaneous New and Extended Verbs . . . xviii
1-1. CCA Security API, Access Layer, Cryptographic Engine . . . 1-3
2-1. CCA Node, Access Control and Master Key Management Verbs . . . 2-1
| 2-2. Coprocessor-to-Coprocessor Master Key Transfer . . . 2-11
2-3. CSUAACI Rule_Array Input Keywords . . . 2-14
2-4. Contents of the verb_data_1 field . . . 2-15
2-5. Contents of the verb_data_2 field . . . 2-15
2-6. Contents of the Name Variable by Rule-array Keyword . . . 2-18
2-7. Contents of the Output_data Variable by Rule-array Keyword . . . 2-18
2-8. Cryptographic_Facility_Query Rule_Array Output Keywords . . . 2-28
2-9. CSUALCT Rule_Array Input Keywords . . . 2-39
2-10. Contents of the authentication parameters field . . . 2-39
2-11. Contents of the authentication data field . . . 2-40
3-1. Public-Key Key-Administration Services . . . 3-1
3-2. PKA96 Verbs with Key Token Flow . . . 3-2
3-3. PKA_Key_Token_Build Rule_Array Keywords . . . 3-13
3-4. PKA_Key_Token_Build Key Values Structures . . . 3-14
3-5. PKA_Key_Token_Change Rule_Array Keywords . . . 3-18
4-1. Hashing and Digital Signature Services . . . 4-1
4-2. Digital_Signature_Generate Rule_Array Keywords . . . 4-5
4-3. Digital_Signature_Verify Rule_Array Keywords . . . 4-8
4-4. One_Way_Hash Rule_Array Keywords . . . 4-11
5-1. Basic CCA DES Key Management Verbs . . . 5-1
5-2. Flow of Cryptographic Command Processing in a Cryptographic Facility . . . 5-5
5-3. Generic Key Types and Verb Usage . . . 5-6
| 5-4. Key_Token_Build Keyword Combinations . . . 5-7
5-5. Control Vector Key-Usage Keywords . . . 5-7
5-6. Key_Token Contents . . . 5-8
5-7. Key Identifier, Key Tokens, and Key Labels . . . 5-9
5-8. Key Processing Verbs . . . 5-12
5-9. Key Exporting and Importing . . . 5-14
5-10. Key_Type and Key_Form Keywords for One Key . . . 5-28
5-11. Key_Type and Key_Form Keywords for a Key Pair . . . 5-29
5-12. Key Lengths by Key Type . . . 5-30
5-13. Key_Part_Import Rule_Array Keywords . . . 5-34
5-14. Key_Token_Build Rule_Array Keywords . . . 5-39
5-15. Key_Token_Change Rule_Array Keywords . . . 5-42
5-16. Key_Token_Build Form Keywords . . . 5-45
6-1. Data Confidentiality and Data Integrity Verbs . . . 6-1
6-2. Decipher Rule_Array Keywords . . . 6-5
6-3. Encipher Rule_Array Keywords . . . 6-8
7-1. Key Storage Record Services . . . 7-1
7-2. Key_Token_BuildRule_Array Keywords . . . 7-5
7-3. Key_Token_BuildRule_Array Keywords . . . 7-13
7-4. Key_Token_BuildRule_Array Keywords . . . 7-20

© Copyright IBM Corp. 1997-98

8-1. Financial Services Support Verbs 8-1
| 8-2. Financial PIN Verbs 8-3
| 8-3. PIN Verb, PIN Calculation Method, and PIN-block Format Support Summary 8-5
| 8-4. Pad-Digit Specification by PIN-Block Format 8-10
| 8-5. PIN Extraction Method Keywords by PIN-Block Format 8-11
| 8-6. Clear_PIN_Generate_Alternate Rule_Array Keywords (First Element) 8-20
| 8-7. Clear_PIN_Generate_Alternate Rule_Array Keywords (Second Element) 8-21
| 8-8. Encrypted_PIN_Generate Rule_Array Keywords 8-26
| 8-9. Encrypted_PIN_Translate Rule_Array Keywords (First Element) . . 8-31
| 8-10. Encrypted_PIN_Translate Rule_Array Keywords (Second Element) 8-32
| 8-11. Encrypted_PIN_Translate Required Hardware Commands 8-33
| 8-12. Encrypted_PIN_Verify Rule_Array Keywords (First Element) 8-37
| 8-13. Encrypted_PIN_Verify Rule_Array Keywords (Second Element) . . 8-37
A-1. Return Code Values A-1
A-2. Reason Codes for Return Code 0 . . . . . A-2
A-3. Reason Codes for Return Code 4 . . . . . A-3
A-4. Reason Codes for Return Code 8 . . . . . A-5
A-5. Reason Codes for Return Code 12 . . . . . A-13
A-6. Reason Codes for Return Code 16 . . . . . A-14
A-7. Reason Codes for Return Code 24 . . . . . A-15
B-1. PKA Null Key Token Format B-2
B-2. Internal Key Token Format B-3
B-3. External Key Token Format B-4
B-4. Key Token Flag Byte 1 . . . . . B-4
B-5. Key Token Flag Byte 2 . . . . . B-4
B-6. RSA Token Header B-7
B-7. RSA Private Key, 1024-Bit Modular-Exponentiation Format B-7
B-8. Private Key, 2048-Bit Chineese-Remainder Format B-8
B-9. RSA Public Key B-9
B-10. RSA Private-key Name B-10
| B-11. RSA Public-key Certificate(s) Section Header B-10
| B-12. RSA Public-key Certificate(s) Public Key Subsection B-11
| B-13. RSA Public-key Certificate(s) Optional Information Subsection Header B-11
| B-14. RSA Public-key Certificate(s) User Data TLV B-11
| B-15. RSA Public-key Certificate(s) Environment Identifier (EID) TLV . . . B-11
| B-16. RSA Public-key Certificate(s) Serial Number TLV B-12
| B-17. RSA Public-key Certificate(s) Signature Subsection B-12
B-18. RSA Private-key Blinding Information B-13
B-19. Cipher, MAC_Generate, and MAC_Verify Chaining Vector Format . B-13
B-20. Key Storage File Header, Record 1 . . . . . B-14
B-21. Key Storage File Header, Record 2 . . . . . B-15
B-22. Key Record Format in Key Storage B-15
B-23. Key Record List Data Set Format B-16
B-24. Role layout B-18
B-25. Aggregate role structure with header B-19
B-26. Access control point structure B-20
B-27. Functions permitted in Default Role B-21
B-28. Profile layout B-21
B-29. Layout of profile Activation and Expiration dates B-21
B-30. Aggregate profile structure with header B-22
B-31. Layout of the Authentication Data field B-23

B-32. Authentication Data for each authentication mechanism B-24
| B-33. Passphrase authentication data structure B-25
| B-34. User profile data structure B-26
| B-35. Aggregate profile structure B-27
| B-36. Access Control Point List B-27
| B-37. Role data structure B-28
| B-38. Aggregate role data structure B-29
| B-39. Cloning Information Token Data Structure B-30
| B-40. Master Key Share TLV B-30
| B-41. Cloning Information Signature TLV B-30
| B-42. FCV Distribution Structure B-31
C-1. Control Vector Default Values for Generic Key Types C-3
C-2. Control Vector Base Bit Map C-4
| C-3. Multiply-Enciphering and Multiply-Deciphering CCA Keys C-9
C-4. Exchanging a Key with a Non-Control-Vector System C-12
D-1. Enciphering Using the CBC Method D-4
D-2. Deciphering Using the CBC Method D-4
D-3. Enciphering Using the ANSI X9.23 Method D-6
D-4. Deciphering Using the ANSI X9.23 Method D-6
D-5. MAC Calculation Method D-7
| D-6. PKA96 Clear DES Key Record D-8
| D-7. NL-EPP-5 Key Record Format D-10
D-8. EDE2 Algorithm D-11
D-9. DED2 Algorithm D-12
D-10. EDE3 Algorithm D-13
D-11. DED3 Algorithm D-14
D-12. Example of logon key computation D-16
| E-1. 3624 PIN Block Format E-8
| E-2. ISO-0 PIN Block Format E-9
| E-3. ISO-1 PIN Block Format E-10
| E-4. ISO-2 PIN Block Format E-11
F-1. Security API Verbs in Supported Environments F-1
G-1. Access control point codes G-1

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

3090
ACF/VTAM
AIX
AIX/6000
Application System/400
AS/400
CICS
Enterprise System/3090
Enterprise System/9000
Enterprise System/9370
ES/3090
ES/9000
ES/9370
IBM
IBM Registry
IBM World Registry
Micro Channel
MVS/DFP
MVS/ESA
MVS/SP
MVS/XA
Operating System/2
OS/2
Operating System/400
OS/400
Personal Security
Personal System/2
PS/2
PS/ValuePoint
POWERserver
POWERstation
RACF
RS/6000
SecureWay
System/360
System/370
System/390
S/390 G3 Enterprise Server
S/390 Multiprise
Systems Application Architecture
XGA

The following terms, denoted by a double asterisk (**) in this publication, are the trademarks of other companies:

Diebold	Diebold Incorporated
Docutel	Docutel
MASTERCARD	MasterCard International, Incorporated
Pentium	Intel Corporation
NCR	National Cash Register Corporation
RSA	RSA Data Security, Inc.
UNIX	UNIX Systems Laboratories, Incorporated
VISA	VISA International Service Association

About This Publication

The manual is intended for systems and applications analysts and application programmers who will evaluate or create programs for the IBM 4758 Common Cryptographic Architecture (CCA) support.

Prerequisite to using this manual is familiarity with the contents of the *IBM 4758 PCI Cryptographic Coprocessor General Information Manual*, IBM form number GC31-8608, that discusses topics important to the understanding of the information presented in this manual:

- The IBM 4758 PCI Cryptographic Coprocessor
- An overview of cryptography
- Supported cryptographic functions
- Function sets implemented by various IBM CCA products
- Organization of the relevant publications.

Revision History

Second Edition, CCA Support Program Release 1.3

This is the second edition of the *IBM 4758 CCA Basic Services Reference and Guide*, IBM form number SC31-8609-01.

This manual describes the *Common Cryptographic Architecture* (CCA) application programming interface (API) that is supported by the CCA Support Program feature Release 1.3 for the IBM 4758 PCI Cryptographic Coprocessor. *Depending on the Release level of the software that you have installed* certain capabilities described in this manual may not be available to your application program.

Changes and extensions to material previously published in the Basic Services manual are marked with the revision bar as shown at the left.

The CCA Support Program feature, Release 1.3, includes functional changes and enhancements in these catagories:

© Copyright IBM Corp. 1997-98

- Items related to:
  - Generation of a random master key
  - Distribution of master key shares in an “m of n” scheme
  - Optional retention of newly-generated RSA private keys within the Coprocessor
  - Registration of RSA public keys within the Coprocessor
  - Management of Coprocessor-retained keys.

Figure 0-1. New and Modified Verbs for Support of Master Key Loading and Coprocessor-Retained Keys

Verb	Page	Service Modification
Cryptographic_Facility_Control	2-22	Zeriozes (resets, reinitializes) the CCA node, sets the Coprocessor clock, and resets the intrusion latch. SET-EID and SET-MOFN keywords are added to the rule array to initialize the environment Id (EID) and the number of master key shares that can be distributed and that must be received to clone a master key.
Cryptographic_Facility_Query	2-26	Retrieves information about the coprocessor. Extended to return information about the m-of-n master key distribution shares values and a list of shares distributed and/or received.
Master_Key_Distribution	2-42	Supports the distribution and reception of master key shares. New verb.
Master_Key_Process	2-46	Enables the introduction of a master key into the coprocessor. Extended with the **RANDOM** keyword for generation of a random master key, and with the **CLR-OLD** keyword to empty the Old Master Key Register.
PKA_Key_Generate	3-6	Generate an RSA key-pair. Added the RETAIN keyword and signature support.
PKA_Public_Key_Hash_Register	3-22	Register the hash of a public key used later to verify an offered public key, see PKA_Public_Key_Register. New verb.
PKA_Public_Key_Register	3-24	Register a public key used later to verify an offered public key. Registration requires that a hash of the public key has previously been registered within the Coprocessor, see PKA_Public_Key_Hash_Register. New verb.
Retained_Key_Delete	7-21	Delete a key retained within the cryptographic engine. New verb.
Retained_Key_List	7-22	List keys retained within the cryptographic engine. New verb.

Note: The key token structure extensions are described “RSA Public-key Certificate Section” on page B-10.

- Items related to the processing of Finance Industry PINs (personal identification numbers). Six new verbs are described in

Figure 0-2. New Verbs for Support of Finance Industry PIN Processing

Verb	Page	Service Modification
Clear_PIN_Encrypt	8-12	This verb formats a PIN into a PIN-block and outputs the PIN-block as an encrypted quantity. This verb is extended with keyword **RANDOM** to generate random PINs that are output in encrypted PIN blocks.
Clear_PIN_Generate	8-15	This verb generates a clear PIN, or a PIN offset.
Clear_PIN_Generate_Alternate	8-18	This verb extracts a customer-selected PIN or institution-assigned PIN from an encrypted PIN-block and generates a PIN offset.
Encrypted_PIN_Generate	8-24	This verb generates a PIN and formats the PIN into an encrypted PIN block.
Encrypted_PIN_Translate	8-29	This verb operates in two modes. Translate mode re-encrypts a PIN block under a different key. Reformat mode does one or more of the following: • Reformats a PIN from one PIN block format into another PIN block format • Changes selected non-PIN digits in a PIN block • Re-encrypts a PIN block.
Encrypted_PIN_Verify	8-34	This verb extracts and verifies a PIN by using the specified PIN calculation method. An offset value can be included in the verification of the value in the input PIN block.
Key_Generate	5-25	This verb is used to generate DES keys. The documentation is updated to reflect the additional PIN key types that can be generated and to note the requirement for use of the Extended Key Generate command with selected key-type combinations.
Note: The PIN block formats and the PIN generation processes are described in Appendix E, "Financial PIN Calculation Methods and PIN Blocks" on page E-1.		

- Items related to additional capabilities for the distribution of DES keys using public key techniques.

Figure 0-3. CCA RSA-Based Key Management Extended Verbs		
Verb	**Page**	**Service Modification**
PKA_Symmetric_Key_Export	5-47	Exports a symmetric key under an RSA public key.
PKA_Symmetric_Key_Generate	5-49	For "asymmetric" DES keys (achieved through the use of control vectors), generate a local master-key-encrypted key, and its asymmetric counterpart, encrypted under an RSA public key.
PKA_Symmetric_Key_Import	5-52	Imports a symmetric key under an RSA private key.

- Miscellaneous items
  - A new verb, Diversified_Key_Generate, and a new key-generating key-type, to generate a "diversified key" in support of operations with finance industry smart cards
  - Extension of the Key_Generate verb to support generation of key-generating keys and double-length MAC and MACVER keys.
  - Extension of the Key_Test verb to perform an "encrypt zeros" key test pattern generation and verification function
  - Extension of the master key process to permit zeroization of the old master key register
  - Extension of the MAC_Generate and MAC_Verify verbs to perform ANSI X9.19 double-length DES key MAC generation and verification.

Figure 0-4. Miscellaneous New and Extended Verbs		
Verb	**Page**	**Service Modification**
Diversified_Key_Generate	5-20	Generates a "diversified" key by encrypting an input value with a supplied key-generating key.
Key_Generate	5-25	Generates a random DES key or DES key pair, enciphers the keys, and updates or creates internal and external key tokens. The verb is extended to support generation of the key-generating key key-type and the double-length MAC/MACVER key types.
Key_Test	5-35	Generates or verifies a verification pattern for keys and key parts. A verb extension permits the use of the "encrypt zeros" key test method with the use of keyword **ENC-ZERO**.
Master_Key_Process	2-46	Manages the contents of the master key registers. A verb extension permits the contents of the Old Master Key Register to be zeroized with the use of keyword **CLR-OLD**.
MAC_Generate	6-10	Generates a message authentication code (MAC). An extension provides for support of the ANSI X9.19 double-length MAC-key process.
MAC_Verify	6-13	Verifies a message authentication code (MAC). An extension provides for support of the ANSI X9.19 double-length MAC-key process.

- Modifications of the original implementation of the SET services are documented in Chapter 8, “Financial Services Support Verbs.” These changes were made in release 1.1 and are formally published in this manual revision.

Organization

This manual includes:

- Chapter 1, “Introduction to Programming for the IBM CCA” presents an introduction to programming for the CCA application programming interface and products.
- Chapter 2, “CCA Node Management and Access Control” provides a basic explanation of the access control system implemented within the hardware. The chapter also explains the master key concept and administration, and introduces CCA DES key management.
- Chapter 3, “RSA Key Administration” explains how to generate and distribute RSA keys between CCA nodes and with other RSA implementations.
- Chapter 4, “Hashing and Digital Signatures” explains how to protect and confirm the integrity of data using data hashing and digital signatures.
- Chapter 5, “Basic CCA DES Key Management” explains basic DES key management services available with CCA.
- Chapter 6, “Data Confidentiality and Data Integrity” explains how to encipher data using DES and how to verify the integrity of data using the DES-based Message Authentication Code (MAC) process. The ciphering and MACing services are described.
- Chapter 7, “Key Storage Verbs” explains how to use key labels and how to employ key storage managed by the accesses software.
- Chapter 8, “Financial Services Support Verbs” explains services for the cryptographic portions of the Secure Electronic Transaction (SET) protocol and PIN processing services.

These appendices are included:

- Appendix A, “Return Codes and Reason Codes” describes the return codes issued by the TSS products.
- Appendix B, “Data Structures” describes the various data structures for key token, chaining vector records, key storage records, and the key record list data set.
- Appendix C, “CCA Control Vector Definitions and Key Encryption” describes the control vector bits and provides rules for the construction of a control vector.
- Appendix D, “Algorithms and Processes” describes, in further detail, the algorithms and processes mentioned in this book.
- Appendix E, “Financial PIN Calculation Methods and PIN Blocks” describes processes and formats implemented by the PIN processing support.

Related Publications

In addition to the manuals listed below, you may wish to refer to other CCA product publications which may be of use with applications and systems you might develop for use with the IBM 4758 product. While there is substantial commonality in the API supported by the CCA products, and while this manual seeks to guide you to a common subset supported by all CCA products, other individual product publications may provide further insight into potential issues of compatibility.

All of the IBM 4758 related publications can be obtained from the Library page that you can reach from the IBM 4758 home page at http://www.ibm.com/security/cryptocards.

IBM 4758 PCI Cryptographic Coprocessor General Information Manual, GC31-8608
The General Information manual is prerequisite reading for this manual.

IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Guide, SC31-8610
Describes the installation of the CCA Support Program and the operation of the Cryptographic Node Management utility.

IBM 4758 PCI Cryptographic Coprocessor Installation Manual, SC31-8623.

Chapter 1. Introduction to Programming for the IBM CCA

This chapter will introduce you to the IBM *Common Cryptographic Architecture* (CCA) application programming interface (API). This chapter explains some basic concepts you use to obtain cryptographic and other services from the IBM 4758 PCI Cryptographic Coprocessor and its CCA Support Program feature. Before continuing to read this manual, please review the "About This Publication" on page xv and first become familiar with prerequisite information as described in that section.

In this chapter you can read about:

- What CCA services are available with the IBM 4758
- An overview of the CCA environment
- The Security API, programming fundamentals
- How the verbs are organized in the remainder of the book.

What CCA Services Are Available with the IBM 4758

CCA products provide a variety of cryptographic processes and data security techniques. Your application program can call *verbs* (services) to perform these types of functions:

- Encrypt and decrypt information, generally using the DES algorithm in the cipher block chaining mode to enable *data confidentiality*
- Hash data to obtain a *digest*, or process the data to obtain a message authentication code that is useful in demonstrating *data integrity*
- Form and validate *digital signatures* to demonstrate both data integrity and *non-repudiation*
- Generate, encrypt, transform, and verify finance industry PINs with a comprehensive set of *PIN-processing* services
- Manage the various keys necessary to perform the above operations. CCA is especially strong and versatile in this area; inadequate key-management techniques are a major source of weakness in many cryptographic implementations.
- Administrative services for controlling the initialization and operation of the CCA node.

This book describes the many available services in the following chapters. The services are grouped by topic and within a chapter are listed in alphabetical order by name. Each chapter opens with an introduction to the services found in that chapter.

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

© Copyright IBM Corp. 1997-98

An Overview of the CCA Environment

Figure 1-1 on page 1-3 provides a conceptual framework for positioning the CCA *Security API*. Application programs make procedure calls to the API to obtain cryptographic and related I/O services. The CCA 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 will receive a synchronous response. That is, your application program will lose control until the access layer returns a response at the conclusion of processing your request.

The products that implement the CCA API consist of both hardware and software components. The software consists of application development support and runtime software components.

- The application development support 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 utility programs and application programs
  - An "agent" function that is logically part of the calling application program or utility
  - An environment-dependent request routing function
  - The *server* environment that gives access to the cryptographic engine.

  Generally, the cryptographic engine is implemented in a hardware device that includes a general purpose processor and often also includes specialized cryptographic electronics. These components are encapsulated in a protective environment to enhance security.

The utility programs include support for administering the hardware access controls, administering DES and public-key cryptographic keys, and configuring the software support. See the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program*, SC31-8610., for a description of the utility programs provided with the Cryptographic Adapter Services licensed software.

You can create application programs that use the products via the CCA API, or you can purchase applications from IBM or other sources. This book is the primary source of information for designing systems and application programs that use the CCA API with the IBM 4758 Coprocessor.

IBM 4758 PCI Cryptographic Coprocessor: The coprocessor provides a secure programming and hardware environment wherein DES and RSA processes are performed. The CCA support program enables applications to employ a set of DES- and RSA-based cryptographic services utilizing the IBM 4758 hardware. Such services include:

- RSA key-pair generation.
- Digital signature generation and verification.
- Cryptographic key wrapping and unwrapping, including the SET-standardized "OAEP" key-wrapping process.
- Data encryption and MAC generation/verification.

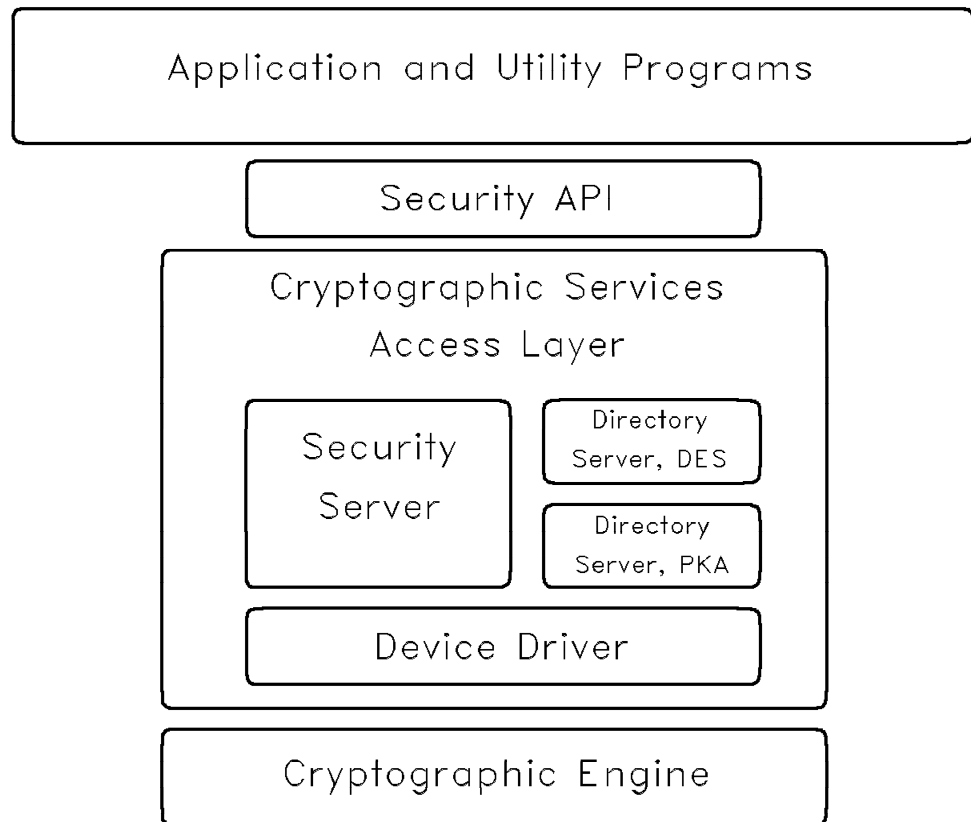

Figure 1-1. CCA Security API, Access Layer, Cryptographic Engine

- PIN processing for the financial services industry
- Other services, including DES key management based on CCA's control-vector enforced key separation.

CCA: IBM has created the IBM Common Cryptographic Architecture (CCA) as the basis for a consistent cryptographic product family. Implementations of this architecture were first released in 1989, and it has been extended throughout the years. The IBM 4758 and its CCA support program feature are a recent CCA product offering that today implements a portion of those functions available with older products as well as many new services such as the support of the SET protocol.

Applications employ the CCA security API to obtain services from and to manage the operation of a cryptographic system that meets CCA architecture specifications.

Cryptographic Engine: The CCA architecture defines a cryptographic subsystem that contains a *cryptographic engine* operating within a protected boundary; see Figure 1-1. The coprocessor's tamper-resistant, tamper-responding environment provides physical security for this boundary, and the CCA architecture provides the concomitant logical security needed for the full protection of critical information.

Access Control: Each CCA node has an access control system enforced by the hardware and protected software. This access control system permits you to determine whether programs and persons can use the cryptographic and data storage services. Although your computing environment may be considered open,

the specialized processing environment provided by the cryptographic engine can be kept secure; selected services are provided only when logon requirements are met. The access control decisions are performed within the secured environment of the cryptographic engine and can not 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 key securely held at the manufacturing facility, and then the signed *device key* is stored within the coprocessor. The manufacturing facility key has itself been signed by a securely-held key unique to the IBM 4758 product line.

The private key within the coprocessor—known as the *device private key*—is retained in the coprocessor. From this time on, the coprocessor sets all security-relevant keys and data items to zero if tampering is detected ***or if the coprocessor batteries are removed***. This *zeroization* will result in the loss of the factory-certified device key, the device private key, and all other data stored in battery-protected memory. Certain critical data stored in the Coprocessor flash memory is encrypted. The key used to encrypt such data is itself retained in the battery protected memory that is zerioized upon a tamper detection event.

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 of the protected environment. These working keys are wrapped (triple-enciphered) by a *master key*. The master key is held in the clear (not enciphered) within the the cryptographic engine.

The number of keys a node can use is restricted only by the storage capabilities of the node, not by the finite amount of storage within the coprocessor secure module. In addition, keys can be used by other cryptographic nodes that have the same master key data. This feature is useful in high-availability or high-throughput environments where multiple cryptographic processors must function in parallel.

Establishing a 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 will have the same master key[1] unless master key cloning is authorized and in use. If the coprocessor detects tampering and destroys the master key, there is no way to recover the working keys that it wrapped.

Another method enables authorized users to enter 168-bit *key parts* into the cryptographic engine. As each part is entered, that part is exclusive-ORd with the contents of the new master key register. When all three 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. "Understanding and Managing Master Keys" on page 2-8 provides additional detail about master key management.

[1] Unless, out of the 2^{168} possible values, another node randomly generates the same master key data.

A master key can be "cloned" (copied) from one IBM 4758 CCA node to another IBM 4758 CCA node through a process of master-key-shares distribution. Under this process that is protected through the use of digital certificates and authorizations, the master key can be reconstituted in one or more additional IBM 4758s through the transport of encrypted shares of the master key.

CCA Verbs: Application and utility programs (*requesters*) obtain service from the CCA support program by issuing service requests ("verb calls" or "procedure calls") to the runtime subsystem. 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 of 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: In order to ensure that only designated individuals (or programs) can execute sensitive commands such as master key loading, each command is assigned a *control point* value within the cryptographic engine access control system.

The access control system includes *roles*; each role defines the permissible activities for users associated with that role. The access control system also has a set of user *profile* entries that associate a user ID (UID) with its assigned role, its logon identification information, and a session key. Within a host *process*, one and only one user can be logged on at a time. The *default* role defines the operations permitted in the absence of a logged-on user. The coprocessor supports multiple logons from different host processes. The coprocessor also supports requests from multiple threads within a single host process.

After a user successfully logs on, subsequent requests made to the cryptographic system are permitted or denied based on the permissions defined by the role associated with the user profile. Verb requests and responses are MACd using the session key. "Access Control Algorithms" provides details about the logon method, and "CCA Access Control" on page 2-2 provides a further explanation of the access control system.

The Logon_Control verb call establishes a session key. This key is held in user application memory space and is used to compute a DES MAC on the information in a verb call. The cryptographic engine validates the MAC as one of its checks to ensure the authenticity of the verb request. Verb responses are protected in a similar manner.

The user can issue another instance of Logon_Control in order to logoff; logoff causes the cryptographic engine to destroy its copy of the session key and to mark the user profile as not active.

How Application Programs Obtain Service

Application programs and utility programs (*requestors*) obtain services from the products by issuing service requests (*verb* calls) to the runtime subsystem of software and hardware. These requests are in the form of procedure calls that must be programmed according to the rules of the language in which the application is coded. The services that are available are collectively described as

the *security API*. All of the software and hardware accessed through the security API should be considered an integrated subsystem.

The cryptographic services access layer can receive requests concurrently from multiple application programs, will serialize the requests, and return a response to each requestor. There are other multi-processing 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 in which application programs and utilities are linked to the API services depends on the computing environment. In the OS/2, AIX, and NT environments, the operating systems dynamically link application security API requests to the subsystem DLL code (AIX: shared library). Details for linking to the API are covered in the guide book for the individual software products, *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program* book, SC31-8610.

Together, the security API stub code or DLL and the *environment-dependent request routing mechanism* act as an *agent* on behalf of the application and present a request to the server. In the OS/2, AIX and NT environments, the requests can be issued by one or more programs. Each request is processed by the server as a self-contained unit of work from a first-in, first-out queue. The programming interface can be called concurrently by applications running as different processes. The API is also thread safe; that is, use of the API by more than one thread within a process is permitted and the application programmer need not take any special precautions to serialize use of the cryptographic API. Both 16-bit and 32-bit entry point service is provided. You control the choice of entry point through your use of the import library portion of the cryptographic adapter services software; see the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program*, SC31-8610. (The cryptographic adapter services software is implemented as 32-bit support.)

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 through the use of a *key label*. Before deciding whether to use the key storage facility or to let the application retain the keys, you must consider system design trade-off factors, such as key backup, the impact of master key changing, the lifetime of a key, and so forth.

The Security API, Programming Fundamentals

The security application programming interface (API) is the interface for accessing the services provided by the SecureWay cryptographic products in OS/2, AIX, and NT workstations.

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 will be used with other CCA products, you should compare the other product literature for differences.

Your application program requests a service through the security API by using a procedure call for a *verb*.[2] 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; see Appendix F, "Verb List" for a list of supported verbs and where information about the verb is published.

The security API is designed for use with high-level languages, such as C, COBOL, PL/I, or Pascal, 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 that 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 (service) is described in the reference material and provides an explanation of the characteristics of the security API.

Each callable service, or 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 (general language name)
- Entry-point name (computer language name)
- Supported environments
- Description
- Restrictions
- Format
- Selected parameters
- Hardware command requirements.

Entry-Point Name: Each verb has an entry-point name that is used in your program to call the verb. Each verb's entry point name begins with one of the following:

CSNB generally the DES services

CSND RSA public key services (PKA96)

[2] The term *verb* implies an action that an application program can initiate; other systems and publications might use the term *callable service* instead of *verb*.

CSUA Cryptographic-node and hardware control services.

The last three letters in the entry point name identify the specific service in a group and are often the first letters of the principal words in the verb pseudonym.

You use the entry point name in the call statement in your application program to call the verb.

Format Section: The format section in each verb description lists the entry-point name on the first line in bold type. This is followed by the list of parameters for the verb. Generally the direction in which the variable identified by the parameter is passed is listed along with the type of variable (integer or string), and the size, number, or other special information about the variable.

The format section for each verb lists the parameters after the entry-point name in the sequence in which they must be coded.

Parameters: All information that is exchanged between your application program and a verb is through the variables that are 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 entry-point name and the parameters for each verb are shown in the following format:

Parameter name	Direction	Data Type	Length of Data
entry_point_name			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
Parameter_5	Direction	Data Type	Length
Parameter_6	Direction	Data Type	Length
⋮			
Parameter_n	Direction	Data Type	Length

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 1-10. The remaining parameters (*parameter_5*, *parameter_6*, ... *parameter_n*), are unique for each verb. For descriptions of these parameters, see the definitions with the individual verbs.

Variable 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.

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

Variable Type: A variable that is identified by a verb parameter can be a single value or a one-dimensional array. If a parameter identifies an array, each data element of the array is of the same data type. If the number of elements in the

array is variable, a preceding parameter identifies a variable that contains the actual number of elements in the associated array. Unless otherwise stated, a variable is a single value, not an array.

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

Integer A 4-byte (32-bit), signed, twos-complement binary number.

In the AIX environment, integer values are presented in 4 bytes in the sequence high-order to low-order (*big endian*). In the personal computer (Intel) environments, integer values are presented in 4 bytes in the sequence low-order to high-order (*little endian*).

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 type-casting 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 "Rule_Array and Other Keyword Parameters" on page 1-11 below.

Variable Length: This is the length, in bytes, of the variable identified by the parameter being described. This length may be expressed as a specific number of bytes or it may be expressed in terms of the contents of another variable.

For example, the lengths of the exit_data variable is expressed in this manner. The length of the exit_data string variable is specified in the exit_data_length variable. This length is shown in the parameter tables as "*exit_data_length* bytes," The rule_array variable, on the other hand, is an array whose elements are eight-byte strings. The number of elements in the rule array is specified in the rule_array_count variable and its length is shown as "rule_array_count * 8 bytes."

Note: Variable lengths (integer, for example) that are implied by the variable data type are not shown in these tables.

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
- Rule_array and other keyword parameters
- Key_identifiers, key_labels, and key_tokens.

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 data storage.

Entry_point_name			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes

Return_Code

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" for more information about return codes

Reason_Code

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" for more information about reason codes

Exit_Data_Length

The *exit_data_length* parameter is a pointer to an integer value containing the length of the string (in bytes) that is returned by the *exit_data* parameter. *The exit_data_length parameter should be set to zero to ensure compatibility with any future extension or other operating environment*.

Exit_Data

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 exit.

Note: The IBM 4758 CCA Support Program does not support user exits. The *exit_data_length* and *exit_data* must be declared in the parameter list. *The exit_data_length parameter should be set to zero to ensure compatibility with any future extension or other operating environment*.

Return Code and Reason Code Overview: The *return code* provides a general indication of the results of verb processing and is the value that your application program should use in determining the course of further processing. The *reason code* provides more specific information about the outcome of verb processing. Note that 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.

The return codes have these general meanings:

Value	Meaning
0	Normal completion; a few nonzero reason codes are associated with this return code.
4	The verb processing completed, but without full success. For example, this return code can signal that a supplied PIN was found to be invalid.
8	Indicates that the verb stopped processing. Generally the application programmer will need to investigate the problem and will need to know the associated reason code.
12	Indicates that the verb stopped processing. The reason is most likely related to a problem in the setup of the hardware or in the configuration of the software.
16	Indicates that the verb stopped processing. A processing error occurred in the product. If these errors persist, a repair of the hardware or a correction to the product software may be required.

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.

Rule_Array and Other Keyword Parameters

Rule_array parameters and some other parameters use keywords to transfer information. Generally, a rule array consists of data elements that contain keywords that direct specific details of the verb process. Almost all keywords, in a rule array or otherwise, are 8 bytes in length, and should be uppercase, left-justified, and padded with space characters. While some implementations can fold lower-case characters to upper case, you should always code the keywords in upper case.

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.

Key_Identifiers, Key_Labels, and Key_Tokens

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; see “Key Tokens” on page B-1 for a detailed description of the key token structures.

Keys are described as one of:

Internal A key that is encrypted for local use. The cryptographic engine will decrypt (unwrap) an internal key to use the key in a local operation. Once a key is entered into the system it is always encrypted (wrapped) if it appears outside of 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 that is complete and ready for use. During entry of a key, the internal key token can contain a flag that indicates the key information is incomplete.

External A key that is either in the clear, or is encrypted (wrapped) by some *key-encrypting key*. Generally, when a key is to be transported from place to place, or is to be held for a significant

period of time, it is required to encrypt the key with a *transport key*. A key wrapped by a transport key-encrypting key is designated External.

RSA public keys are not encrypted values (in PKA96), and when not accompanied by private key information, are retained in an external key token.

Internal key tokens can be stored in a flat file that is maintained by the *directory server*. These key tokens are referenced by use of a *key label*. A key label is an alphanumeric string that 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 that will be used to locate a *key record* in key storage

Key identifier The variable can contain either a key token or a key label. The first byte in the variable defines if 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 will be processed as a key label. There are additional restrictions on the value of a key label, see "Key Label Content" on page 7-2. The first byte in all key token structures is in the range of X'01' to X'1F'. X'FF' as the first byte of a key-related variable passed to the API raises an error condition.

How the Verbs Are Organized in the Remainder of the Book

Now that you have a basic understanding of the API, you can find these topics in the remainder of the book:

- Chapter 2, "CCA Node Management and Access Control" explains how the cryptographic engine and the rest of the cryptographic node is administered. There are two topics:
  - Master key administration
  - Access control administration.

  Keeping cryptographic keys private or secret can be accomplished by retaining them in secure hardware. Keeping the keys in secure hardware can be inconvenient or impossible if there are a large number of keys, or the key has to be usable with more than one hardware device. In the CCA implementation, a *master key* is used to encrypt (wrap) locally-used keys. The master key itself is securely installed within the cryptographic engine and can not be retrieved from the engine.

  As you examine the verb descriptions throughout this book, you will see reference to "**Required Commands**." Almost all of the verbs request the cryptographic engine (the "adapter" or "Coprocessor") to perform one or more commands in the performance of the verb. Each of these commands have to be authorized for use. Access control administration concerns managing those authorizations.

- Chapter 3, “RSA Key Administration” explains how you can generate and protect an RSA key-pair. The chapter also explains how you can control the distribution of the RSA private key for backup and archive purposes and to enable multiple cryptographic engines to use the key for performance or availability considerations. Related services for creating and parsing RSA key tokens are also described.

  When you wish to backup an RSA private key, or supply the key to another node, you will use a double-length DES key-encrypting key, a *transport key*. You will find it useful to have a general understanding of the DES key management concepts found in chapter Chapter 5, “Basic CCA DES Key Management.”

- Chapter 4, “Hashing and Digital Signatures” explains how you can
  - Provide for demonstrations of the integrity of data --demonstrate that data has not been changed
  - Attribute data uniquely to the holder of a private key.

  These problems can be solved through the use of a digital signature. The chapter explains how you can hash data (obtain a number that is characteristic of the data, a *digest*) and how you can use this to obtain and validate a digital signature.

- Chapter 5, “Basic CCA DES Key Management” explains the many services that are available to manage the generation, installation, and distribution of DES keys.

  An important aspect of DES key management is the means by which these keys can be restricted to selected purposes. Deficiencies in key management are the main means by which a cryptographic system can be broken. Controlling the use of a key and its distribution is almost as important as keeping the key a secret. CCA employs a non-secret quantity, the *control vector* to determine the use of a key and thus improve the security of a node. Control vectors are described in detail in Appendix C, “CCA Control Vector Definitions and Key Encryption.”

- Chapter 6, “Data Confidentiality and Data Integrity” explains how you can encrypt data. The chapter also describes how you can use DES to demonstrate the integrity of data through the production and verification of *message authentication codes*.

- Chapter 7, “Key Storage Verbs” explains how you can label, store, retrieve, and locate keys in the cryptographic-services access-layer managed *key storage*.

- Chapter 8, “Financial Services Support Verbs” explains how you can cryptographically process keys and information related to the *Secure Electronic Transaction (SET)* protocol. A suite of verbs for processing personal identification numbers (PIN) in various phases of automated teller machine transaction processing and PIN generation are described.

Chapter 2. CCA Node Management and Access Control

This chapter discusses:

- The access control system that you can use to control who can perform various sensitive operations at what times
- Controlling the cryptographic facility
- The CCA Master Key, what it is and how you manage the key
- How you can initialize the cryptographic key storage that is managed by the support software.

The verbs that you use to accomplish these tasks are listed in Figure 2-1.

Figure 2-1. CCA Node, Access Control and Master Key Management Verbs

Verb	Page	Service	Entry Point	Svc Lcn
Access_Control_Initialization	2-13	Initialize or update access control tables in the Coprocessor.	CSUAACI	E
Access_Control_Maintenance	2-16	Query or control installed roles and user profiles.	CSUAACM	E
Cryptographic_Facility_Control	2-22	Reinitializes the CCA application, sets the adapter clock, resets the intrusion latch, sets the CCA environment identifier (EID), sets the number of master key shares required and possible for distributing the master key, loads the CCA function control vector (FCV) that manages international export and import regulation limitations.	CSUACFC	E
Cryptographic_Facility_Query	2-26	Retrieves information about the coprocessor and the state of master key shares distribution processing.	CSUACFQ	E
Key_Storage_Initialization	2-36	Initializes one or the other of the key storage files that can store DES or RSA (public/private) keys.	CSNBKSI	S/E
Logon_Control	2-38	Logs on or off the cryptographic adapter.	CSUALCT	E
Master_Key_Distribution	2-42	Supports the distribution and reception of master key shares. New verb.	CSUAMKD	E
Master_Key_Process	2-46	Enables the introduction of a master key into the coprocessor, the random generation of a master key, the setting and clearing of the master key registers.	CSNBMKP	E
Svc Lcn: Service location: E: Engine, S: Software				

© Copyright IBM Corp. 1997-98

CCA Access Control

This section describes these CCA Access Control system topics:

- Understanding access control
- Role-based access control
- Initializing and managing the access control system
- Logging on and logging off
- Protecting your transaction information.

Understanding Access Control

Access Control is the process that determines which services of the 4758 Cryptographic Coprocessor are available to a user at any given time. The system administrator can give users differing authority, so that some users have the ability to use CCA services that are not available to others. In addition, a given user's authority may be limited by parameters such as the time of day, or the day of the week.

Role-based Access Control

The IBM 4758 Cryptographic Coprocessor uses *Role-based* access control. In a role-based system, the administrator defines a set of *roles*, which correspond to the classes of coprocessor users. Each user will be enrolled by defining a *user profile*, which will map the user to one of the available roles. Profiles are described in "Understanding Profiles" on page 2-3.

Note: For the purposes of this discussion, a user is defined as either a human user or an automated, computerized process.

As an example, a simple system might have the following three roles.

General User The user class which includes all coprocessor users who do not have any special privileges.

Key Management Officer Those people who have the authority to change cryptographic keys for the coprocessor.

Access Control Administrator Those people who have the authority to enroll new users into the coprocessor environment, and modify the access rights of those users who are already enrolled.

There would be only a few people who hold the role of Key Management Officer or Access Control Administrator, but there would be a large population of people with the role of General User.

A role-based system is more efficient than one in which the authority is assigned individually for each user. In general, the users can be separated into just a few different categories of access rights. The use of roles allows the administrator to define each of these categories just once, in the form of a role.

Understanding Roles

Each Role defines the permissions and other characteristics associated with users having that Role. The role contains the following information.

Role ID A character string which defines the name of the role. This name is referenced in user profiles, to show which role defines the user's authority.

Permitted Operations A list defining which restricted operations the user will be allowed to perform in the coprocessor. Each command corresponds to one of the primitive functions that make up the access control system.

Required User Authentication Strength Level The access control system is designed to allow a variety of user authentication mechanisms. Although the only one supported today is passphrase authentication, the design is ready for others that may be used in the future.

All user authentication mechanisms are given a strength rating, an integer value where zero is the minimum strength, corresponding to no authentication at all. If the strength of the user's authentication mechanism is less than the required strength for the role, the user is not permitted to log on.

Valid Time and Valid Days-of-Week These values define the times of the day, and the days of the week when the users with this role will be permitted to log on. If the current time is outside the values defined for the role, logon will not be allowed. It is possible to choose values that will let users log on at any time on any day of the week.

Note: Times must be specified in Greenwich Mean Time (GMT).

In addition, the role contains control and error checking fields. The detailed layout of the role data structure can be found in “Role Structure” on page B-18.

The Default Role: Every coprocessor must have at least one role, called the *default role*. Any user who has not logged on and been authenticated will operate with the capabilities and restrictions defined in the default role.

Note: Since unauthenticated users have authentication strength equal to zero, the Required User Authentication Strength Level of the Default Role must also be zero.

The coprocessor can have a variable number of additional roles, as needed by the customer. For simple applications, the default role by itself may be sufficient. Any number of roles can be defined, as long as the coprocessor has enough available storage to hold them.

Understanding Profiles

Any user who needs to be authenticated to the coprocessor must have a *user profile*. Users who only need the capabilities defined in the default role do not need a profile.

A profile defines a specific user to the card. Each profile contains the following information:

User ID This is the “name” used to identify the user to the coprocessor. The User ID is an eight byte value, with no restrictions on its content. Although it will typically be an unterminated ASCII character string, any 64-bit string is acceptable.[1]

Role ID This character string identifies the role that contains the user's authorization information. The authority defined in the role takes effect after the user successfully logs on to the coprocessor.

Activation and Expiration Dates These values define the first and last date on which this user is permitted to log on to the coprocessor. An administrator whose role has the necessary authority can reset these fields to extend the user's access period.

All four digits of the year are stored, so that there will be no problem at the turn of the century.

Logon failure count This field contains a count of the number of consecutive times the user has failed a logon attempt, due to incorrect authentication data. The user will no longer be allowed to log on after three consecutive failures. This lockout condition can be reset by an administrator whose role has sufficient authority.

Authentication Data The authentication data is the information used to verify the identity of the user. It is a self-defining structure, which can accommodate many different authentication mechanisms. In the current coprocessor, user identification is accomplished by means of a passphrase entered by the user at the client workstation.

The profile's authentication data field can hold data for more than one authentication mechanism. If more than one is present in a user's profile, any of the mechanisms can be used to log on. Different mechanisms, however, may have different strengths.

The structure of the authentication data is described in “The Authentication Data Structure” on page B-22.

In addition to these fields, the profile contains a *header* which contains the following information.

Profile structure version This is a two-byte structure which defines the version of the profile data structure that follows. For release 1.x of the CCA Support Program, the structure version is 10. This is specified with a version field containing 1 in the first byte, and 0 in the second byte.

Profile length This two-byte structure contains the number of bytes contained in the remainder of the profile, following the header.

When the user enrolls, the profile is stored in non-volatile memory inside the secure module on the coprocessor. When the user logs on, this stored profile is used to authenticate the information presented to the coprocessor. In most applications, the majority of the users will operate under the default role, and will

[1] In many cases, a utility program will be used to enter the user ID. That utility may restrict the ID to ASCII characters.

not have user profiles; only the security officers and other special users will need profiles.

In addition, the profile contains other control and error checking fields. The detailed layout of the profile data structure can be found in "Profile Structure" on page B-21.

Initializing and Managing the Access Control System

Before you can use a coprocessor with a newly loaded CCA Support Program, it must be initialized with roles, profiles, and other data. You will also need to update some of these values from time to time. Access control initialization and management are the processes you will use to accomplish this.

You can initialize and manage the access control system in either of two ways.

- You can use the Cryptographic Node Management utility program
- You can write programs that use the access control verbs described in this chapter.

The verbs allow you to write programs that do more than the utility program included with the CCA Support Program. If your needs are simple, however, the utility program may do everything you need. The Cryptographic Node Management utility is described in the *IBM 4758 Cryptographic Coprocessor; CCA Support Program*, SC31-8610.

The Access Control Management and Initialization Verbs

Two verbs provide all of the access control management and initialization functions.

CSUAACI Perform access control initialization functions.

CSUAACM Perform access control management functions.

With Access_Control_Initialization, you can perform functions such as:

- Loading roles and user profiles
- Changing the expiration date for a user profile
- Changing the authentication data in a user profile
- Resetting the authentication failure count in a user profile.

With Access_Control_Maintenance, you can perform functions such as:

- Getting a list of the installed roles or user profiles
- Retrieving the non-secret data for a selected role or user profile
- Deleting a selected role or user profile from the coprocessor.
- Get a list of the users who are logged on to the coprocessor.

These two verbs are fully described on pages 2-13 and 2-16 respectively.

Permitting Changes to the Configuration

It is possible to initialize the coprocessor so no one is authorized to perform *any* functions, including further initialization. It is also possible to program the coprocessor where operational commands are available, but not initialization commands; meaning you could never change the configuration of the coprocessor. This happens if you initialize the coprocessor with no roles having the authority to perform initialization functions.

Take care to ensure that you define roles that have the authority to perform initialization, including the **RQ-TOKEN** and **RQ-REINT** options of the Cryptographic_Facility_Control (CSUACFC) verb.

You must also ensure there are active profiles that use these roles.

If you accidentally configure your coprocessor so that initialization is not allowed, you can recover by reloading the coprocessor firmware. This will delete all information previously loaded, and restore the coprocessor to its new state.

Configuration and Greenwich Mean Time (GMT)

The coprocessor always operates with GMT time. This means that the time, date, and day-of-the-week values in the coprocessor are measured according to GMT. This can be confusing because of its effect on access control checking.

Each user has operating time limits, based on values in their role and profile. These include:

- Profile activation and expiration dates
- Time-of-day limits
- Day-of-the-week limits.

All of these limits are measured using time in the *coprocessor's* frame of reference, not the user's. If your role says that you are authorized to use the coprocessor on days Monday through Friday, it means Monday through Friday *in the GMT time zone*, not your local time zone. In like manner, if your profile expiration date is December 31, it means December 31 in GMT.

In the Eastern United States, your time differs from GMT by four hours during the part of the year daylight savings time is in effect. At noon local time, it is 4:00 PM GMT. At 8:00 PM local time, it is midnight GMT which is the time the coprocessor increments its date and day-of-the-week to the next day.

For example, at 7:00 PM on Tuesday, December 30 local time, it is 11:00 PM, Tuesday, December 30 to the coprocessor. Two hours later, however, at 9:00 PM, Tuesday, December 30 local time, it is 1:00 AM *Wednesday, December 31* to the coprocessor. If your role only allows you to use the coprocessor on Tuesday, you would have access until 8:00 PM on Tuesday; after that, it would be Wednesday in the GMT time frame used by the coprocessor.

This happens because the coprocessor does not know where you are located, and how much your time differs from GMT. Time zone information could be obtained from your local workstation, but this information could not be trusted by the coprocessor; it could be forged in order to obtain access at times the system administrator intended to keep you from using the coprocessor.

Note: During the portions of the year when Daylight savings time is not in effect, the time difference between Eastern Standard Time and GMT is 5 hours.

Logging On and Logging Off

A user must log on to the coprocessor in order to activate a user profile. This is the only way to use a role other than the default role. You log on and log off using the Logon_Control verb, which is described in detail on 2-38.

When you successfully log on, you establish a *session* with the coprocessor. As part of that session, you establish a randomly derived *session key* which is subsequently used to protect information you interchange with the coprocessor. This protection is described in detail in the next section. The logon process and its algorithms are described in "Passphrase Verification Protocol" on page D-16.

In order to log on, you must prove your identity to the coprocessor. This is accomplished using a *passphrase*, a string of up to 64 characters which are known only to you and the coprocessor. A good passphrase should not be too short, and it should contain a mixture of alphabetic characters, numeric characters, and special symbols such as "*," "+," "!," and others. It should not be comprised of familiar words or other information which someone might be able to guess.

When you log on, no part of your passphrase ever travels over any interface to the coprocessor. The passphrase is hashed and processed into a key that encrypts information passed to the Coprocessor. The Coprocessor has a copy of the hash and can construct the same key to recover and validate the log-on information. CCA does not communicate your passphrase outside of the memory owned by the calling process.

When you have finished your work with the coprocessor, you must log off in order to end your session. This invalidates the session key you established when you logged on, and frees resources you were using in the host system and in the coprocessor.

Protecting Your Transaction Information

When you are logged on to the coprocessor, the information transmitted to and from the CCA coprocessor application is cryptographically protected using your session key. A message authentication code is used to ensure that the data was not altered during transmission. Since this code is calculated using your session key, it also verifies that you are the originator of the request, not someone else attempting to impersonate you.

For some verbs, it is also important to keep the information *secret*. This is especially important with the Access_Control_Initialization verb, which is used to send new role and profile data to the coprocessor. To ensure secrecy, some verbs offer a special *protected* option, which causes the data to be encrypted using your session key. This prevents disclosure of the critical data, even if the message is intercepted during transmission to the coprocessor.

Understanding and Managing Master Keys

In a CCA node, the master key is used to encrypt (“wrap”) working keys used by the node that can appear outside of the cryptographic engine. The working keys are triple-encrypted. This method of securing keys enables a node to operate on an essentially unlimited number of working keys without concern for storage space within the confines of the cryptographic engine.

The CCA design supports three master key registers, *new*, *current*, and *old*. While a master key is being assembled, it is accumulated in the new master key register. Then the Master_Key_Process verb is used to transfer (*set*) the contents of the new master key register to the current master key register.

Working keys are normally encrypted by the current master key. To facilitate continuous operations, CCA implementations also have an old master key register. When a new master key is transferred to the current master key register, the pre-existing (if any) contents of the current master key register are transferred to the old master key register. With the IBM 4758 CCA implementation, whenever a working key must be decrypted by the master key, *master key verification pattern* information that is included in the key token is used to determine if the current or the old master key must be used to recover the working key. Special status is returned in case of use of the old master key so that your application programs can arrange to have the working key updated to encryption by the current master key (using the Key_Token_Change verb). Whenever a working key is encrypted for local use, it is encrypted using the current master key.

Master keys are established in one of three ways:

Introduction of master key parts. The parts are exclusive-ORed within the cryptographic engine. Knowledge of a single part gives no information about the final key when multiple (random-valued) parts are exclusive-ORed.

A common technique is to record the values of the parts (typically on paper or diskette) and independently store these values in locked safes. When the master key is to be instantiated in a cryptographic engine, individuals who are trusted to not share the key-part information retrieve the parts and enter the information into the cryptographic engine. The Master_Key_Process verb supports this operation.

Entering the first and subsequent parts is authorized by two different control points so that a cryptographic engine (the Coprocessor) can enforce that two different roles, and thus profiles, are activated to install the master key parts. Of course this requires that roles exist that enforce this separation of responsibility.

Setting of the master key is also a unique command with its own control point. Therefore you can setup the access control system to require the participation of at least three individuals.

You can check that appropriate information has been entered through the use of the Key_Test verb. You can check the contents of any of the master key registers, and the key parts as they are entered.

Random generation of a new master key. The Master_Key_Process verb can be used to randomly generate a new master key within the cryptographic engine. The value of the new master key is not available outside of the cryptographic engine.

This method, which is a separately authorized command invoked through use of the Master_Key_Process verb, ensures that no one has access to the value of the master key. Random generation of a master key is useful when the shares technique described next is used, and when keys shared with other nodes are distributed using public key techniques or when DES *transport keys* are established between nodes. In these cases, there is no need to re-establish a master key with the same value.

'Cloning' a master key from one cryptographic engine to another cryptographic engine. In certain high-security applications, it is desirable to copy a master key from one cryptographic engine to another without exposure of the value of the master key. The IBM 4758 CCA implementation supports copying the master key through a process of splitting the master key into **n** *shares*. **m** shares ($1 \leq m \leq n \leq 15$) are required to reconstitute the master key in another engine.

The term "cloning" is used to differentiate the process from "copying" because no one share, or any combination of fewer than m shares, provide sufficient information needed to reconstitute the master key.

You establish the 'm' and 'n' values through the use of the Cryptographic_Facility_Control verb.

The shares of the current master key are prepared using one mode of the Master_Key_Distribution verb. Prepared shares are also obtained from the cryptographic engine using this mode of the verb. The other mode of the Master_Key_Distribution verb is used to enter an individual share into the target cryptographic engine. When sufficient shares have been entered, the verb returns status that the cloned master key is now complete within the new master key register of the target engine.

The master key shares are signed by the source engine. Each signed share is then triple-encrypted by a fresh triple-length DES key, the *share-encrypting key*. To obtain a share of the master key, you present a certified public key from the target cryptographic engine. After validating the certificate, the share-encrypting key is wrapped (encrypted) using the supplied public key.

At the target cryptographic engine, an encrypted share and its wrapped share-encrypting key are presented to the engine. The private key to unwrap the share-encrypting key must exist within the cryptographic engine as a "retained key" (a private key that never leaves the engine). This private key must also have been marked as suitable for operation with the Master_Key_Distribution verb when it was generated.

When receiving a share, you must also supply the *share-signing key* in a certificate to the Master_Key_Distribution verb. The engine validates the certificate, and uses the validated public key to validate the individual master key share.

The certificates used to validate the share-signing public key and the target-engine public key used to wrap the share-encrypting key are validated by the cryptographic engine using a *retained public key*. A retained public key is introduced into a cryptographic engine in a two-part process using the PKA_Public_Key_Hash_Register and PKA_Public_Key_Register verbs. This

allows you to establish two distinct roles. Two different individuals are authorized so that split-authority and dual control can be enforced in setting up the certificate validating public key.

You identify the nodes with unique 16-byte identifiers of your choice. The *environment Id* (EID) is also established through the use of the Cryptographic_Facility_Control verb.

The processing of a given share (share 1, 2, ..., n) requires authorization to a distinct control point so that you can enforce split responsibility in obtaining and installing the shares.

This secure master-key cloning process is supported by the Cryptographic Node Management (CNM) utility. See Chapter 5 of the *CCA Support Program* manual. That utility can hold the certificates and shares in a "data base" that you can transport on diskette between the various nodes:

- The public key certifying node
- The master-key-share source node
- The master-key-share target node.

The certifying node can be either the share source or target node as you desire, or can be an independent node that might be located in a cryptographic control center.

Although not currently supported by IBM products, the shares could be stored on intermediate devices (e.g., smart cards), provided that the devices could perform the required key management and digital signature functions.

With the current capabilities of the IBM 4758 CCA Support Program, you must have initialized the target Coprocessor with its retained private key, and have had the associated public key certified, before you obtain shares for the target Coprocessor. This implies that the target Coprocessor has been initialized and is not reset before a master key is cloned to the Coprocessor. Once the master key has been cloned, the target Coprocessor can be removed from an active machine.

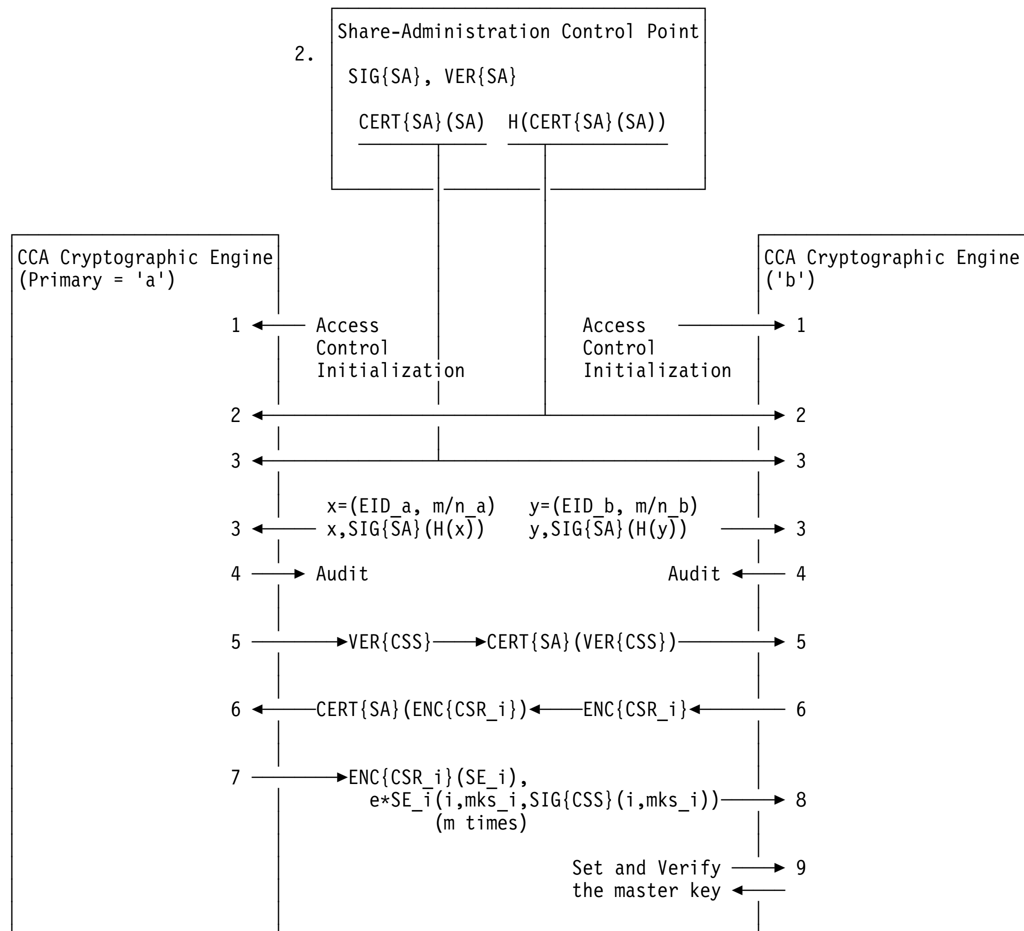

Figure 2-2. Coprocessor-to-Coprocessor Master Key Transfer

Figure 2-2 depicts the steps of a master key cloning scenario. These steps include:

1. Install and audit appropriate access control roles and profiles. Have operators change their profile passwords. Ensure that the roles provide the degree of responsibility-separation that you require.
2. Generate a retained RSA private key (the *Share-Administration* (SA) key) to certify the public keys used in the scheme and certify the associated public key. Distribute the hash of this certificate to the source(s) and target(s) nodes.
3. Distribute the certified public key (the SA key) that is validated by the already distributed hash of the certificate. Also install the environment Ids (EIDs) and m-of-n values.
4. Audit the node arrangements.
5. Generate the *Coprocessor Share Signing* (CSS) key and have this key certified by the SA key.
6. Generate the *Coprocessor Share Receiving* (CSR) key and have this key certified by the SA key.
7. Obtain shares of the master key. Note that generally fewer shares are required to reconstitute the master key than that which can be obtained from

the source node. Thus corruption of some of the information that is in transit between source and target can be tolerated.

8. Deliver master key shares.
9. Set and verify the master key.

Note that the keys in master key registers can be tested through the use of the Key_Test verb.

Access_Control_Initialization(CSUAACI)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The Access_Control_Initialization verb is used to initialize or update parameters and tables for the Access Control system in the 4758 Cryptographic Coprocessor.

You can use this verb to perform the following services:

- Load roles and user profiles
- Change the expiration date for a user profile
- | Change the authentication data, such as a passphrase, in a user profile
- Reset the authentication failure count in a user profile.

You select which service to perform by specifying the corresponding keyword in the input rule array. You can only perform one of these services per verb call.

Restrictions

None.

Format

CSUAACI			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Inp/Outp	Integer	
rule_array	Inp/Outp	String array	rule_array_count * 8 bytes
	~~*name*~~	~~Input~~	~~String~~	~~8 bytes~~
verb_data_1_length	Input	Integer	
verb_data_1	Input	String	verb_data_1_length bytes
verb_data_2_length	Input	Integer	
verb_data_2	Input	String	verb_data_2_length bytes

| **Note:** In the first printing of this manual, a *name* parameter was incorrectly
| included in the parameter list.

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable.

On input, this contains the number of elements you provide in the input rule array. On output, the verb will set this to the number of rule array elements it returns to the application program.

rule_array

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

On input, the keywords in the rule array specify the operation being performed. The rule array keywords are shown below:

Figure 2-3. CSUAACI Rule_Array Input Keywords

Keyword	Meaning
Keywords used to select which function to perform	
INIT-AC	Initializes roles and user profiles.
CHGEXPDT	Changes the expiration date in a user profile.
CHG-AD	Changes authentication data in a user profile or a user's passphrase. **Note:** The **PROTECTD** keyword must also be used whenever you use **CHG-AD**. You must authenticate yourself before you are allowed to change authentication data, and the use of protected mode verifies that you have been authenticated.
RESET-FC	Resets the count of consecutive failed logon attempts for a user. Clearing the count permits a user to log on again, after being locked out due to too many failed attempts.
Keywords used to select options	
PROTECTD	Specifies that the verb should operate in *protected* mode. Data sent to the card is protected by encrypting the data with the user's session key, K_S. If the user has not successfully logged on, there will be no session key in effect, and the **PROTECTD** keyword will result in an error.
REPLACE	Specifies that a new role or profile can replace an existing role or profile with the same name. This keyword applies only when the rule array contains the **INIT-AC** keyword. Without the **REPLACE** keyword, any attempt to load a role or profile which already exists will be rejected. This protects against accidentally overlaying a user's profile with one for a different user, who has chosen the same profile ID as one that is already on the card. Although less likely, it also protects against the same kind of problem with duplicate role IDs.

verb_data_1_length

The verb_data_1_length parameter is a pointer to an integer variable containing the length (in bytes) of the verb_data_1 variable.

verb_data_1

The verb_data_1 parameter is a pointer to a string variable containing data used by the verb.

This field is used differently depending on the function being performed. Figure 2-4 shows the content for each of the rule array keywords that selects a different function.

Figure 2-4. Contents of the verb_data_1 field	
Keyword	**Contents of** *verb_data_1* **field**
INIT_AC	The field contains a list of zero or more user profiles to be loaded into the coprocessor.
CHGEXPDT, CHG-AD, or RESET-FC	The field contains the eight-character profile ID for the user profile that is to be modified.

verb_data_length_2
The verb_data_length_1 parameter is a pointer to an integer variable containing the length (in bytes) of the data in the verb_data_2 field.

verb_data_2
The verb_data_2 parameter is a pointer to a string variable containing data used by the verb.

This field is used differently depending on the function being performed. Figure 2-5 shows the content for each of the rule array keywords that selects a different function.

Figure 2-5. Contents of the verb_data_2 field	
Keyword	**Contents of** *verb_data_2* **field**
INIT_AC	The field contains a list of zero or more roles to be loaded into the coprocessor.
CHGEXPDT	The field contains the new expiration date to be stored in the specified user profile. The expiration date is an eight character string, in the form **YYYYMMDD**.
CHG-AD	The field contains the new authentication data, to be used in the specified user profile. Authentication data structures are described in "Access Control Data Structures" on page B-18. If the profile currently contains authentication data for the same authentication mechanism, that data is replaced by the new data. If the profile does not contain authentication data for the mechanism, the new data is *added* to the data currently stored for the specified profile.
RESET-FC	The verb_data_2 field is empty. Its length is zero.

Required Commands

The Access_Control_Initialization verb requires the following commands to be enabled:

- Initialize the access control system roles and profiles (offset X'0112') with the **INIT-AC keyword**
- Change the expiration date in a user profile (offset X'0113') with the **CHGEXPDT** keyword
- Change the authentication data in a user profile (offset X'0114') with the **CHG-AD** keyword
- Reset the logon failure count in a user profile (offset X'0115') with the **RESET-FC** keyword.

Access_Control_Maintenance (CSUAACM)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The Access_Control_Maintenance verb is used to query or control installed roles and user profiles.

You can use this verb to perform the following services:

- Get a list of the installed roles or user profiles
- Retrieve the non-secret data for a selected role or user profile
- Delete a selected role or user profile from the cryptographic coprocessor
- Get a list of the users who are logged on to the coprocessor.

You select which service to perform by specifying the corresponding keyword in the input rule array. You can only perform one of these services per verb call.

Restrictions

None.

Format

CSUAACM

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
name	Input	String	8 bytes
output_data_length	Output	Integer	
output_data	Output	String	output_data_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable.

On input, this contains the number of elements you provide in the input rule array. ~~On output, the verb will set this to the number of rule array elements it returns to the application program.~~

rule_array

The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

On input, the keywords in the rule array specify the operation being performed. The rule array keywords are shown below:

On input, you put keywords into the rule array to specify what The rule array keywords are shown below:

Keyword	Meaning
Keywords used to select which function to perform	
LSTPROFS	Retrieves a list of the user profiles currently installed in the coprocessor. Keyword **Q-NUM-RP** shows how to determine how much data this request will return to the application program.
LSTROLES	Retrieves a list of the roles currently installed in the coprocessor Keyword **Q-NUM-RP** shows how to determine how much data this request will return to the application program.
GET-PROF	Reads the non-secret part of a specified user profile.
GET-ROLE	Reads the non-secret part of a role definition from the coprocessor
DEL-PROF	Deletes a specified user profile.
DEL-ROLE	Deletes a specified role definition from the coprocessor
Q-NUM-RP	Queries the number of roles and profiles presently installed in the coprocessor. This allows the application program to know how much data will be returned with the **LSTROLES** or **LSTPROFS** keywords.
Q-NUM-UR	Queries the number of users who are currently logged on to the coprocessor. It can be used to predict the amount of data that will be returned with the **LSTUSERS** keyword. Users may log on or log off between the time you use **Q-NUM-UR** and the time you use **LSTUSERS**, so the list of users may not always contain exactly the number the coprocessor reported were logged on.
LSTUSERS	Retrieves a list of the profile IDs for all users who are currently logged on to the coprocessor.

name

The *name* parameter is a pointer to a string variable containing the eight byte name of a role or user profile which is the target of the request.

This field is used differently depending on the function being performed. Figure 2-6 shows the content of this field for each of the possible rule array keywords.

Figure 2-6. Contents of the Name Variable by Rule-array Keyword	
Keyword	**Contents of *name* variable**
LSTPROFS, LSTROLES, Q-NUM-RP, Q-NUM-UR, or LSTUSERS	The *name* field is unused.
GET-PROF or DEL-PROF	The *name* field contains the eight-character profile ID for the user profile that is to be retrieved or deleted.
GET-ROLE or DEL-ROLE	The *name* field contains the eight-character role ID for the role that is to be retrieved or deleted.

output_data_length

The *output_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *output_data* field. This buffer must be a multiple of four bytes.

On input, this parameter must be set to the total size of the buffer pointed to by the *output_data* parameter. On output, it will contain the number of bytes of data returned by the verb in the *output_data* field.

output_data

The *output_data* parameter is a pointer to a string variable containing data returned by the verb. Any integer value returned in the *output_data* field is in *big-endian* format; the high order byte of the value is in the lowest numbered address in memory.

This field is used differently depending on the function being performed. Figure 2-7 shows the content for each of the rule array keywords.

Figure 2-7 (Page 1 of 3). Contents of the Output_data Variable by Rule-array Keyword	
Keyword	**Contents of *verb_data_2* Variable**
LSTPROFS	The *output_data* field contains a list of the profile IDs for all the user profiles stored in the coprocessor.
LSTROLES	The *output_data* field contains a list of the role IDs for all the roles stored in the coprocessor.

Figure 2-7 (Page 2 of 3). Contents of the Output_data Variable by Rule-array Keyword

Keyword	Contents of *verb_data_2* Variable
GET-PROF	The *output_data* field contains the non-secret portion of the selected user profile. This includes the following data, in the order listed. **Profile version** Two bytes containing two one-byte integer values, where the first byte contains the major version number and the second byte contains the minor version number. **Comment** A 20-character field containing a comment which describes the profile. **Role** The eight character name of the user's assigned Role. **Logon failure count** A one-byte integer containing he number of consecutive failed logon attempts by the user. **Pad** A one-byte padding value, which will contain X'00'. **Activation date** The first date on which the profile is valid. The date consists of a two-byte integer containing the year, followed respectively by a one-byte integer for the month and a one-byte integer for the day of the month. **Expiration date** The last date on which the profile is valid The format is the same as the *Activation date* described above. **List of enrolled authentication mechanism information** For each authentication mechanism associated with the profile, the verb returns a series of three integer values: 1. The two-byte *Mechanism ID*. 2. The two-byte *Mechanism Strength*. 3. The four-byte authentication data *Expiration Date*, which has the same form as the *Activation date* described above. Note that the authentication data itself is not returned; only the IDs, strength, and expiration date of the data are returned.

Figure 2-7 (Page 3 of 3). Contents of the Output_data Variable by Rule-array Keyword

Keyword	Contents of *verb_data_2* Variable
GET-ROLE	The field contains the non-secret portion of the selected role. This includes the following data, in the order listed. **Role version** Two bytes containing integer values, where the first byte contains the major version number and the second byte contains the minor version number. **Comment** A 20-character field containing a comment which describes the role. This field is non-terminated. **Required authentication strength level** A two-byte integer defining how secure the user authentication must be in order to authorize this role. **Lower time limit** The earliest time of day that this role can be used. The time limit consists of two integer values, a one-byte hour, followed by a one-byte minute. The hour can range from 0-23, and the minute can range from 0-59. **Upper time limit** The latest time of day that this role can be used. The format is the same as the *Lower time limit*. **Valid days of the week** A one-byte field defining which days of the week this role can be used. Seven bits of the byte are used to represent Sunday through Saturday, where a '1' bit means that the day is allowed, while a '0' bit means it is not. The first bit (MSB) is for Sunday, and the last bit (LSB) is unused and will be set to zero. **Access control point list** The access control point bit map, defining which functions a user with this role is permitted to execute.
DEL-PROF or DEL-ROLE	The *output_data* field is empty. Its length is zero.
Q-NUM-RP	The *output_data* field contains an array of two four-byte integers. The first integer is the number of roles currently loaded with use of the Access_Control_Initialization verb while the second integer is the number of user profiles currently loaded with use of the same verb.
Q-NUM-UR	The *output_data* field contains a single integer value, which indicates the number of users currently logged on to the coprocessor.
LSTUSERS	The *output_data* field contains an array of eight-character profile IDs, one for each user currently logged on to the coprocessor. The list is not in any meaningful order.

Required Commands

The Access_Control_Maintenance verb requires the following commands be enabled in the hardware:

- Read public access control information (offset X'0116') with the **LSTPROFS**, **LSTROLES**, **GET-PROF**, **GET-ROLE**, , and **Q-NUM-RP** keywords
- Delete a user profile (offset X'0117') with the **DEL-PROF** keyword
- Delete a role (offset X'0118') with the **DEL-ROLE** keyword,

Cryptographic_Facility_Control (CSUACFC)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

Use the Cryptographic_Facility_Control verb to perform the following services:

- Reinitialize the CCA application in the coprocessor
- Set the time and date in the coprocessor clock
- Reset the coprocessor Intrusion Latch.
- Load or clear the Function Control Vector, which defines limitations on the cryptographic functions available in the coprocessor
- Establish the environment identification (EID), which is a user-setable identifier
- Establish the minimum and maximum number of “cloning information” shares that are required and that can be used to pass sensitive information from one Coprocessor to another Coprocessor.

Select which service to perform by specifying the corresponding keyword in the input rule array. You can only perform one of these services per verb call. space character.

Restrictions

Use only these characters in an environment identifier (EID): a...z, A...Z, 0...9, @, &, #, and the space character.

Format

CSUACFC			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
verb_data_length	Inp/Outp	Integer	
verb_data	Inp/Outp	String	verb_data_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array.

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

This verb requires two keywords in the rule array. One specifies the coprocessor to which the request is intended, the other specifies the function to perform. No rule array elements are set by the verb; the rule array is empty on output. The rule_array keywords are shown below:

Keyword	Meaning
Specifying which adapter to use	
ADAPTER1	Specifies which adapter the request will go to. ADAPTER1 is the only value supported.[00]
Specifying what control function to perform	
RQ-TOKEN	Requests a random eight-byte token from the adapter, which is returned in the verb_data parameter. This is the first step when reinitializing the coprocessor. The second step for reinitialization uses **RQ-REINT**, described below.
RQ-REINT	For **RQ-REINT**, you must set the verb_data field to the one's complement of the token that was returned by the card when you executed the verb using the **RQ-TOKEN** keyword. This is the second and final step when reinitializing the coprocessor. This two-step process provides protection against accidental reinitialization of the card.
SETCLOCK	Sets the date and time on the coprocessor. You must put the date and time values in the verb_data parameter, as described under the description of that parameter.
RESET-IL	Clears the Intrusion Latch on the coprocessor.
LOAD-FCV	Loads a new Function Control Vector into the coprocessor.
CLR-FCV	Deletes the Function Control Vector from the coprocessor.
SET-EID	Establishes an *environment identification*, or EID value.
SET-MOFN	Establish the minimum and maximum number of “cloning information” shares that are required and that can be used to pass sensitive information from one Coprocessor to another Coprocessor.

verb_data_length

The verb_data_length parameter is a pointer to an integer containing the number of bytes of data in the *verb_data* field.

verb_data

The *verb_data* parameter is a pointer to a string variable containing data used by the verb on input, or generated by the verb on output.

This field is used differently depending on the value of the function selection rule array keyword.

- For **RQ-TOKEN**, *verb_data* is an output parameter. It receives an eight-byte randomly generated value, which the application uses with the **RQ-REINT** keyword on a subsequent call.

On input, *verb_data_length* must contain the length of the buffer addressed by the verb_data pointer. This buffer must be at least eight bytes in length.

- For **RQ-REINT**, *verb_data* is an input parameter. You must set it to the one's complement of the token you received as a result of the **RQ-TOKEN** call.
- For **SETCLOCK**, *verb_data* is an input parameter. It must contain a character string which contains the current GMT time and date. This string has the form **YYYYMMDDHHmmSSWW**, where these fields are defined as follows.

  YYYY The current year.

  MM The current month, from 01 to 12.

  DD The current day of the month, from 01 to 31.

  HH The current hour of the day, from 00 to 23.

  mm The current minutes past the hour, from 00 to 59.

  SS The current seconds past the minute, from 00 to 59.

  WW The current day of the week, where Sunday is represented as 01, and Saturday by 07.

- For **LOAD-FCV** ...
- For **CLR-FCV** ...
- For **SET-EID**, verb data is an input variable. The variable contains a 16-byte *environment identification*, or EID value. This identifier is used in verbs such as PKA_Key_Generate, PKA_Symmetric_Key_Export and PKA_Symmetric_Key_Import. Use only these characters in an environment identifier: a...z, A...Z, 0...9, @, &, #, and the space character.
- For **SET-MOFN**, verb data is an input variable. The variable contents establish the minimum and maximum number of “cloning information” shares that are required and that can be used to pass sensitive information from one Coprocessor to another Coprocessor. Verb_data contains a two element array of integers. The first element is the **m** required number of shares to reconstruct cloned information (see the Master_Key_Distribution verb). The second element is the **n** maximum number of shares that can be issued to reconstruct cloned information (see the Master_Key_Distribution verb).

Required Commands

The Cryptographic_Facility_Control verb requires the following commands be enabled in the hardware:

- Reinitialize Device (offset X'0111') with the **RQ-TOKEN**, **RQ-REINT** keywords
- Set Clock (offset X'0110') with the **SETCLOCK** keyword
- Reset Intrusion Latch (offset X'010F') with the **RESET-IL** keyword.
- Load a Function Control Vector (offset X'0119') with the **LOAD-FCV** keyword.

- Clear the Function Control Vector (offset X'011A') with the **CLR-FCV** keyword.
- Set EID command (offset X'011C') with the **SET-EID** keyword.
- Initialize Master Key Cloning command (offset X'011D') with the **SET-MOFN** keyword.

Cryptographic_Facility_Query (CSUACFQ)

Platform/ Product	OS/2	AIX	NT
IBM-4758	**X**	**X**	**X**

The Cryptographic_Facility_Query verb is used to retrieve information about the Cryptographic Coprocessor and the CCA application program in that coprocessor. This information includes the following:

- General information about the coprocessor
- General information about the CCA application program in the coprocessor
- Status of master key shares distribution
- Environment identifier, EID
- Diagnostic information from the coprocessor
- Export control information from the coprocessor
- Time and date information.

Restrictions

None.

Format

CSUACFQ			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
verb_data_length	Inp/Outp	Integer	
verb_data	Inp/Outp	String	verb_data_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable.

On input, the verb will examine input rule array elements until either the count of elements is exceeded, or until an element of eight space characters is encountered.

On output, the verb will set this variable to the number of rule array elements it returns to the application program, a number that will be less than or equal to the number input.

rule_array

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

On input, you set the rule array to specify the type of information you want to retrieve. There are two input rule array elements, as described below.

Keyword	Meaning
Specifying which adapter to use	
ADAPTER1	Specifies the cryptographic coprocessor for which the request is intended. ADAPTER1 is the only value supported.
Specifying what information to return	
STATCCA	Gets CCA-related status information.
STATCARD	Gets coprocessor-related basic status information.
STATDIAG	Gets diagnostic information.
STATEXPT	Gets function control vector-related status information.
STATMOFN	Gets master key shares distribution information.
STATEID	Gets the environment identifier, EID.
TIMEDATE	Reads the current time, date, and day of the week from the secure clock within the coprocessor.

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, STATCARD, STATDIAG, STATEXPT, STATMOFN**, or **STATEID**.

For rule array elements that contain numbers, those numbers are represented by numeric characters which are left-justified and padded on the right with space characters. For example, a rule array element which contains the number 2 will contain the character string **“2   ”**.

On output, the rule elements can have the values shown in the table below.

[2] If multiple adapters are supported in the workstation, they will be referenced using ADAPTER1, ADAPTER2, and so on.

Figure 2-8 (Page 1 of 7). Cryptographic_Facility_Query Rule_Array Output Keywords		
Element Number	**Name**	**Description**
Output rule array for option STATCCA		
1	NMK Status	State of the New Master Key register. • 1 means the register is clear • 2 means the register contains a partially complete key • 3 means the register contains a complete key.
2	CMK Status	State of the Current Master Key register. • 1 means the register is clear • 2 means the register contains a key.
3	OMK Status	State of the Old Master Key register. • 1 means the register is clear • 2 means the register contains a key.
4	CCA application version	An eight character string that identifies the version of the CCA application that is running in the coprocessor.
5	CCA application build date	An eight character string containing the build date for the CCA application that is running in the coprocessor.
6	User Role	An eight character string containing the Role identifier which defines the host application user's current authority.

Figure 2-8 (Page 2 of 7). Cryptographic_Facility_Query Rule_Array Output Keywords

Element Number	Name	Description
Output rule array for option STATCARD		
1	Number of adapters installed	The number of active cryptographic coprocessors installed in the machine. This will always be 1 in the current implementation.
2	DES hardware level	A numeric character string containing an integer value identifying the version of DES hardware that is on the coprocessor.
3	RSA hardware level	A numeric character string containing an integer value identifying the version of RSA hardware that is on the coprocessor.
4	POST version	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.
5	Card Operating System name	A character string identifying the operating system firmware on the coprocessor.
6	Card Operating System version	A character string identifying the version of the coprocessor's operating system firmware.
7	Card part number	A character string containing the eight-character part number identifying the version of the coprocessor.
8	Card EC level	A Character string containing the eight-character EC (Engineering Change) level for this version of the coprocessor.
9	Miniboot version	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.
10	CPU speed	a character string containing the operating speed of the microprocessor chip, in Megahertz.
11	Adapter ID	A unique identifier programmed into the coprocessor. The coprocessor's Adapter ID is an eight-byte binary value.

Figure 2-8 (Page 3 of 7). Cryptographic_Facility_Query Rule_Array Output Keywords

Element Number	Name	Description
12	Flash memory size	A character string containing the size of the Flash EPROM memory on the coprocessor, in kilobytes.
13	DRAM memory size	A character string containing the size of the dynamic RAM (DRAM) memory on the coprocessor, in kilobytes.
14	Battery-backed memory size	A character string containing the size of the battery-backed RAM on the coprocessor, in kilobytes.
15	Serial Number	The unique serial number of the coprocessor. The serial number is factory-installed.
Output rule array for option STATDIAG		
1	Battery state	A numeric character string containing a value which indicates whether the battery on the coprocessor needs to be replaced. • 1 means that the battery is good • 2 means that the battery should be replaced.
2	Intrusion Latch state	A numeric character string containing a value which indicates whether the Intrusion Latch on the coprocessor is set or cleared. • 1 means that the latch is cleared • 2 means that the latch is set.
3	Error log status	A numeric character string containing a value which indicates whether there is data in the coprocessor CCA error log. • 1 means that the error log is empty • 2 means that the error log contains data, but is not yet full • 3 means that the error log is full, and cannot hold any more error data.
4	Mesh intrusion	A numeric character string containing a value to indicate whether the coprocessor has detected tampering with the protective mesh that surrounds the secure module. This indicates a probable attempt to physically penetrate the module. • 1 means no intrusion had been detected • 2 means an intrusion attempt detected

Figure 2-8 (Page 4 of 7). Cryptographic_Facility_Query Rule_Array Output Keywords

Element Number	Name	Description
5	Low voltage detected	A numeric character string containing a value to indicate whether a power supply voltage was below the minimum acceptable level. This may indicate an attempt to attack the security module. • 1 means only acceptable voltages have been detected • 2 means a voltage has been detected below the low voltage tamper threshold
6	High voltage detected	A numeric character string containing a value to indicate whether a power supply voltage was above the maximum acceptable level. This may indicate an attempt to attack the security module. • 1 means only acceptable voltages have been detected • 2 means a voltage has been detected above 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 the acceptable limits. This may indicate an attempt to obtain information from the module. • 1 means the temperature is acceptable • 2 means the temperature has been detected outside of acceptable limits
8	X-ray radiation detected	A numeric character string containing a value to indicate whether X-ray radiation was detected inside the secure module. This may indicate an attempt to obtain information from the module. • 1 means no X-ray radiation has been detected • 2 means X-rays radiation has been detected

Figure 2-8 (Page 5 of 7). Cryptographic_Facility_Query Rule_Array Output Keywords

Element Number	Name	Description
9, 11, 13, 15, 17	Last five commands executed	These five rule array elements contain the last five commands that were executed by the coprocessor CCA application. They are in chronological order, with the most recent command in element 9. Each element contains the SAPI command code in the first four characters, and the subcommand code in the last four characters.
10, 12, 14, 16, 18	Last five return codes	These five rule array elements contain the SAPI 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.

Figure 2-8 (Page 6 of 7). Cryptographic_Facility_Query Rule_Array Output Keywords

Element Number	Name	Description
Output rule array for option STATEXPT		
1	Base CCA services availability	A numeric character string containing a value to indicate whether base CCA services are available. • 0 means basic CCA services are not available • 1 means base CCA services are available,
2	CDMF availability	A numeric character string containing a value to indicate whether CDMF encryption is available. • 0 means CDMF encryption is not available • 1 means CDMF encryption is available,
3	56-bit DES availability	A numeric character string containing a value to indicate whether 56-bit DES encryption is available. • 0 means 56-bit DES encryption is not available • 1 means 56-bit DES encryption is available,
4	Triple-DES availability	A numeric character string containing a value to indicate whether Triple-DES encryption is available. • 0 means Triple-DES encryption is not available • 1 means Triple-DES encryption is available,
5	SET services availability	A numeric character string containing a value to indicate whether SET (Secure Electronic Transactions) services are available. • 0 means SET services are not available • 1 means SET services are available,
6	Maximum modulus for symmetric key encryption	A numeric character string containing the maximum modulus size that is 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.

Figure 2-8 (Page 7 of 7). Cryptographic_Facility_Query Rule_Array Output Keywords		
Element Number	**Name**	**Description**
Output rule array for option STATMOFN Elements 1 and 2, and elements 3 and 4, are each treated as a 16-byte string with the high-order 15 bytes having meaningful information and the sixteenth byte containing a space character. Each byte provides status information about the 'i'th share, 1≤i≤15, of master key information.		
1, 2	Master key shares generation	The 15 individual bytes are set to one of these character values: **0** Can not be generated **1** Can be generated **2** Has been generated but not distributed **3** Generated and distributed once **4** Generated and distributed more than once.
3, 4	Master key shares reception	The 15 individual bytes are set to one of these character values: **0** Can not be received **1** Can be received **3** Has been received **4** Has been received more than once.
5	'm'	The minimum number of shares required to instantiate a master key through the master key shares process. The value is returned in two characters, valued from 01 to 15, followed by six space characters.
6	'n'	The maximum number of distinct shares involved in the master key shares process. The value is returned in two characters, valued from 01 to 15, followed by six space characters.
Output rule array for option STATEID		
1,2	EID, Environment Identifier	The two elements when concatenated provide the 16-byte EID value.
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), and DD represents the day of the month (01-31).
2	Time	The current GMT time of day is returned as a character string of the form HHMMSS.
3	Day of the week	The day of the week is returned as a number between 1 (Sunday) and 7 (Saturday).

verb_data_length
The *verb_data_length* parameter is a pointer to an integer variable containing the length (in bytes) of data in the *verb_data* field.

verb_data
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 the verb. Its use depends on the options specified by the host application program.

The *verb_data* parameter is not currently used by this verb.

Required Commands

Cryptographic_Facility_Query is a universally-authorized verb. There are no access control restrictions on its use.

Key_Storage_Initialization (CSNBKSI)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The Key_Storage_Initialization verb can initialize a key storage file using the current master key.

Restrictions

None

Format

CSNBKSI			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_storage_file_name_length	Input	Integer	
key_storage_file_name	Input	String	key_storage_file_name_length bytes
key_storage_description_length	Input	Integer	≤64
key_storage_description	Input	String	key_storage_description_length bytes
clear_master_key	Input	String	

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* variable must be two for this verb.

rule_array

The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Specify a master key source (One required)	
CURRENT	Specifies that the current master key of the default cryptographic facility is to be used for the initialization.
Key Storage Selection (One, required)	
DES	Initialize the DES key storage.
PKA	Initialize the PKA key storage.

key_storage_file_name_length
The *key_storage_file_name_length* parameter is a pointer to an integer variable containing length of the key_storage_file_name variable. The length must be within the range of 1 to 64 bytes.

key_storage_file_name
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
The *key_storage_description_length* parameter is a pointer to an integer containing the length of the key_storage_description variable.

key_storage_description
The *key_storage_description* parameter is a pointer to a string variable containing description string that is stored in the key storage file after it is initialized.

clear_master_key
The *clear_master_keys* parameter is unused, but it must be declared and point to data in application storage.

Required Commands

The Key_Storage_Initialization verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the hardware.

Logon_Control (CSUALCT)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

Use the Logon_Control verb to perform the following services:

- Log on to the coprocessor, using your access control profile
- Log off of the coprocessor.
- Save or restore logon content information.

Select the service to perform by specifying the corresponding keyword in the input rule array. Only one service is performed for each call to this verb.

If you log on to the adapter when you are already logged on, the existing logon session is replaced with a new session.

Restrictions

None.

Format

CSUALCT			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
user_id	Input	String	8 bytes
auth_parm_length	Input	Integer	
auth_parm	Input	String	auth_parm_length bytes
auth_data_length	Input	Integer	
auth_data	Input	String	auth_data_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be one or two for this verb.

rule_array

The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

On input, you put keywords into the rule array to specify the operation to perform. The rule array keywords are shown below:

Figure 2-9. CSUALCT Rule_Array Input Keywords	
Keyword	**Meaning**
Keywords used to log on	
LOGON	Tells the coprocessor that you want to log on. When you use the LOGON keyword, you must also use a second keyword, PPHRASE, to indicate how you will identify yourself to the coprocessor.
PPHRASE	Specifies that you are going to identify yourself using a *passphrase*.
Keywords used to log off	
LOGOFF	Tells the coprocessor you want to log off.
FORCE	Tells the coprocessor that a specified user is to be logged off. The user's profile ID is specified by the *user_id* parameter.
Keywords used to save and restore logon context information	
GET-CNTX	Obtains a copy of the logon context information that is currently active in your session. See "The use of Logon Context information" on page 2-40
PUT-CNTX	Restores the logon context information that was saved using the GET_CNTX keyword. See "The use of Logon Context information" on page 2-40

user_id

The *user_id* parameter is a pointer to an eight-character string variable containing the id string which identifies the user to the system. The user id must be exactly eight characters in length; shorter user ids should be padded on the right with space characters.

The *user_id* parameter is always used when logging on. It is also used when the **LOGOFF** keyword used in conjunction with the **FORCE** keyword to force a user off.

auth_parm_length

The *auth_parm_length* parameter is a pointer to an integer variable containing the length (in bytes) of data in the auth_parm variable.

On input, this variable contains the length (in bytes) of the auth_parms variable.. On output, this variable contains the number of bytes of data returned in the auth_parms variable.

auth_parms

The *auth_parms* parameter is a pointer to a string variable containing data used in the authentication process.

This field is used differently depending of the authentication method specified in the rule array. Figure 2-10 shows the content of this field for each of the authentication methods.

Figure 2-10. Contents of the authentication parameters field	
Keyword	**Contents of *auth_parms* field**
PPHRASE	The authentication parameter field is empty. Its length is zero.

auth_data_Length

The *auth_data_length* parameter is a pointer to an integer variable containing the length (in bytes) of the data in the auth_data variable

On input, this field contains the length (in bytes) of the auth_data variable. On output, this field contains the number of bytes of data returned in the auth_data variable.

auth_data

The *auth_data* parameter is a pointer to a string variable containing data used in the authentication process.

This field is used differently depending on the keywords specified in the rule_array. Figure 2-11 shows the content of this field for each of the authentication methods.

Figure 2-11. Contents of the authentication data field

Keyword	Contents of *auth_data* field
PPHRASE	The authentication data field contains the user-provided passphrase.
GET-CNTX	The authentication data field receives the active logon context information. The size of the buffer provided for the auth_data field must be at least 256 bytes.
PUT-CNTX	The authentication data field contains your active logon context,

The use of Logon Context information

When logging on to the cryptographic coprocessor, a session is established between your application program and the coprocessor's access control system. The Security Application Program Interface (SAPI) holds the logon context information, which contains the session information needed by the host computer to protect and validate transactions sent to the coprocessor.

This logon context information resides in memory owned by your application and is lost when the application ends. If your application is made up of multiple programs which are separately executed, you must make this the logon context information available to each program. The Logon Control verb offers this capability through the **GET-CNTX** and **PUT-CNTX** keywords.

The **GET-CNTX** keyword is used to retrieve a copy of your logon context information, which you can store until another program needs it. The **PUT-CNTX** keyword is used to give the context information back to the API, allowing it to continue with your logon session. If the context is not saved and restored, the coprocessor thinks you are still logged on, but the API does not.

As an example, consider a simple application which contains two programs.

- The program **LOGON** logs you on to the coprocessor using your passphrase.
- The program **ENCRYPT** encrypts some data. The roles defined for your system require you to be logged on in order to use the **ENCIPHER** function.

These programs will must use the **GET-CNTX** and **PUT-CNTX** keywords in order to work properly. They should work as follows.

LOGON

1. Log the user on to the coprocessor using **CSUALCT** verb with the **PPHRASE** keyword.
2. Retrieve the logon context information using **CSUALCT** with the **GET-CNTX** keyword.
3. Save the logon context information in a place that will be available to the ENCIPHER program. This could be as simple as a disk file, or it could be something more complicated such as shared memory or a background process.

ENCIPHER

1. Retrieve the logon context information saved by the **LOGON** program.
2. Restore the logon context information to SAPI using the **CSUALCT** verb with the **PUT-CNTX** keyword.
3. Encipher the data.

You ***only*** need to be concerned about the logon context information if you log on to the coprocessor using one program, then make use of the coprocessor with one or more additional programs.

CAUTION:
You should take care in storing the logon context information. Design your software so that the saved context is protected from disclosure to others who may be using the same computer. If someone is able to obtain your logon context information, they may be able to impersonate you for the duration of your logon session.

Required Commands

The Logon_Control verb requires the Force User Logoff of a specified user command (offset X'011B') to be enabled in the hardware for use with the **FORCE** keyword.

Master_Key_Distribution (CSUAMKD)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The Master_Key_Distribution verb is used to perform these operations related to the distribution of shares of the master key:

- Generate and distribute a share of the current master-key
- Receive a master key share. When sufficient shares are received, reconstruct the master key in the new master-key register.

OBTAIN and **INSTALL** rule array keywords control the operation of the verb.

With the **OBTAIN** keyword...

- You specify:
  - The share number, i; 1 ≤ i ≤ 15 and i ≤ the maximum number of shares to be distributed as defined with the **SET-MOFN** option in the Cryptographic_Facility_Control verb.
  - The private-key_name of the coprocessor-retained key used to sign a generated master key share
  - The certifying_key_name of the public key already registered in the Coprocessor used to validate the following certificate
  - The certificate and its length that provides the public key used to encrypt the clone_information_encrypting_key
  - The length and location of the clone_information field that will receive the encrypted cloning information (master key share).
- The verb performs:
  - Generation of master key shares, as required, and formatting of the information to be cloned
  - Signing of the cloning_information
  - Generation of an encryption key and encryption of the cloning information
  - Recovery and validation of the public key used to encrypt the clone_information_encrypting_key
  - Encryption of the clone_information_encrypting_key.
- The verb returns:
  - The encrypted cloning information
  - The encrypted clone_information_encrypting_key.

With the **INSTALL** keyword...

- You specify:
  - The share number, ‘i’, presented in this request
  - The private_key_name of the coprocessor-retained key used to decrypt the clone_information_encrypting_key
  - The certifying_key_name of the public key already registered in the Coprocessor used to validate the following certificate
  - The certificate and its length that provides the public key used to validate the signature on the cloning information

  - The length and location of the clone_information field that provides the encrypted cloning information (master key share).

- The verb performs:
  - Recovery of the clone_information_encrypting_key
  - Decryption of the cloning information
  - Recovery and validation of the public key used to validate the cloning information signature
  - Validation of the cloning information signature
  - Retention of a master-key share
  - Regeneration of a master key in the new master key register when sufficient shares have been received.

- The verb returns:
  - A return code valued to 4 if the master key has been recovered into the new master key register. A return code of zero indicates that processing was normal, but a master key was not recovered into the new master key register. (Other return codes, and various reason codes can also occur in abnormal cases.)

Restrictions

When using the **OBTAIN** keyword, the current master key register must be full.

When using the **INSTALL** keyword, the new master key register must be clear (empty).

Format

CSUAMKD			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
share_index	Input	Integer	
private_key_name	Input	String	64 bytes
certifying_key_name	Input	String	64 bytes
certificate_length	Input	Integer	
certificate	Input	String	
clone_info_encrypting_key_length	Inp/Outp	Integer	
clone_info_encrypting_key	Inp/Outp	String	
clone_info_length	Inp/Outp	Integer	
clone_info	Inp/Outp	String	

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable.

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

On input, the keywords in the rule array specify the operation being performed. The rule array keywords are shown below:

Keyword	Meaning
OBTAIN	Generate and output a master key share and other cloning information.
INSTALL	Receive a master key share and other cloning information.

share_index
The share_index parameter points to an integer valued to the number of the share to be generated or received by the coprocessor.

private_key_name
The private_key_name parameter points to a 64-byte string variable that contains the name of the Coprocessor-retained private key used to sign the cloning information (OBTAIN mode), or recover the cloning information encrypting key (INSTALL mode).

certifying_key_name
The certifying_key_name parameter points to a 64-byte string variable that contains the name of the Coprocessor-retained public key used to verify the offered certificate.

certificate_length
The certificate_length parameter points to an integer variable set to the length of the public key certificate.

certificate
The certificate parameter points to a string variable containing the certificate that can be validated using the public key identified with the certifying_key_name variable.

clone_info_encrypting_key_length
The clone_info_encrypting_key_length parameter points to an integer variable containing the length of the clone_info_encrypting_key variable.

clone_info_encrypting_key
The clone_info_encrypting_key parameter points to a string variable containing the encrypted key used to recover the cloning information.

clone_info_length
The clone_info_length parameter points to an integer variable containing the length of the clone_info variable.

clone_info
The clone_info parameter points to a string variable containing the encrypted cloning information (master key share).

Required Commands

The Master_Key_Distribution verb requires the following commands to be enabled based on the requested share-number, 1≤i≤15, and the use of either the **OBTAIN** or the **INSTALL** rule array keyword:

- Clone-info Obtain command (offset X'0210'+share_index, e.g. for share 10, X'021A')

- Clone-info Install command (offset X'0220'+share_index, e.g. for share 12, X'022C').

Master_Key_Process (CSNBMKP)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The Master_Key_Process verb operates on the three master key registers: new, current, and old. Use the verb to:

- Clear the new and clear the old master key registers
- Generate a random master key value in the new master key register
- Exclusive-OR 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, the new master key to the current master key register, and clears the new master key register. **SET** also clears the master-key shares tables.

For IBM 4758 Cryptographic Coprocessor implementations, the master key is a triple length, 168-bit, 24-byte value.

Before starting to load new master key information, ensure that the new master key register is cleared, 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.

You can remove a prior master key from the Coprocessor with the **CLR-OLD** keyword; the contents of the old master key register are removed AND subsequently only current-master-key encrypted keys will be usable. If there is a value in the old master key register, this master key can also be used to decrypt an enciphered working key.

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 should 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 key value.

When the **LAST** master key part is entered, this additional processing is performed:

- If any two of the eight-byte parts of the *new* master key have the same value, a warning is issued. *This warning should not be ignored* and a key with this property should not be used.
- The key-token master key verification pattern (MKVP) of the *new* master key is compared against the key-token MKVP of the *current* and the *old* master keys. If they are the same, the service is failed.

- If any of the eight-byte parts of the *new* master key compares with one of the weak DES keys, the service is failed. See page 2-48 for a list of these "weak" keys.

As part of the **SET** process, if a DES and/or PKA key storage exists, the header record of each key storage is updated with the verification pattern of the (new) current master key.

Restrictions

Only the IBM 4758 implementations support the **SET** keyword and treat the **ADAPTER** keyword as a default.

Format

CSNBMKP

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_part	Input	String	

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable. The value of the rule_array_count variable may be 1 or 2 for this verb.

rule_array
The *rule_array* parameter is a pointer to 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 rule_array keywords are shown below:

Keyword	Meaning
Cryptographic Component (optional)	
ADAPTER	Specifies the coprocessor. This is the default for IBM 4758 implementations.

Keyword	Meaning
Master Key Process	(one required)
CLEAR	Specifies to clear the new master key register.
CLR-OLD	Specifies to clear the old master key register and set the status for this register to empty. You can use the **CLR-OLD** keyword to cause the old master key register to be cleared. The status response in the Cryptographic_Facility_Query verb, STATCCA, shows the condition of this register.
FIRST	Specifies to load the first key_part.
MIDDLE	Specifies to XOR the second, third, or other intermediate key_part into the new master key register.
LAST	Specifies to XOR the last key_part into the new master key register.
RANDOM	Causes generation of a random master key value in the new master key register.
SET	Specifies to advance the current master key to the old master key register, to advance the new master key to the current master key register, and to clear the new master key register. This keyword is valid only with IBM 4758 implementations.

key_part

The key_part parameter is a pointer to a string variable containing a 168-bit (3x56-bit, 24-byte) clear key_part.

If you use the **CLEAR** or **SET** keywords, the information in the variable is ignored, but you must declare the variable.

Required Commands

The Master_Key_Process verb requires the following commands to be enabled in the hardware:

- Clear New Master Key Register command (offset X'0032') with the **CLEAR** keyword
- Clear Old Master Key Register command (offset X'0033') with the **CLR-OLD** keyword
- Load First Master Key Part command (offset X'0018') with the **FIRST** keyword
- Combine Master Key Parts command (offset X'0019') with the **MIDDLE** or **LAST** keywords
- Generate Random Master Key command (offset X'0020') with the **RANDOM** keyword
- Set Master Key command (offset X'001A') with the **SET** keyword.

Related Information

The following are considered questionable DES keys:

```
01 01 01 01 01 01 01 01
FE FE FE FE FE FE FE FE
1F 1F 1F 1F 0E 0E 0E 0E
E0 E0 E0 E0 F1 F1 F1 F1
```

```
01 FE 01 FE 01 FE 01 FE  /* semi-weak */
FE 01 FE 01 FE 01 FE 01  /* semi-weak */
1F E0 1F E0 0E F1 0E F1  /* semi-weak */
E0 1F E0 1F F1 0E F1 0E  /* semi-weak */
01 E0 01 E0 01 F1 01 F1  /* semi-weak */
E0 01 E0 01 F1 01 F1 01  /* semi-weak */
1F FE 1F FE 0E FE 0E FE  /* semi-weak */
FE 1F FE 1F FE 0E FE 0E  /* semi-weak */
01 1F 01 1F 01 0E 01 0E  /* semi-weak */
1F 01 1F 01 0E 01 0E 01  /* semi-weak */
E0 FE E0 FE F1 FE F1 FE  /* semi-weak */
FE E0 FE E0 FE F1 FE F1  /* semi-weak */
1F 1F 01 01 0E 0E 01 01  /* possibly semi-weak  */
01 1F 1F 01 01 0E 0E 01  /* possibly semi-weak  */
1F 01 01 1F 0E 01 01 0E  /* possibly semi-weak  */
01 01 1F 1F 01 01 0E 0E  /* possibly semi-weak  */
E0 E0 01 01 F1 F1 01 01  /* possibly semi-weak  */
FE FE 01 01 FE FE 01 01  /* possibly semi-weak  */
FE E0 1F 01 FE F1 0E 01  /* possibly semi-weak  */
E0 FE 1F 01 F1 FE 0E 01  /* possibly semi-weak  */
FE E0 01 1F FE F1 01 0E  /* possibly semi-weak  */
E0 FE 01 1F F1 FE 01 0E  /* possibly semi-weak  */
E0 E0 1F 1F F1 F1 0E 0E  /* possibly semi-weak  */
FE FE 1F 1F FE FE 0E 0E  /* possibly semi-weak  */
FE 1F E0 01 FE 0E F1 01  /* possibly semi-weak  */
E0 1F FE 01 F1 0E FE 01  /* possibly semi-weak  */
FE 01 E0 1F FE 01 F1 0E  /* possibly semi-weak  */
E0 01 FE 1F F1 01 FE 0E  /* possibly semi-weak  */
01 E0 E0 01 01 F1 F1 01  /* possibly semi-weak  */
1F FE E0 01 0E FE F0 01  /* possibly semi-weak  */
1F E0 FE 01 0E F1 FE 01  /* possibly semi-weak  */
01 FE FE 01 01 FE FE 01  /* possibly semi-weak  */
1F E0 E0 1F 0E F1 F1 0E  /* possibly semi-weak  */
01 FE E0 1F 01 FE F1 0E  /* possibly semi-weak  */
01 E0 FE 1F 01 F1 FE 0E  /* possibly semi-weak  */
1F FE FE 1F 0E FE FE 0E  /* possibly semi-weak  */
E0 01 01 E0 F1 01 01 F1  /* possibly semi-weak  */
FE 1F 01 E0 FE 0E 01 F1  /* possibly semi-weak  */
FE 01 1F E0 FE 01 0E F1  /* possibly semi-weak  */
E0 1F 1F E0 F1 0E 0E F1  /* possibly semi-weak  */
FE 01 01 FE FE 01 01 FE  /* possibly semi-weak  */
E0 1F 01 FE F1 0E 01 FE  /* possibly semi-weak  */
E0 01 1F FE F1 01 0E FE  /* possibly semi-weak  */
FE 1F 1F FE FE 0E 0E FE  /* possibly semi-weak  */
1F FE 01 E0 E0 FE 01 F1  /* possibly semi-weak  */
01 FE 1F E0 01 FE 0E F1  /* possibly semi-weak  */
1F E0 01 FE 0E F1 01 FE  /* possibly semi-weak  */
01 E0 1F FE 01 F1 0E FE  /* possibly semi-weak  */
01 01 E0 E0 01 01 F1 F1  /* possibly semi-weak  */
1F 1F E0 E0 0E 0E F1 F1  /* possibly semi-weak  */
1F 01 FE E0 0E 01 FE F1  /* possibly semi-weak  */
01 1F FE E0 01 0E FE F1  /* possibly semi-weak  */
1F 01 E0 FE 0E 01 F1 FE  /* possibly semi-weak  */
01 1F E0 FE 01 E0 F1 FE  /* possibly semi-weak  */
01 01 FE FE 01 01 FE FE  /* possibly semi-weak  */
1F 1F FE FE 0E 0E FE FE  /* possibly semi-weak  */
FE FE E0 E0 FE FE F1 F1  /* possibly semi-weak  */
E0 FE FE E0 F1 FE FE F1  /* possibly semi-weak  */
```

```
FE E0 E0 FE FE F1 F1 FE  /* possibly semi-weak  */
E0 E0 FE FE F1 F1 FE FE  /* possibly semi-weak  */
```

Chapter 3. RSA Key Administration

Figure 3-1. Public-Key Key-Administration Services

Verb	Page	Service	Entry Point	Svc Lcn
PKA_Key_Generate	3-6	Generates a public-private RSA key-pair.	CSNDPKG	E
PKA_Key_Import	3-10	Imports a public-private public key key-pair.	CSNDPKI	E
PKA_Key_Token_Build	3-12	Builds a public key key token.	CSNDPKB	S
PKA_Key_Token_Change	3-18	Re-encipher an RSA private key from the old master key to the current master key.	CSNDKTC	E
PKA_Public_Key_Extract	3-20	Extracts a public key from a public-private public key token.	CSNDPKX	S
PKA_Public_Key_Hash_Register	3-22	Register the hash of a public key used later to verify an offered public key, see PKA_Public_Key_Register.	CSNDPKH	E
PKA_Public_Key_Register	3-24	Register a public key used later to verify an offered public key. Registration requires that a hash of the public key has previously been rigistered within the Coprocessor, see PKA_Public_Key_Hash_Register.	CSNDPKR	E
Service location (Svc Lcn): E=Cryptographic Engine, S=Security API software				

This chapter describes the management of RSA public and private keys and how you can:

- Generate keys with various characteristics
- How you can receive keys from other systems
- How a private key can be protected and moved from one node to another

The verbs listed in Figure 3-1 are used to perform cryptographic functions and assist you to obtain key_token structures.

RSA Key Management

This implementation of the CCA, and many others, support a set of public key cryptographic services that are collectively designated *PKA96*. The PKA96 services support the RSA public key algorithm and related hashing methods including MD5 and SHA-1. Figure 3-2 on page 3-2 shows the relationship among the services, the public-private key_token, and other data involved with supporting digital signatures and symmetric (DES) key exchange.

These topics are discussed in this section:

- How you can generate an RSA key pair
- How you can receive keys from other systems
- How you can update a private key when the master key that protects a private key is changed
- How you can use the RSA keys and provide for their protection
- How you can use a private key at multiple nodes
- How you can register and retain a public key.

© Copyright IBM Corp. 1997-98

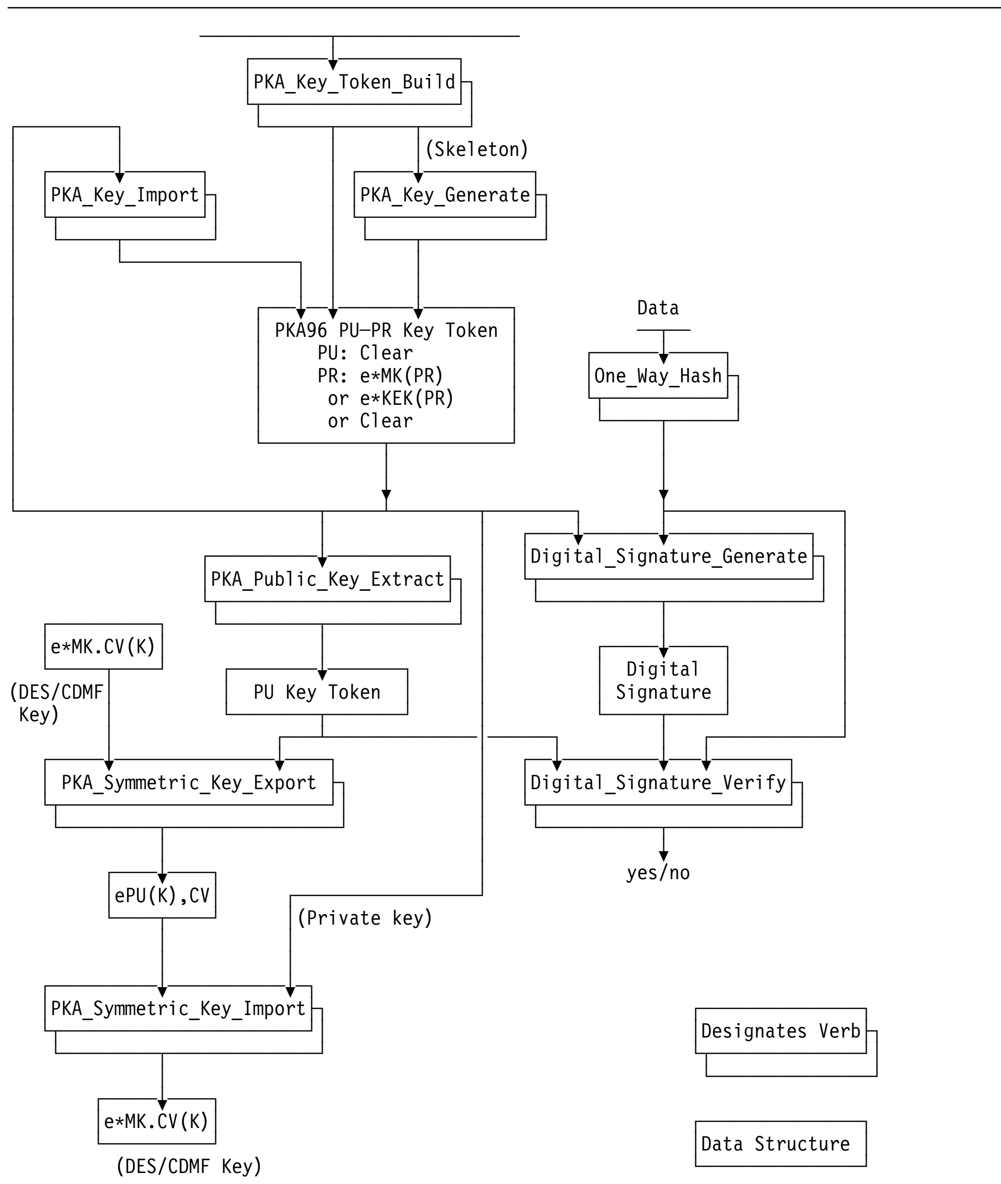

Figure 3-2. PKA96 Verbs with Key Token Flow

Key Generation

RSA public and private key-pairs can be generated with the PKA_Key_Generate verb. When generating the key-pair you must determine:

- The key-length
- The encryption of the private key (to control where the key can be used)
- An association with a key name that in some systems can limit key usage
- Whether the key should be usable in symmetric key-exchange operations
- The form of the private key: modular-exponent or Chinese Remainder
- Where the private key can reside
- How, or if, the private key should be encrypted.

All but the last two items are determined by the *skeleton_key_token* that you supply to the verb. A *skeleton_key_token* is prepared using the PKA_Key_Token_Build verb.

This PKA_Key_Generate verb outputs the generated secret key in one of three forms so you can control where the private key is employed:

- Cleartext

  Both the RSA private and public keys are returned as clear text. This option requires that you provide protection for the private key by means other than encryption within the key-generating step. This option is provided so the user can test, or interface with, other systems or applications that require the private key to be in the clear.

- Enciphered by the local master key

  You can request that the key-generating service return the private key enciphered by the master key within the cryptographic engine. Since there is no service available to re-encrypt the private key other than the current or a replacement master key, the generated private key is effectively locked to the generating node.

- Enciphered by a transport Key-Encrypting Key

  You can request the service to encrypt the generated private key under either an IMPORTER key or an EXPORTER key. An IMPORTER key will permit the private key to be imported and used later at the generating node.

  Or, the Key-Encrypting Key can be an EXPORTER transport key. An EXPORTER key is shared with one or more nodes. This allows you to distribute the key to another node(s). For example, you could obtain a private key in this form for distribution to a S/390 large server's integrated RSA cryptographic processor, as that processor can not generate private keys in an encrypted form.

  Note: EXPORTER and IMPORTER key-encrypting “transport” keys are discussed in Chapter 5, “Basic CCA DES Key Management.”

| You can also request that the generated private key be retained within the
| secure cryptographic engine through the use of the **RETAIN** keyword on the
| PKA_Key_Generate verb. In this case, only the public key is returned. You use
| the retained private key by referring to it with a key label.

Because you can obtain the private key, it can be made functional on more than one cryptographic engine and used for backup or additional throughput. Your administration procedures control where the key can be used. The private key can be transported securely between nodes in its encrypted form. You can set up one-way key distribution channels between nodes and lock the private-key receiving key to a particular node or nodes so that you can be certain where the private key exists. This ability to replicate a key to multiple nodes is especially important to high-throughput server systems and important for backup processing purposes.

In systems with an access monitor like RACF on S/390 large servers, the key_name that you associate with a private key gives you the ability to enforce restricted key usage. These systems can determine if a requesting process has the right to use the particular key-name that is cryptographically bound to the private key. You specify such a key-name when you build the *skeleton_key_token* in the PKA_Key_Token_Build verb.

You decide if the key should be returned in modular-exponent form or as a series of numbers for use in the Chinese-Remainder-Theorem (CRT) form which generally yields faster performance in key-using services. This decision is represented by the form of the private key that you indicate in the skeleton_key_token. You can reuse an existing key_token having the desirable properties, or you can build the skeleton_key_token with the PKA_Key_Token_Build verb. Not all systems can employ a private key in the CRT form generated by the PKA_Key_Generate verb. In particular the S/390 large server integrated cryptographic feature requires the private key in modular-exponent form.

The characteristics of the generated key including key length are specified in a *skeleton_key_token*. You specify the key-length (modulus length) and decide if the public exponent should be valued to three, $2^{16}+1$, or fully random. Also, in the PKA_Key_Token_Build verb you can indicate that the key should be usable for both digital signature signing *and* symmetric key exchange, or you can indicate that the key should be usable only for digital signature signing.

The key can be generated as a random value, or the key can be generated based on a seed derived from *regeneration data* provided by the application program.

Key Import

To be useful, an RSA private key must be enciphered by a master key on the CCA node where it will be used to sign a digital signature or to receive a symmetric key in a key-exchange scenario. You can use the PKA_Key_Import verb to get a private key deciphered from a transport key and enciphered by the master key. Also, you can get a clear (unenciphered) private key enciphered by the master key using the *PKA_Key_Import* verb.

The public and private RSA keys must be presented in a PKA external key-token (see "RSA Key Token Formats" on page B-5). You can use the PKA_Key_Token_Build verb to structure the key into the proper token format.

You provide or identify the operational transport key (Key-Encrypting Key) and the encrypted private key with its associated public key to the import service. The service will return the private key encrypted under the master key along with the public key.

Re-enciphering a Private Key Under an Updated Master Key

When the master key at a CCA node is changed, operational keys, such as RSA private keys enciphered by the master key, must be securely decrypted from under the preexisting master key and enciphered under the replacement master key. You can accomplish this task using the PKA_Key_Token_Change verb.

After the preexisting master key has become the old-master key and the replacement master key has become the current-master key, you use the PKA_Key_Token_Change verb to effect the re-encipherment of the private key. (You use the Master_Key_Process verb to *set* the master key.)

Using the RSA Keys

The RSA keys that you create (generate) or import can be used in these services:

For Private keys, see:
- Digital_Signature_Generate, 4-4
- PKA_Symmetric_Key_Import, 5-52

For Public keys, see:
- Digital_Signature_Verify, 4-7
- PKA_Symmetric_Key_Export, 5-47
- PKA_Symmetric_Key_Generate, 5-49

You must arrange appropriate protection for the RSA private key. A CCA node can help ensure that the key will remain confidential. However, you must ensure that the master key and any transport keys are protected, usually through split-knowledge, dual-control procedures. Or, you can choose to retain the private key in the secure cryptographic engine.

Besides the confidentiality of the private key, you must also ensure that only authorized applications can use the private key. You can hold the private key in application-managed storage and pass the key to the cryptographic services as required. This will generally limit the access other applications might have to the key. In systems with an access monitor, such as RACF on MVS systems, it is possible to associate a *key name* with the private key and have use of the key-name authorized by the access monitor.

Using the Private Key at Multiple Nodes

You can arrange to use a private key at multiple nodes if the nodes have the same master key, or if you arrange to have the same transport key installed at each of the target nodes. In the latter case, you need to arrange to have the transport key under which the RSA private key is enciphered installed at each target node.

Having the private key installed at multiple nodes enables you to provide increased service levels for greater throughput, and to maintain operation when a primary node goes out of service. Of course, having a private key installed at more than one node increases the risk of someone misusing or compromising the key. You have to weigh the advantages and disadvantages as you design your system or systems.

Registering and Retaining a Public Key

You can use the PKA_Public_Key_Hash_Register and the PKA_Public_Key_Register verbs to “register” a public key in the secure cryptographic engine under dual control. Authorize the related commands in two different roles to enforce a dual control policy. Your applications can subsequently reference the public key logged within the engine with the confidence that the key has been entered under dual control. Note that the Master_Key_Distribution verb makes use of registered public keys in the master-key shares distribution scheme.

PKA_Key_Generate (CSNDPKG)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The PKA_Key_Generate verb is used to generate a public-private key-pair for use with the RSA algorithm.

The *skeleton_key_token* specified by the verb determines the following characteristics of the generated key-pair:

- The key type: RSA
- The RSA key length (modulus size)
- The RSA public key exponent, valued to 3, 2^{16}+1, or random
- Any RSA private key optimization (modulus-exponent vs “Chinese Remainder” form)
- Any signatures and signature-information that should be associated with the public key

The skeleton_key_token can be created using the PKA_Key_Token_Build verb.

The key is normally randomly generated. By providing “regeneration data,” a seed can be derived so that the same value of the generated key can be obtained in multiple instances. This may be useful in testing situations or where the regeneration data can be securely held for key generation. The process for generating a particular key-pair from regeneration data may vary between product implementations, therefore you should not rely on obtaining the same key-pair for a given regeneration data string between products.

The generated private key can be returned in one of three forms:

- In cleartext form
- Enciphered by the master key of the local node
- Enciphered by a transport key, either a DES IMPORTER or DES EXPORTER Key-Encrypting Key. If the private key is enciphered by an IMPORTER key, it can be imported to the generating node. If the private key is enciphered by an EXPORTER key, it can be imported to a node where the matching IMPORTER key is installed.

  Using the **RETAIN** rule array keyword you can cause the private key to be retained within the cryptographic engine. You place the name by which you will later reference the newly generated key in the “key name” section of the skeleton key token. Later, you use this key name to employ the key in verbs such as Digital_Signature_Generate, &vbnmski., Master_Key_Distribution, and SET_Block_Decompose. The generated_key_identifier variable will not contain the private key key-token section.

Rule array keyword **CLONE** marks a generated and retained private key to be flagged as usable in an engine “cloning” process. Cloning is a technique for copying sensitive adapter information from one adapter to another.

If you include a *public-key certificate section* within the skeleton key token, you cause the cryptographic engine to sign a certificate with the key that is

| designated in the *public-key certificate signature subsection*. Using this
| technique, you can cause the cryptographic engine to sign the newly generated
| public key using another key that has been retained within the engine, including
| the newly generated key (producing a "self-signature"). You can obtain more
| than one signature on the public key when yuou include multiple signature
| subsections in the skeleton key token.

Restrictions

- Not all IBM implementations of PKA96 verbs may support an optimized form of the RSA private key; check the product-specific literature. The Fortress product family implementation of PKA96 supports an optimized RSA private key (a key in "Chinese Remainder" form).
- When the private key is enciphered for use at another node, determine that the control vector values used with the transport key are compatible with permissible control vector values at the receiving node.

Format

CSNDPKG

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
regeneration_data_length	Input	Integer	
regeneration_data	Input	String	regeneration_data_length bytes
skeleton_key_token_length	Input	Integer	
skeleton_key_token	Input	String	skeleton_key_token_length bytes
transport_key_identifier	Input	String	
generated_key_identifier_length	In/Output	Integer	
generated_key_identifier	In/Output	String	generated_key_identifier_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Private Key Encryption (One keyword required)	
MASTER	The private key should be enciphered under the master key. The transport_key_token should specify a null key_token.
XPORT	The private key should be enciphered under the IMPORTER or EXPORTER key-encrypting key identified by the *transport_key_token* parameter.
CLEAR	The private key is returned in cleartext.
RETAIN	The private key is retained within the cryptographic engine and the public key is returned in the generated_key_identifier variable. The name presented in the generated_key_identifier variable is used later to access the retained private key.
Options (Optional Keywords)	
CLONE	A retained private key is flagged as usable in a cryptographic engine "cloning" operation.

regeneration_data_length
The *regeneration_data_length* parameter is a pointer to an integer variable containing the length (in bytes) of the regeneration data. This must be a value of zero, or in the range 8 to 256. If the value is zero, the generated keys will be based on a random seed value. If this value is between 8 and 256, the regeneration data will be hashed to form a seed value used in the key generation process to provide a means for recreating a public-private key pair.

regeneration_data
The regeneration_data parameter is a pointer to a string variable containing a string used as the basis for creating a particular public-private key pair in a repeatable manner. The regeneration data will be hashed to form a seed value used in the key generation process and provides a means for recreating a public-private key pair.

skeleton_key_token_length
The *skeleton_key_token_length* parameter is a pointer to an integer variable containing at least the length (in bytes) of the field containing the skeleton_key_token. The maximum size is 2500 bytes.

skeleton_key_token
The *skeleton_key_token* parameter is a pointer to a string containing a skeleton_key_token. This information provides the characteristics for the PKA key-pair to be generated. A skeleton_key_token can be created using the PKA_Key_Token_Build verb.

transport_key_identifier
The *transport_key_identifier* parameter is a pointer to a string containing an internal Key-Encrypting Key token or a key label of an internal Key-Encrypting Key token, or a null key token. If the rule_array keyword is not **XPORT**, this parameter should point to a null key token. Otherwise, the specified key enciphers the private key and can be an IMPORTER or an EXPORTER key type. Use an IMPORTER key to encipher a private key to

be used at this node. Use an EXPORTER key to encipher a private key to be used at another node.

generated_key_identifier_length
The *generated_key_identifier_length* parameter is a pointer to an integer variable containing at least the length (in bytes) of the field containing the target private key token or key label. The maximum size is 2500 bytes. On output, and if the field size is of sufficient length, the variable is updated with the actual length of the *generated_key_token*.

Generated_key_identifier
The *generated_key_identifier* parameter is a pointer to a string variable containing either a key label identifying a key storage record, or is other information that will be overwritten. If the key label identifies a key record in key storage, the generated key token will replace any key token associated with the label. If the first byte of the identified string did not indicate a key label (not in the range X'20' to X'FE'), and the field is of sufficient length to receive the result, then the generated key token will be returned in the identified field.

Required Commands

Enable one of these commands in the hardware depending on rule array keyword usage and the content of the skeleton key token:

- With the **CLONE** rule-array keyword, the PKA Clone Key Generate command (offset X'0204')
- With the **CLEAR** rule-array keyword, the PKA Clear Key Generate command (offset X'0204')

Otherwise the PKA Key Generate command (offset X'0103')

PKA_Key_Import (CSNDPKI)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The PKA_Key_Import verb is used to import a public-private key-pair or a public-only key. When a private key is present, the associated public key must be present also. A source private key may be in the clear or it may be enciphered.

Generally, the PKA_Key_Generate verb will be the source of the key token imported with this verb. The PKA_Key_Token_Build verb may be helpful in creating the source_key_token if the key originates in a non-CCA system.

If the source private key is enciphered, the verb will decipher the private key using the DES IMPORTER key identified by the *transport_key_identifier*.

The imported keys are returned in the target_key_token. A public-private key-pair is returned in an internal key_token with the private key enciphered by the master key. If only a public key is imported, the key is returned in an external key_token.

Restrictions

- Not all IBM implementations of PKA96 verbs may support an optimized form of the RSA private key; check the product-specific literature. The Fortress product family implementation of PKA96 supports an optimized RSA private key (a key in “Chinese Remainder” form).
- Not all IBM implementations of this verb support the use of a key label with the target key identifier; check the product-specific literature.

Format

CSNDPKI			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
source_key_token_length	Input	Integer	
source_key_token	Input	String	source_key_token_length bytes
transport_key_identifier	Input	String	
target_key_identifier_length	In/Out	Integer	
target_key_identifier	In/Out	String	target_key_identifier_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero for this verb.

rule_array
The *rule_array* parameter is not presently used in this service, but must be specified.

source_key_token_length
The *source_key_token_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field that contains the source key token. The maximum length is 2500 bytes.

source_key_token
The *source_key_token* parameter is a pointer to a string variable containing a PKA96 key token. The key token must contain both public and private RSA key information. The private key can be in cleartext or it can be enciphered.

transport_key_identifier
The *transport_key_identifier* parameter is a pointer to a string variable containing either a key encrypting key token or a key label of a key encrypting key token, or a null key token. This key will be used to decipher an encrypted private key; the designated DES key must be an IMPORTER key type with IMPORT capability enabled in its control vector.

If the source key is not encrypted, a null key token must be specified (the first byte of the key token must be X'00').

target_key_identifier_length
The *target_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field containing the target key token or key label. The maximum length is 2500 bytes. On output, the identified variable will be updated with the actual length of the token.

target_key_identifier
The *target_key_identifier* parameter is a pointer to a string variable containing either a key label identifying a key storage record, is a null token (the first byte is X'00'), or an existing key token. The key label, null key token, or existing key token is overwritten with the imported key.

Required Commands

The PKA_Key_Import verb requires the PKA key import command (offset X'0104') to be enabled in the hardware.

PKA_Key_Token_Build (CSNDPKB)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The PKA_Key_Token_Build verb constructs an RSA PKA96 key token from the information supplied.

This verb is used to create the following:

- A skeleton_key_token for use with the PKA_Key_Generate verb
- A key token with a public key that has been obtained from another node or source
- A key token with a clear private key and the associated public key.

  The verb builds key tokens that support private keys in these forms:

  – 512 to 1024-bit modular-exponentiation format
  – 512 to 2048-bit Chineese-remainder format.

“RSA Key Token Formats” on page B-5 provides the format and content of a PKA96 token for both types of RSA keys. Other than a skeleton_key_token prepared for use with the PKA_Key_Generate verb, every PKA96 token contains public-key information. A PKA96 token can contain private-key information also.

Some implementations may provide special processing for RSA private keys that can be used for distribution of symmetric keys. If an RSA private key will be used to import a symmetric key, include the **KEY-MGMT** keyword in the rule_array.

Restrictions

- The RSA key length is limited to 512 to 2048-bits.
- A *key_name* can be provided only for a key token containing a private key.

Format

CSNDPKB

return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_values_structure_length	Input	Integer	
key_values_structure	Input	String	key_values_structure_length bytes
key_name_length	Input	Integer	
key_name	Input	String	null
reserved_1_length	Input	Integer	
reserved_1	Input	String	null
reserved_2_length	Input	Integer	
reserved_2	Input	String	null
reserved_3_length	Input	Integer	
reserved_3	Input	String	null
reserved_4_length	Input	Integer	
reserved_4	Input	String	null
reserved_5_length	Input	Integer	
reserved_5	Input	String	null
token_length	In/out	Integer	
token	Output	String	token_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be one or two for this verb.

rule_array

The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 3-3 (Page 1 of 2). PKA_Key_Token_Build Rule_Array Keywords

Keyword	Meaning
Required Keyword (One Required)	
RSA-PRIV	Create a key token for an RSA public key and a private key in modulus-exponent form.
RSA-PUBL	Create a key token for an RSA public key in modulus-exponent form.
RSA-OPT	Create a key token for an RSA public key and a key in optimized, Chinese-Remainder form.

Figure 3-3 (Page 2 of 2). PKA_Key_Token_Build Rule_Array Keywords	
Keyword	**Meaning**
RSA Key Usage Control (Select one, optional)	
SIG-ONLY	Indicates that an RSA private key can not be used in symmetric key distribution. This is the default.
KEY-MGMT	Indicates that an RSA private key can be used in distribution of symmetric keys, and in digital signature services.

key_values_structure_length

The *key_values_structure_length* parameter is a pointer to an integer variable containing the length (in bytes) of the structure that contains the key values. The maximum size of the key_Values_structure variable is 2500-bytes.

key_values_structure

The *key_values_structure* parameter is a pointer to a string that is a structure containing the lengths and data for the components of the key or keys. The contents of this structure is shown in Figure 3-4, and sample data is described on page 3-17.

Figure 3-4 (Page 1 of 3). PKA_Key_Token_Build Key Values Structures		
Offset (Bytes)	**Length (Bytes)**	**Description**
RSA Key Values Structure, modulus-exponent form (RSA-PRIV or RSA-PUBL)		
000	002	Length of the modulus in bits (512 to 2048)
002	002	Length of the modulus field, n, in bytes, “nnn.” This value must not exceed 256 for a 2048-bit key. This value should be zero when preparing a skeleton key token for use with the PKA_Key_Generate verb.
004	002	Public exponent field length in bytes, “eee.” This value should be zero when preparing a skeleton key token to generate a random-exponent public key in the PKA_Key_Generate verb. This value must not exceed 256.
006	002	Private exponent field length in bytes, “ddd.” This value can be zero indicating that private key information is not provided. This value must not exceed 256.
008	nnn	Modulus, n, integer value, $1<n<2^{2048}$; n=pq for prime p and prime q.
8+nnn	eee	Public exponent, e, integer value, 1<e<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 three values: 3, 65537 ($2^{16}+1$), or to 0 (zero) to indicate that a full random exponent should be generated. The exponent field can be a null-length field in a skeleton_key_token.
8+nnn +eee	ddd	Private exponent, d, integer value, 1<d<n, $d=e^{-1}\bmod(p-1)(q-1)$.
Note: • All length fields are in binary. • All binary fields (exponents, lengths, etc.) are stored with the high order byte first (left, low-address, S/390 format).		

Figure 3-4 (Page 2 of 3). PKA_Key_Token_Build Key Values Structures

Offset (Bytes)	Length (Bytes)	Description
Optimized RSA Key Values Structure, Chinese Remainder form (RSA-OPT)		
000	002	Length of the modulus in bits (512 to 2048)
002	002	Length of the modulus field, n, in bytes, “nnn.” This value can be zero if the key token will be used as a skeleton_key_token in the PKA_Key_Generate verb. This value must not exceed 256.
004	002	Length of the public exponent, e, in bytes: “eee.” (Can be zero in a skeleton_key_token.)
006	002	Length of the prime number, p, in bytes: “ppp.” (Can be zero in a skeleton_key_token.)
008	002	Length of the prime number, q, in bytes: “qqq.” (Can be zero in a skeleton_key_token.)
010	002	Length of the d_p, in bytes: “rrr.” (Can be zero in a skeleton_key_token.)
012	002	Length of the d_q, in bytes: “sss.” (Can be zero in a skeleton_key_token.)
014	002	Length of the A_p, in bytes: “ttt.” (Can be zero in a skeleton_key_token.)
016	002	Length of the A_q, in bytes: “uuu.” (Can be zero in a skeleton_key_token.)
018	nnn	Modulus, n
018 +nnn	eee	Public exponent, e, integer value, $1<e<n$, 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 one of the following values: 3, 65537 (2^{16}+1), or 0 (zero) to indicate that a full random exponent should be generated. The exponent field can be a null-length field if the exponent value is zero.
018 +nnn +eee	ppp	Prime number, p
018 +nnn +eee +ppp	qqq	Prime number, q
018 +nnn +eee +ppp +qqq	rrr	d_p = d mod(p-1)

Note:

- All length fields are in binary.
- All binary fields (exponents, lengths, etc.) are stored with the high order byte first (left, low-address, S/390 format).

Figure 3-4 (Page 3 of 3). PKA_Key_Token_Build Key Values Structures		
Offset (Bytes)	**Length (Bytes)**	**Description**
018 +nnn +eee +ppp +qqq +rrr	sss	d_q = d mod(q-1)
018 +nnn +eee +ppp +qqq +rrr +sss	ttt	$A_p = q^{p-1}$ mod(n)
018 +nnn +eee +ppp +qqq +rrr +sss +ttt	uuu	$A_q = (n+1-A_p)$
Note: • All length fields are in binary. • All binary fields (exponents, lengths, etc.) are stored with the high order byte first (left, low-address, S/390 format).		

key_name_length
The *key_name_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field containing the optional key_name. If this variable contains zero, the key name section is not included in the target token. If a key name is to be included, the variable must contain 64. A key name is permissible only in a key token that contains a private key.

key_name
The *key_name* parameter is a pointer to a string variable containing the key_name. The key name can consist of the characters A...Z, 0...9, #, $, @, or period (.), and must begin with an alphabetic character.

reserved_x_length(s)
The *reserved_x_length* parameter is a pointer to an integer variable containing the length (in bytes) of a field that is reserved for future use; the variable should contain zero.

reserved_x
The *reserved_x* parameter is a pointer to a string variable. At present, this variable is reserved for future use and this parameter should contain a pointer to a null string.

token_length
The *token_length* parameter is a pointer to an integer variable containing the length (in bytes) of the token field. On output, the length is the length of token returned in the token field. The maximum length is 2500 bytes.

Token

The *token* parameter is a pointer to a string variable to contain the assembled token.

Related Information

Three samples for the *key_value* structure are shown below:

1. The key_value structure for a 1024-bit RSA-PRIV skeleton key token with a public exponent value of 2^{16}+1 for use with the PKA_Key_Generate verb:
   - Expressed as a series of numbers: 1024, 0, 3, 0, [null], 65537, [null]
   - Expressed as a hexadecimal string: X'0400 0000 0003 0000 010001'
2. The key_values structure for a 512-bit RSA-OPT skeleton key token with a public exponent value of 2^{16}+1 for use with the PKA_Key_Generate verb:
   - Expressed as a series of numbers:
     512, 0, 3, 0, 0, 0, 0, 0, 0, [null], 65537, [null], [null], [null], [null], [null], [null]
   - Expressed as a hexadecimal string:
     X'0200 0000 0003 0000 0000 0000 0000 0000 0000 010001'
3. The key_values structure to create a PKA96 RSA key token with a public exponent value of 2^{16}+1 and a provided 1024-bit public key value:
   - Expressed as a series of numbers: 1024, 128, 3, 0, n, 010001, [null]
   - Expressed as a hexadecimal value:
   - X'0400 0080 0003 0000 nnnn...nnnn 010001'

     Where X'nnnn...nnnn' represents the 128-byte modulus bit-string expressed in hexadecimal.

Note: All values in the *key_values* structure must be stored in “big endian” format to ensure compatibility among different computing platforms. “big endian” format specifies the high-order byte be stored at the low address in the field.

Data stored by Intel architecture processors is normally stored in “little endian” format. “Little endian” format specifies the low-order byte be stored in the low address in the field.

Required Commands

None.

PKA_Key_Token_Change (CSNDKTC)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The PKA_Key_Token_Change verb changes RSA keys from encipherment with the old master key to encipherment with the current master key. You identify the task with the rule array keyword, and the internal key token to change with the *key_identifier* parameter.

Note: This verb is similar in function to the CSN**B**KTC Key_Token_Change verb used with DES key tokens.

Restrictions

Certain implementations of CCA may not support this verb.

Format

CSNDKTC			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Inp/Outp	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_identifier_length	In/Out	Integer	
key_identifier	In/Out	String	key_identifier_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 3-5. PKA_Key_Token_Change Rule_Array Keywords

Keyword	Meaning
Required Keyword	
RTCMK	Changes an RSA key from encipherment with the old master key to encipherment with the current master key.

key_identifier_length
The *key_identifier_length* parameter is a pointer to an integer containing the length in bytes of the field that contains the key token or key label. On

output, the length is the length of token returned in the updated *key_identifier* field a key token (not a key label) was specified. The maximum size length is 2500 bytes.

Key_Identifier
The *key_identifier* parameter is a pointer to a string variable containing an internal key token or a key label of an internal key token record in key storage. The key within the token is securely re-enciphered under the current master key.

Required Commands

When you specify the re-encipher option, the PKA_Key_Token_Change verb requires the Token Change command (offset X'0102') to be enabled in the hardware.

PKA_Public_Key_Extract (CSNDPKX)

Platform/ Product	OS/2	AIX	NT
IBM-4758	X	X	X

The PKA_Public_Key_Extract verb is used to extract a public key from a public-private key-pair. The public key is returned in a PKA public key token.

Both the public key and the related private key must be present in the source key token. The source private key may be in the clear or may be enciphered.

Restrictions

None

Format

The entry point name and the parameters for this verb are shown in the following table:

CSNDPKX			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Inp/Outp	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
source_key_identifier_length	Input	Integer	
source_key_identifier	Input	String	source_key_identifier_length bytes
target_key_token_length	In/out	Integer	
target_key_token	Output	String	target_key_token_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer that contains the number of elements in the rule array. The value of the *rule_array_count* must be zero (no rule array is currently used in this verb).

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The *rule_array* parameter is not presently used by this verb, but must be specified as a parameter.

source_key_identifier_length
The *source_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field that contains the source key identifier. The maximum size that should be specified is 2500 bytes.

source_key_identifier
The *source_key_identifier* parameter is a pointer to a string variable containing either a key label identifying a key storage record or is a PKA96 key token.

target_key_token
The *target_key_identifier* parameter is a pointer to a string variable to receive the PKA96 public key token.

Required Commands

None

PKA_Public_Key_Hash_Register (CSNDPKH)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Public_Key_Hash_Register verb is used to register a hash value for a public key in anticipation of verifying the public key offered in a subsequent use of the PKA_Public_Key_Register verb.

Restrictions

None

Format

CSNDPKH			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
public_key_name	String	64 bytes	
hash_data_length	Input	Integer	
hash_data	Input	String	hash_data_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Hash Type (One keyword required)	
SHA-1	The hash algorithm used to create the hash value.
Special Usage (Optional)	
CLONE	Indicates that the public key associated with this hash value can be employed in a CCA node cloning process provided that this usage is confirmed when the public key is registered.

public_key_name
The *public_key_name* parameter is a pointer to a string variable containing the name under which the registered key will be accessed.

hash_data_length
The *hash_data_length* parameter is a pointer to an integer variable containing the length (in bytes) of the hash data.

hash_data
The hash_data parameter is a pointer to a string variable valued to the SHA-1 hash of a public key certificate that will be offered with the use of the PKA_Public_Key_Register verb. The format of the public key certificate is defined in "RSA Public-key Certificate Section" on page B-10.

Required Commands

The PKA_Public_Key_Hash_Register verb requires the:

- Register PKA Public Key Hash, with Cloning, command (offset X'0201') when using the **CLONE** keyword

Otherwise,

- Register PKA Public Key Hash command (offset X'0200')

to be enabled in the hardware.

PKA_Public_Key_Register (CSNDPKR)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Public_Key_Register verb is used to register a public key in the cryptographic engine. Keywords in the rule array designate the subsequent permissible uses of the registered public key.

The public key offered for registration must be contained in a token that contains a certificate section. The public key value contained in the certificate will be the key that is registered. A pre-registered hash value over the certificate section, exclusive of the certificate signature bits, is used to independently validate the offered key; see the PKA_Public_Key_Hash_Register verb and “RSA Key Token Formats” on page B-5.

Restrictions

None

Format

CSNDPKR

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
public_key_name	Input	String	64 bytes
public_key_certificate_length	Input	Integer	
public_key_certificate	Input	String	certificate length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Special Usage (Optional)	
CLONE	Indicates that the registered public key can be employed in a CCA node cloning process provided that this usage was also asserted when the hash value was registered.

public_key_name
The *public_key_name* parameter is a pointer to a string variable containing the name under which the registered key will be accessed.

public_key_certificate_length
The *public_key_certificate_length* parameter is a pointer to an integer variable containing the length (in bytes) of the public key certificate.

public_key_certificate
The *public_key_certificate* parameter is a pointer to a string variable containing a public key to be registered. The public key must be presented in an RSA public-key certificate section; see "RSA Public-key Certificate Section" on page B-10.

Required Commands

The PKA_Public_Key_Register verb requires the PKA Public Key Register with Cloning command (offset X'0201') to be enabled in the hardware when the **CLONE** rule array keyword is employed, otherwise the PKA Public Key Register command (offset X'0202') must be enabled in the hardware.

Chapter 4. Hashing and Digital Signatures

Verb	Page	Service	Entry Point	Svc Lcn
Digital_Signature_Generate	4-4	This verb generates a digital signature.	CSNDDSG	E
Digital_Signature_Verify	4-7	This verb verifies a digital signature.	CSNDDSV	E
One_Way_Hash	4-10	This verb generates a hash using either the SHA-1 or the MD5 one-way hashing functions.	CSNBOWH	S
Svc Lcn: Service location: E: Cryptographic engine, S: Security API software				

Figure 4-1. Hashing and Digital Signature Services

This chapter discusses the data hashing and the digital signature techniques you can use to determine data integrity. A digital signature may also be used to establish the non-repudiation security property. (Another approach to data integrity based on message authentication codes is discussed in Chapter 6, "Data Confidentiality and Data Integrity.")

- Data integrity and data authentication techniques enable you to determine that a data object (a string of bytes) has not been altered from some known state.
- Non-repudiation permits you to assert that the originator of a digital signature may not later deny having created the digital signature.

This section explains how to determine the integrity of data. Determining data integrity involves determining whether individual values of a string of bytes have been altered. Two techniques are described:

- Hashing
- Digital signatures

Digital signatures uses both hashing and public-key cryptography.

Hashing

Data hashing functions have long been used to determine the integrity of a block of data. The application of a hash function to a data string produces a new quantity called a *hash value*. Many different strings supplied to a given hashing function will produce the same hash value, but because the hash value is a large number, *collisions* (two stings that hash to the same value) are rare.

Hash functions for data integrity applications have a one-way property: given a hash value, it is not likely that a second data string can be found that will hash to the same value as the original. Consequently, if a hash value for a string is known, you can compute the hash value for another string suspected to be the same and compare the two. If both hash values are identical, there is a very high probability that the strings producing them are identical.

The CAA products support the following hash functions:

© Copyright IBM Corp. 1997-98

Secure Hash Algorithm -1 (SHA-1) The SHA-1 is defined in FIPS 180-1 and produces a 20-byte, 160-bit hash value. The algorithm performs best on big-endian, general purpose computers. This algorithm is usually preferred over MD5 if the application designers have a choice of algorithms. SHA-1 is also specified for use with the DSS digital signature standard.

Message Digest -5 (MD5) MD5 is specified in the Internet Engineering Task Force RFC 1321 and produces a 16-byte, 128-bit hash value. This algorithm performs best on little-endian (e.g. Intel), general purpose computers.

There are many different approaches to data integrity verification. In some cases, you can simply make known the hash value for a data string. Anyone wishing to verify the integrity of the data would recompute the hash value and compare the result to the known-to-be-correct hash value.

In other cases, you might want someone to prove to you that they possess a specific data string. In this case, you could randomly generate a challenge string, append the challenge string to the string in question, and hash the result. You would then provide the other party with the challenge string, ask them to perform the same hashing process, and return the hash value to you. This method forces the other party to re-hash the data. When the two hash values are the same you can be confidant that the strings are the same, and the other party actually possesses the data string, and not merely a hash value.

The hashing services described in this chapter allow you to divide a string of data into parts, and compute the hash value for the entire string in a series of calls to the appropriate verb. This can be useful if it is inconvenient or impossible to bring the entire string into memory at one time.

Digital Signatures

You can protect data from undetected modification by including a proof-of-data-integrity value. This proof of data integrity value is called a *digital signature*, and relies on hashing (see “Hashing” above) and public-key cryptography.

When you wish to sign some data you can produce a digital signature by hashing the data and encrypting the results of the hash (the hash value) using your private key. The encrypted hash value is called a digital signature.

Anyone with access to your public key can verify your information as follows:

1. Hash the data using the same hashing algorithm that you used to create the digital signature.
2. Decrypt the digital signature using your public key.
3. Compare the decrypted results to the hash value obtained from hashing the data.

An equal comparison confirms that the data they possess is the same as that which you signed. The Digital_Signature_Generate and the Digital_Signature_Verity verbs described in this chapter perform the hash encrypting and decrypting operations. Their requirements are as follows:

- No one else may have access to your private key, and the use of the key must be controlled so that someone else can not sign data as though they were you.
- The other party must have your public key. They assure themselves that they do have your public key through the use of one-or-more certificates from one-or-more Certification Authorities.

  Note: The verification of public keys also involves the use of digital signatures; however, this subject is outside the scope of this manual.
- The value that is encrypted and decrypted using RSA public-key technology must be the same length in bits as the modulus of the keys. This bit-length is normally 512, 768, 1024, or 2048. Since the hash value is either 128 or 160 bits in length, some process for formatting the hash into a structure for RSA encrypting must be selected.

  Unlike the DES algorithm, the strength of the RSA algorithm is sensitive to the characteristics of the data being encrypted. The digital signature verbs (Verify and Generate) support several different hash-value-formatting approaches. The rule array keywords for the digital signature verbs contain brief descriptions of these formatting approaches.

The receiver of data signed using digital signature techniques can, in some cases, gain *non-repudiation* of the data. The use of digital signatures in legally-binding situations is gaining favor as commerce is increasingly conducted through networked communications. The techniques described in this chapter support the most common methods of digital signing currently in use.

Note: Non-repudiation means that the originator of the digital signature can not later deny having originated the signature, and therefore, the data.

Digital_Signature_Generate (CSNDDSG)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	PKA96

The Digital_Signature_Generate verb is used to generate a digital signature. The hash quantity may be created by the One_Way_Hash or the MDC_Generate verbs.

When an RSA private key is specified (using the PKA key token), the hash formatting method is selected through keywords in the rule_array. The formatted information is then ciphered to obtain the digital signature.

Restrictions

- Not all IBM implementations of this verb may support an optimized form of the RSA private key, however, the Fortress product family implementation of this verb does support an optimized RSA private key (“Chinese Remainder” form).

Format

CSNDDSG			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Inp/Outp	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
PKA_private_key_identifier_length	Input	Integer	
PKA_private_key_identifier	Input	String	PKA_private_key_identifier_length bytes
hash_length	Input	Integer	
hash	Input	String	hash_length bytes
signature_field_length	Inp/Outp	Integer	
signature_bit_length	Output	Integer	
signature_field	Output	String	signature_field_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable. The value of the *rule_array_count* must be zero or one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 4-2. Digital_Signature_Generate Rule_Array Keywords

Keyword	Meaning
RSA-based Digital Signature Hash Formatting Controls	
ISO-9796	Format the hash according to the ISO 9796 standard and generate the digital signature. This is the default.
PKCS-1.0	Calculate the digital signature on the string supplied in the hash variable as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 00
PKCS-1.1	Calculate the digital signature on the string supplied in the hash variable as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 01
ZERO-PAD	Extend the hash by padding on the left with binary zero bits to obtain a bit field with the same length as that of the modulus; cipher the result to obtain the digital signature

Notes:

1. The hash for PKCS-1.0 and PKCS-1.1 should have been created using MD5 or SHA-1 algorithms.
2. The hash for ZERO-PAD can be obtained by any hashing method.

PKA_private_key_identifier_length
The *PKA_private_key_identifier_length* parameter points to an integer variable containing the length (in bytes) of the field containing the public-private key token or key label. The maximum length is 2500 bytes.

PKA_private_key_identifier
The *PKA_private_key_identifier* is a pointer to a string variable containing either a key label identifying a key storage record or an internal public-private PKA96 key token.

hash_length
The *hash_length* parameter is a pointer to an integer variable containing the length (in bytes) of the hash variable.

hash
The *hash* parameter is a pointer to a string variable containing the information to be signed.

Notes:

1. For ISO-9796, the information identified by the *hash* parameter must be less-than-or-equal-to one-half of the number of bytes required to contain the modulus of the RSA key. Although ISO-9796 allows messages of arbitrary bit length up to one-half of the modulus length, this verb requires the input text to be a byte-multiple up to the described maximum length.

2. For PKCS-1.0 or PKCS-1.1, the information identified by the *hash* parameter must be 11 bytes shorter than the number of bytes required to contain the modulus of the RSA key, and should be the ANS.1 BER encoding of the hash value.

   You can create the BER encoding of an MD5 or SHA-1 value by prepending these strings to the 16 or 20-byte hash values, respectively:

   MD5 X'3020300C 06082A86 4886F70D 02050500 0410'
   SHA-1 X'30213009 06052B0E 03021A05 000414'

3. For ZERO-PAD, the information identified by the *hash* parameter must be shorter-than-or-equal-to the number of bytes required to contain the modulus of the RSA key.

signature_field_length
The *signature_field_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field to contain the digital signature. On output, the variable is the actual length of the digital signature. The maximum length is 256 bytes.

signature_bit_length
The *signature_bit_length* is a pointer to an integer variable containing the length (in bits) of the digital signature.

signature_field
The *signature_field* parameter is a pointer to the field where the digital signature is to be stored. Unused bytes at the right of the field are undefined and should be ignored. The digital signature bit field is in the low-order bits of the byte string containing the digital signature.

Required Commands

The Digital_Signature_Generate verb requires the Digital Signature Generate command (offset X'0100') to be enabled in the hardware.

Digital_Signature_Verify (CSNDDSV)

Platform/ Product	**OS/2**	**AIX**	**NT**	Service Group
IBM-4758	**X**	**X**	**X**	PKA96

The Digital_Signature_Verify verb is used to verify a digital signature.

Provide the digital signature, the public key (in a key token), and the hash of the data to be validated. The hash quantity may be created through use of the One_Way_Hash or the MDC_Generate verbs.

The supplied hash information is formatted and compared to the public-key ciphered digital signature. The validation of the digital signature is returned as return code and reason code values.

The hash formatting method is selected through keywords in the rule_array.

Restrictions

Format

CSNDDSV			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Inp/Outp	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
PKA_public_key_identifier_length	Input	Integer	
PKA_public_key_identifier	Input	String	PKA_public_key_identifier_length bytes
hash_length	Input	Integer	
hash	Input	String	hash_length bytes
signature_field_length	Input	Integer	
signature_field	Input	String	signature_field_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

Rule_array_count

The *rule_array_count* is a pointer to an integer that contains the number of elements in the rule array. The value of the *rule_array_count* must be zero or one for this verb.

Rule_array

The *rule_array* is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 4-3. Digital_Signature_Verify Rule_Array Keywords	
Keyword	**Meaning**
RSA-based Digital Signature Hash Formatting Controls	
ISO-9796	Format the hash according to the ISO 9796 standard and generate the digital signature. This is the default.
PKCS-1.0	Calculate the digital signature on the string supplied in the hash variable as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 00
PKCS-1.1	Calculate the digital signature on the string supplied in the hash variable as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 01
ZERO-PAD	Extend the hash by padding on the left with binary zero bits to obtain a bit field with the same length as that of the modulus; cipher the result to obtain the digital signature

Notes:

1. The hash for PKCS-1.0 and PKCS-1.1 should have been created using MD5 or SHA-1 algorithms.
2. The hash for ZERO-PAD can be obtained by any hashing method.

PKA_public_key_identifier_length

The *PKA_public_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field containing the public key token or key label. The maximum length is 2500 bytes.

PKA_public_key_identifier

The *PKA_public_key_identifier* parmaeter is a pointer to a string variable containing either a key label identifying a key storage record, or a PKA96 key token.

hash_length

The *hash_length* is a pointer to an integer variable containing the length (in bytes) of the hash variable.

hash

The *hash* parameter is a pointer to a string variable containing the hash information to be verified.

Notes:

1. For ISO-9796, the information identified by the *hash* parameter must be less-than-or-equal-to one-half of the number of bytes required to contain the modulus of the RSA key. Though ISO-9796 allows messages of arbitrary bit length up to one half of the modulus length, this verb requires the input text to be a byte-multiple up to the correct maximum.

2. For PKCS-1.0 or PKCS-1.1, the information identified by the *hash* parameter must be 11 bytes shorter than the number of bytes required to contain the modulus of the RSA key, and should be the ANS.1 BER encoding of the hash value.

   You can create the BER encoding of an MD5 or SHA-1 value by prepending these strings to the 16 or 20-byte hash values, respectively:

   MD5 X'3020300C 06082A86 4886F70D 02050500 0410'
   SHA-1 X'30213009 06052B0E 03021A05 000414'

3. For ZERO-PAD, the information identified by the *hash* parameter must be shorter-than-or-equal-to the number of bytes required to contain the modulus of the RSA key.

signature_field_length
The *signature_field_length* parameter is a pointer to an integer variable containing the length, (in bytes) of the field containing the digital signature.

signature_field
The *signature_field* parameter is a pointer to a string variable containing the digital signature. The digital signature bit field is in the low-order bits of the byte string containing the digital signature.

Required Commands

The Digital_Signature_Verify verb requires the Digital Signature Verify command (offset X'0101') to be enabled in the hardware.

One_Way_Hash (CSNBOWH)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	PKA96

The One_Way_Hash verb obtains a hash value from a text string using the MD5 or SHA-1 hashing method, as specified in the rule_array.

You can provide all of the data to be hashed in a single call to the verb, or you can provide the data to be hashed using multiple calls. Keywords that you supply in the rule_array to inform the verb of your intention.

Restrictions

If **FIRST** or **MIDDLE** calls are made, the text size must be a multiple of the algorithm block size: 64 bytes for MD5 and SHA-1.

This verb requires that text to be hashed be a multiple of eight bits aligned in bytes. Only data that is a byte multiple can be hashed. (These are not requirements of the standards.)

Format

CSNBOWH			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Inp/Outp	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
text_length	Input	Integer	
text	Input	String	text_length bytes
chaining_vector_length	Input	Integer	
chaining_vector	Inp/Outp	String	chaining_vector_length bytes
hash_length	Input	Integer	
hash	Inp/Outp	String	hash_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable. The value of the *rule_array_count* must be one or two for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 4-4. One_Way_Hash Rule_Array Keywords

Keyword	Meaning
Hash Method (Required)	
MD5	Specifies the use of the MD5 method
SHA-1	Specifies the use of the SHA-1 method.
Chaining Control (Optional)	
FIRST	Specifies the first in a series of calls to compute the hash; intermediate results are stored in the hash variable.
MIDDLE	Specifies this is not the first nor the last in a series of calls to compute the hash; intermediate results are stored in the hash variable.
LAST	Specifies the last in a series of calls to compute the hash; intermediate results are retrieved from the hash variable.
ONLY	Specifies the only call made to compute the hash. This is the default.

text_length
The *text_length* parameter is a pointer to an integer variable containing the length (in bytes) of the text field on which the hash is computed.

Note: If **FIRST** or **MIDDLE** calls are made, the text size must be a multiple of the algorithm block size (64 bytes for MD5 or SHA-1).

text
The *text* parameter is a pointer to a string variable containing the data to be hashed.

chaining_vector_length
The *chaining_vector_length* parameter is a pointer to an integer variable containing the length (in bytes) of the *chaining_vector* field. The *chaining_vector* field must be 128 bytes in length.

chaining_vector
The *chaining_vector* parameter is a pointer to a string variable used by the verb as a work area. Application programs must not alter the contents of this field between related **FIRST**, **MIDDLE**, and **LAST** calls.

hash_length
The *hash_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field where the hash is to be returned. This length must be at least 16 bytes for MD5, and at least 20 bytes for SHA-1. The maximum length is 128 bytes.

hash
The *hash* parameter is a pointer to a string variable that receives the hash value. With use of the **FIRST** or **MIDDLE** keywords, the hash variable receives intermediate results.

Required Commands

None.

Chapter 5. Basic CCA DES Key Management

Figure 5-1. Basic CCA DES Key Management Verbs

Verb	Page	Service	Entry Point	Svc Lcn
Clear_Key_Import	5-16	Enciphers a clear key under the master key, and updates or creates an internal key token for a DATA key.	CSNBCKI	E
Data_Key_Export	5-17	Exports a DES data key and creates an external key token that contains a null control vector.	CSNBDKX	E
Data_Key_Import	5-18	Imports a DES data key and creates an internal key token for the key.	CSNBDKM	E
Diversified_Key_Generate	5-20	Generates a DES key based on supplied information and a key-generating key. The verb often finds us in generating keys for use with smart card transactions.	CSNBDKG	E
Key_Export	5-23	Exports a DES key and creates an external key token.	CSNBKEX	E
Key_Generate	5-25	Generates a random DES key or DES key pair, enciphers the keys, and updates or creates internal or external key tokens.	CSNBKGN	E
Key_Import	5-31	Imports a DES key or a key token, and updates an internal key token or creates an internal key token.	CSNBKIM	E
Key_Part_Import	5-33	Combines clear key parts, enciphers the key, and updates an internal key token.	CSNBKPI	E
Key_Test	5-35	Generates or verifies a verification pattern for keys and key parts.	CSNBKYT	E
Key_Token_Build	5-38	Creates a DES key token from supplied information.	CSNBKTB	S
Key_Token_Change	5-41	Re-encipher a DES key from the old master key to the current master key.	CSNBKTC	E
Key_Translate	5-43	Changes the encipherment of a key from one key-encrypting key to another key-encrypting key.	CSNBKTR	E
Random_Number_Generate	5-45	Generates a random number.	CSNBRNG	E
PKA_Symmetric_Key_Export	5-47	Exports a symmetric key under an RSA public key.	CSNDSYX	E
PKA_Symmetric_Key_Generate	5-49	Generates a new DES key and returns one copy multiply enciphered under the master key and enciphers another copy under an RSA public key.	CSNDSYG	E
PKA_Symmetric_Key_Import	5-52	Imports a symmetric key under an RSA private key.	CSNDSYI	E
Svc Lcn: Service location: E=Cryptographic Engine, S=Security API software				

This chapter describes verbs to perform basic CCA DES key management functions.

The material is presented under these topics:

- Understanding CCA DES Key Management
- Control vectors, key types, and key usage restrictions
- Key tokens, key labels, and key identifiers
- Using the key processing and key storage verbs
- Security precautions
- Basic DES key management verbs in alphabetical order by verb name.

© Copyright IBM Corp. 1997-98

Understanding CCA DES Key Management

The DES algorithm operates on 64 data bits at a time (8 bytes of 8-bit-per-byte data). The results produced by the algorithm are controlled by the value of a *key* that you supply. Each byte of the key contains 7 bits of key information plus a parity bit (the low-order bit in the byte). The parity bit is set so that there are an odd number of one-bits for each key byte. The parity bits do not participate in the DES algorithm.

The DES algorithm is not secret. However, by using a secret key, the algorithm can produce ciphertext that is impossible (for all practical purposes) to decrypt without knowing the secret key. The requirement to keep a key secret, and to have the key available at specific place(s) and time(s), produces a set of activities known collectively as *key management*.

Because the secrecy and reliability of DES-based cryptography is strongly related to the secrecy, control, and use of DES keys, the following aspects of key management are important:

- Securing a cryptographic facility or process. The hardware provides a secure, tamper-resistant environment for performing cryptographic operations and for storing cryptographic keys in the clear. The hardware provides cryptographic functions as a set of commands that are selectively enabled under different profiles. To activate a profile and enable different hardware capabilities, users (programs or persons) must supply identification and a password for verification. Using these hardware capabilities, you can control the use of sensitive key management capabilities.
- Separating key types to restrict the use of each key. A user or a process should be restricted to performing only the processes that are required to accomplish a specific task; therefore, a key should be limited to a set of functions in which it can be used. The cryptographic subsystem uses a system of *control vectors*[1] to separate the cryptographic keys into a set of key types and restrict the use of a key. The subsystem enforces the use of a particular key type in each part of a cryptographic command. To control the use of a key, the control vector is combined with the key that is used to encipher the control vector's associated key. For example, a key that is designated a key-encrypting key can not be employed in the decipher verb so that it can not be used to decrypt keys as though they were data.
- Securely installing and verifying keys. Capabilities are provided for installing keys, either in whole or in parts, and to determine the integrity of the key or the key part to ensure the accurate and secure entry of key information. The hardware commands and profiles allow you to enforce a split-knowledge, dual-control security policy in the installation of keys from clear information.
- Generating keys. The system can generate random clear and enciphered keys. The key generation service creates an extensive set of key types for use in both CCA subsystems and other DES-based systems. Keys can be generated for local use and for distribution to remote nodes.

[1] A control vector is a logical extension of a key variant, which is a method of key separation that some other cryptographic systems use.

- Securely distributing keys manually and electronically. The system provides for unidirectional key distribution channels and a key translation service.

Your application program(s) should provide procedures to perform the following key management activities:

- Generating and periodically replacing keys. A key should be used for a very limited period of time. This can minimize the possibility of an adversary determining the value of a key.
- Archiving keys.
- Destroying keys and media used to distribute keys.
- Auditing the key generation, distribution, installation, archiving, and destruction processes.
- Reacting to unusual occurrences in the key management process.
- Creating management controls for key management.

Before a key is removed from a CCA cryptographic facility for storage in key storage or in application data storage, the key is multiply-enciphered under a master key or another key-encrypting key. The master key is a triple-length DES key composed of three 56-bit DES keys. The key-encrypting keys are double-length DES keys composed of two halves, each half being a 56-bit DES key. While each part of a master key (each 56-bit component) is required to be unique from the other parts, the halves of a key-encrypting key can be the same value. In the latter case, the key-encrypting key operates as though it was a single-length, 56-bit, DES key.

A key that is multiply-enciphered under the master key is an *operational key* (OP). The key is operational because a cryptographic facility can use the master key to multiply-decipher it to obtain original key value. A key that is multiply-enciphered under a key-encrypting key other than the master key is called an *external key*. Two types of external keys are used at a cryptographic node:

- An importable key (IM) is enciphered under an operational key-encrypting key (KEK) whose control vector provides key importing authority.
- An exportable key (EX) is enciphered under an operational key-encrypting key whose control vector provides key exporting authority.

Control Vectors

The CCA cryptographic commands form a complete, consistent, secure command set that performs within tamper-resistant hardware. The cryptographic commands use a set of distinct key types that provide a secure cryptographic system that blocks many attacks that can be directed against it.

The products use a control vector to separate keys into distinct key types and to further restrict the use of a key. A control vector is a non-secret value that is contained in the key token for the key that is cryptographically associated with the key.

A control vector is cryptographically associated with a key by being exclusive-ORed with a master key or another key-encrypting key to form a key

that is used to multiply-encipher or multiply-decipher the key being associated with the control vector. This permanently binds the type and use of the key to the key and ensures the original control vector can not be changed. If the control vector used to decipher a key is different from the control vector that was used to encipher the same key, the correct clear key cannot be recovered. The key-encipherment process is described in detail at "CCA Key Encryption and Decryption Process" on page C-8.

After a key is multiply-enciphered, the originator of the key can ensure that the intended use of the key is preserved by giving the key-encrypting key only to a system that implements the CCA control vector design and that is managed by an audited organization.

Key-encrypting keys in CCA are double-length keys. A double-length DES key consists of two (single-length) 56-bit DES keys that are used together as one key. The first half (left half) of a double-length key, and a single length key are multiply-enciphered using the exclusive-OR of the encrypting key and the control vector. The second half of a double length key is multiply enciphered using the exclusive-OR of the encrypting key and a modification of the control vector; the modification consists of the reversal of control vector bits 41 and 42.

Appendix C, "CCA Control Vector Definitions and Key Encryption" provides detailed information about the construction of a control vector value.

Checking a Control Vector Before Processing a Cryptographic Command

Before a cryptographic facility processes a command that uses a multiply-enciphered key, the facility's logic checks the control vector associated with the key. The control vector must indicate a valid key type for the requested command and any control vector restriction bits must be set appropriately for the command. If the command permits use of the control vector, the cryptographic facility multiply-deciphers the key and uses the key to process the command. (Alteration of the control vector value to permit use of the key in the command would result in recovery of a different, unpredictable key value.)

Figure 5-2 on page 5-5 shows the flow of cryptographic command processing in a cryptographic facility.

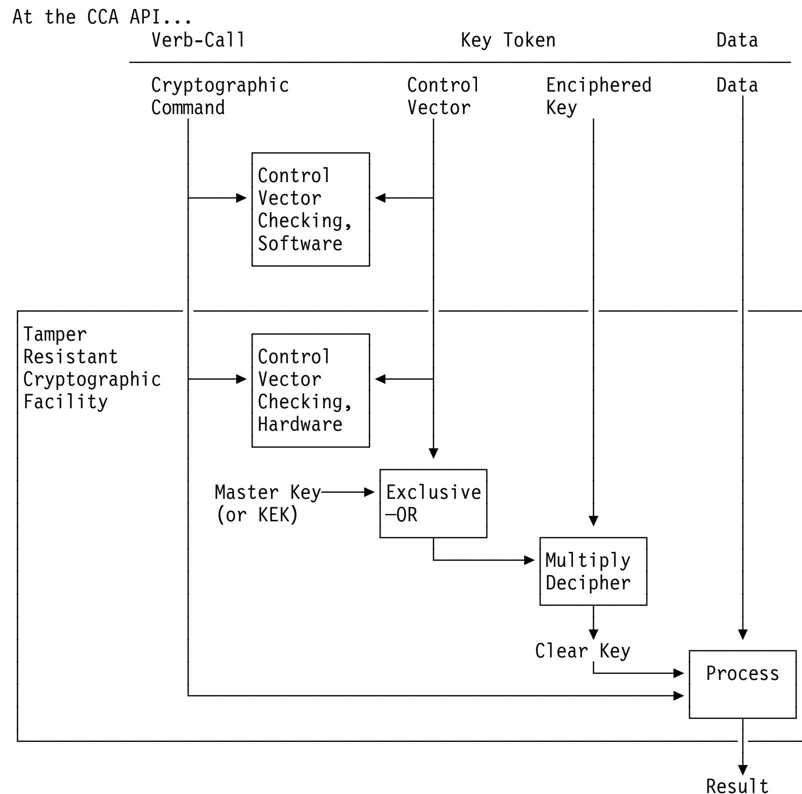

Figure 5-2. Flow of Cryptographic Command Processing in a Cryptographic Facility

Key Types

The CCA implementation in this product defines generic DES key types as shown in Figure 5-3 on page 5-6. The key type in a control vector determines the use of the key, which verbs can use the key, and whether the cryptographic facility processes a key as a symmetric or "asymmetric" DES key. By differentiating keys with a control vector, a given key value can be multiply-enciphered with different control vectors so as to impart different capabilities to copies of the key. This technique creates DES keys having an asymmetric property.

- Symmetric DES keys. A symmetric DES key can be used in two related processes. The cryptographic facility can interpret the following key types as symmetric:
  - DATA. A key with this key type can be used to both encipher and decipher data.
  - MAC. A key with this key type can be used to create a MAC and to verify a trial MAC.
- Asymmetric DES keys. An asymmetric DES key is a key in a key pair in which the keys are used as *opposites.*
  - MAC and MACVER
    Generate and verify a MAC value versus only verify a MAC value.

  The cryptographic facility also interprets key-encrypting keys with the following key types as asymmetric keys that can be used to create one-way key distribution channels:

– EXPORTER or OKEYXLAT. A key with this key type can encipher a key at a node that sends a key.

– IMPORTER or IKEYXLAT. A key with this key type can decipher a key at a node that receives the key.

EXPORTER keys are paired with an IMPORTER or an IKEYXLAT key. IMPORTER keys are paired with an EXPORTER or an OKEYXLAT key. These key types permit the establishment of a uni-directional key distribution channel which is important both to preserve the asymmetric capabilities possible with CCA systems and to further secure a key distribution system from unintended key distribution possibilities.

For information about generating key pairs, see “Generating Keys” on page 5-12.

Depending on the key type, a key can be a single or double-length key. A double-length key that has different values in its left and right halves greatly increases the difficulty for an adversary to obtain the clear value of the enciphered key. A double-length key that has the same values in its left and right halves produces the same results as a single-length key and has the strength of a single-length key.

Figure 5-3. Generic Key Types and Verb Usage

Generic Key Type	Usable with Verbs
MAC Class (Data Operation Key) These keys are used to generate and verify a message authentication code (MAC). They are single-length keys. In operational form and in external form, these keys are associated with a control vector.	
MAC	MAC_Generate, MAC_Verify
MACVER	MAC_Verify
DATA Class (Data Operation Keys) These keys are used to cipher text and to produce and verify message authentication codes. They are single-length keys. In operational form, these keys are always associated with a control vector. In external form, the DATA key-type keys are not usually associated with a control vector.	
DATA	Encipher, Decipher, MAC_Generate, MAC_Verify
Key-Encrypting Key Class These keys are used to cipher other keys. They are double-length keys. In operational form and in external form, these key-encrypting keys are associated with a control vector.	
EXPORTER	Data_Key_Export, Key_Export, Key_Generate, Key_Translate
IMPORTER	Data_Key_Import, Key_Import, Key_Generate, Key_Translate
IKEYXLAT, OKEYXLAT	Key_Translate

Some verbs can create a default control vector for a generic key type. For information about the values for these control vectors, see Appendix C, “CCA Control Vector Definitions and Key Encryption.”

Key Usage Restrictions

In addition to a key type, a control vector contains key-usage values that further restrict the use of a key. The generic key types define a default set of key-usage restrictions in a control vector. These restrictions can be varied by using key-usage keywords when constructing control vector values using the Key_Token_Build verb or by setting bits in the control vector.

Figure 5-4 shows the key type and key-usage keywords that can be combined in the Key_Token_Build verb to create a control vector. The left column lists the generic key types. To the right of the key type are the key-usage keywords that further define a control vector. Default control-vector attributes are noted. Figure 5-5 describes the control vector usage keywords.

For information about the control vector bits, see Appendix C, "CCA Control Vector Definitions and Key Encryption."

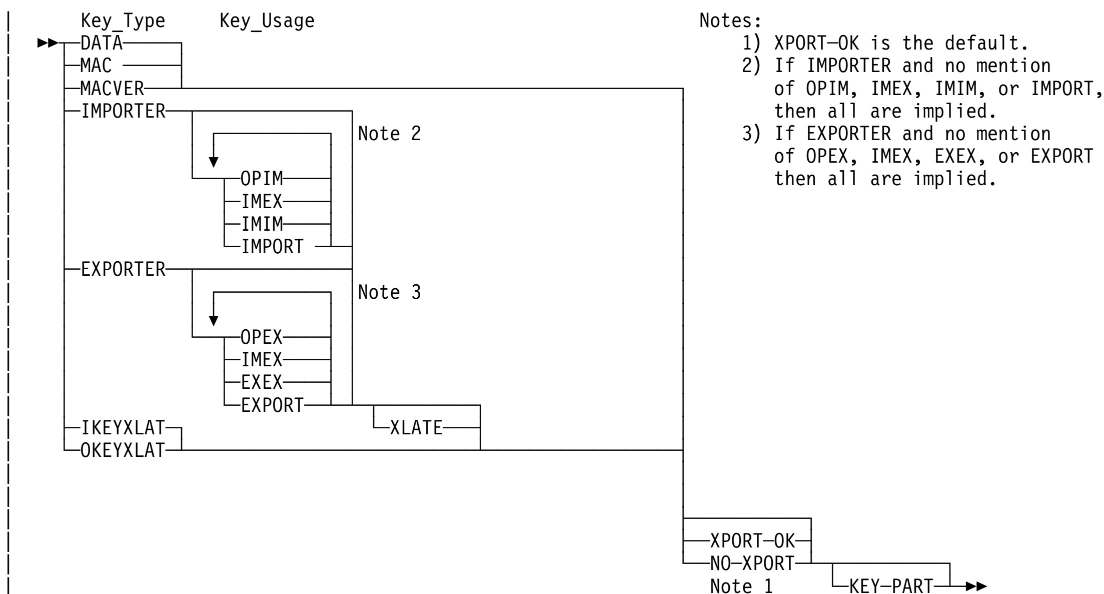

Figure 5-4. Key_Token_Build Keyword Combinations

Figure 5-5 (Page 1 of 2). Control Vector Key-Usage Keywords

Key-Usage Keyword	Meaning
EXPORTER and IMPORTER Key-Encrypting Keys	
OPIM	IMPORTER keys that have a control vector with this attribute can be used in the Key_Generate verb when the key form is OPIM.
IMEX	IMPORTER and EXPORTER keys that have a control vector with this attribute can be used in the Key_Generate verb when the key form is IMEX.
IMIM	IMPORTER keys that have a control vector with this attribute can be used in the Key_Generate verb when the key form is IMIM.
IMPORT	Key-encrypting keys that have a control vector with this attribute can be used to import a key in the Key_Import verb

Figure 5-5 (Page 2 of 2). Control Vector Key-Usage Keywords	
Key-Usage Keyword	**Meaning**
OPEX	EXPORTER keys that have a control vector with this attribute can be used in the Key_Generate verb when the key form is OPEX.
EXEX	EXPORTER keys that have a control vector with this attribute can be used in the Key_Generate verb when the key form is EXEX.
EXPORT	Key-encrypting keys that have a control vector with this attribute can be used to export a key in the Key_Export verb
XLATE	Importer and Exporter key-encrypting keys that have a control vector with this attribute can be used in the Key_Translate verb
Miscellaneous Attributes	
XPORT-OK	Permits the key to be exported by Key_Export or Data_Key_Export.
NO-XPORT	Prohibits the key from being exported by Key_Export or Data_Key_Export.
KEY-PART	Specifies the control vector is for a key part.

Key Tokens, Key Labels, and Key Identifiers

In CCA, a cryptographic key is generally contained within a data structure called a *key token*. The key token can contain the key, a control vector, and other information pertinent to the key. Key tokens can be *null, internal* or *external*. Internal key tokens can be stored in *key storage* and are accessed using a *key label*. The CCA API often permits an application to provide either a key token or a key label, in which case the parameter description is designated as a *key identifier*. Key tokens, labels, and identifiers are discussed in the following sections.

Key Tokens

The security API operates with a *key token* rather than operating simply with a key. A key token is a 64-byte data structure that includes the key and other information frequently needed when the key is needed.

Figure 5-6 shows the general format of a key token. For more information, see Appendix B, "Data Structures."

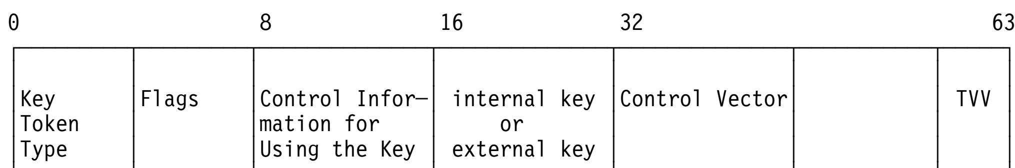

Figure 5-6. Key_Token Contents. In this figure, TVV means token-validation value. See "External Key Token" on page 5-10 and "Internal Key Token" on page 5-10 for information on how the internal and external keys are generated.

A key token contains the following information:

- The key value (multiply enciphered under a key formed by either the master key or a key-encrypting key that is exclusive-ORed with the control vector).
- The control vector for the key. A control vector provides information about the permitted uses of the key.
- Miscellaneous control information (token type, token version layout, and other information).
- A token-validation value (TVV), which is a checksum that is used to validate a token.

You can use the Key_Token_Build verb to assemble a key token. You can also use application code to assemble or disassemble a key token. You should keep in mind, however, the contents and format of key tokens are version and implementation-sensitive. This key-token format is described in Appendix B, "Data Structures" on page B-1.

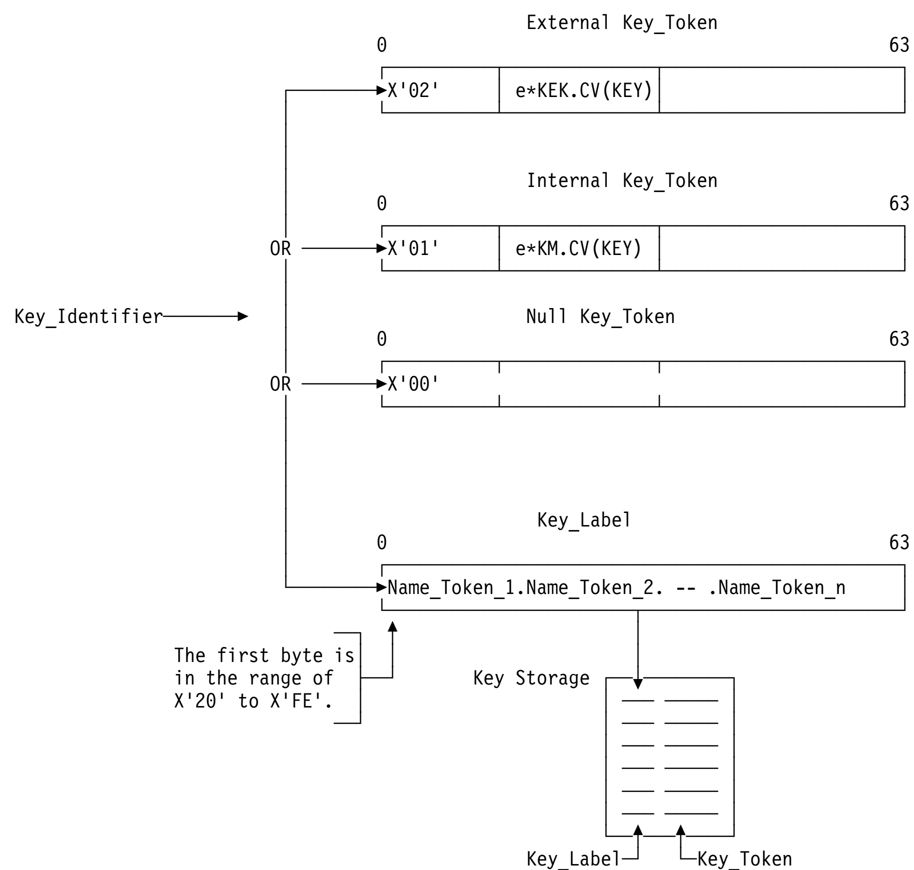

Figure 5-7. Key Identifier, Key Tokens, and Key Labels

The cryptographic system uses external, internal, and null key tokens, as shown in Figure 5-7 on page 5-9.

External Key Token: An external key token contains an external key that is multiply enciphered under a key formed by the exclusive-OR of a key-encrypting key and the control vector that was assigned when the key token was created or updated.

An external key token is specified in a verb call, using a *key_token* parameter. An external key token resides in application data storage. An application program obtains an external key token by calling one of the following verbs:

- Data_Key_Export
- Key_Export
- Key_Generate
- Key_Token_Build

Internal Key Token: An internal key token contains an operational key that is multiply enciphered under a key formed by the exclusive-OR of a master key and the control vector that was used when the key token was created or updated.

An internal key token is specified in a cryptographic verb call by using a *key_identifier* parameter. These verbs produce an internal key token:

- Clear_Key_Import
- Data_Key_Import
- Key_Import
- Key_Generate
- Key_Token_Build
- Symmetric_Key_Import
- Key_Record_Read.

Null Key Token: A null key token is a 64-byte string that begins with the value `X'00'`. A null key token can reside in application data storage or in key storage. Some verbs that create a key token with default values do so when you identify a null key token.

Key Labels

A key label serves as an indirect address for a key token record in key storage. The security server uses a key label to access key storage to retrieve or to store the key token. A *key_identifier* parameter can point to either a key-label or a key-token. Key labels are discussed further at "Key Label Content" on page 7-2.

Key Identifiers

When a verb parameter is described as some form of a *key_identifier*, you can present either a key token or a key label. The key label identifies a key token record in key storage.

Using the Key Processing and Key Storage Verbs

Figure 5-8 on page 5-12 shows key processing and key storage verbs and how they relate to key parts, internal and external key tokens, and key storage. You can create keys in your application programs by using the Key_Generate, Key_Part_Import, Secure_Key_Import, Clear_Key_Import, and Random_Number_Generate verbs.

CCA subsystems do not reveal enciphered keys, and do provide significant control over encrypted keys. Simple key distribution is addressed by the Cryptographic Node Management (CMN) utility's capabilities to read and write encrypted keys from and to key storage and to process key parts with support for dual control of the key parts. Application programs can use the key processing and storage verbs to implement a key distribution system of your design.

The CNM utility, Key_Part_Import verb, Secure_Key_Import verb, and Key_Test verb allow you to install keys securely and verify key installation.

Installing and Verifying Keys

To keep a key secret, it can be installed as a series of key parts. Different individuals can use an application program that loads individual key parts into the cryptographic facility using the Key_Part_Import verb, or the Node Management Utility to enter a key part from a keyboard or diskette.

The key-parts are single-length or double-length, based on the type of key you are accumulating. Key-parts are exclusive-ORed as they are accumulated. Thus, knowledge of a key-part value provides no knowledge about the final key when it is composed of more than one part. An already-entered key-part(s) is stored outside the cryptographic facility enciphered under the master key. When all the key parts are accumulated, the key-part control-vector bit is removed from the key.

A master key key-part is loaded into the new master key register. The key-part replaces the value in the new master key register, or is exclusive-ORed with the existing contents of the register. In a separate command, you can copy the contents of the current master key register to the old master key register and write over the current master key register with the contents of the new master key register.

The commands to load (master) key parts must be individually authorized by appropriate bits being turned on in the active profile register for the Load First (Master) Key Part command or the Load and Combine (Master) Key Part command.

You can use the Key_Test verb to generate a verification pattern and an associated random number. These two values are used together to verify a key or a key part. An application program can use the Key_Test verb to verify the contents of a key-register, an enciphered key, or an enciphered key-part. The utilities also include services to generate and use key and key-part verification patterns.

Though you do not know the value of the key or the key part, you can test a key register, key, or key part to ensure it has a correct value. You can provide to

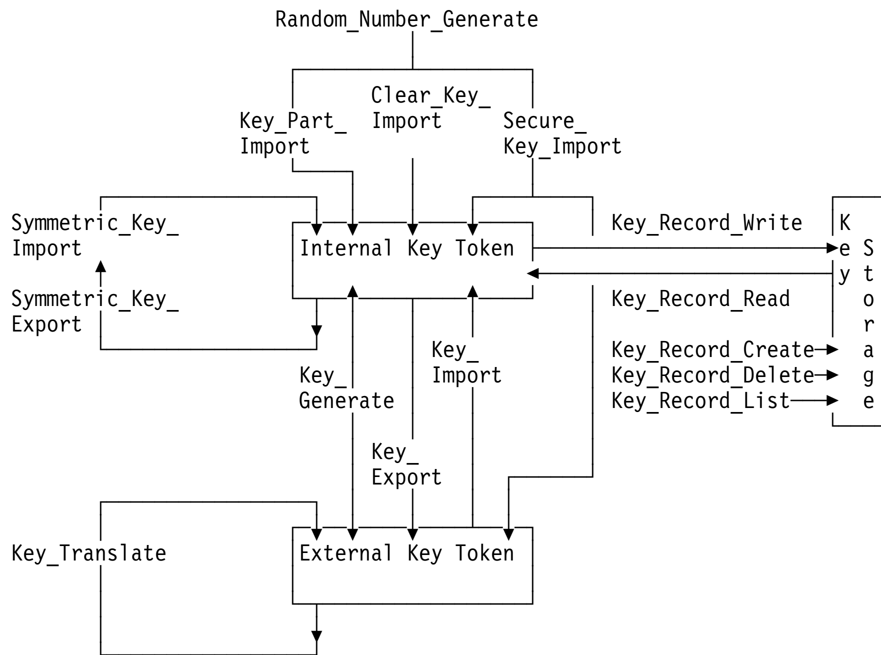

Figure 5-8. Key Processing Verbs

the individual who loads the key parts the verification information for the parts that should already be loaded. If the pattern does not verify, you can instruct the individual or application not to load an additional key part or to set the master key. This procedure can ensure that only valid key parts are used.

In addition to the utilities that are supplied with the hardware, you can use the Key_Part_Import verb in an application program to load keys from individual key parts.

Generating Keys

A CCA cryptographic facility can generate the following keys:

- A clear key. Use a clear key with the Encode, Decode, and Secure_Key_Import verbs. To generate a clear key, use the odd-parity mode of the Random_Number_Generate verb.
- A key part. To generate a key part, use the odd-parity mode of the Random_Number_Generate verb. You can use a key part with the Key_Part_Import verb.
- A multiply-enciphered key or pair of keys. To generate a random, multiply-enciphered key, use the Key_Generate verb. The Key_Generate verb multiply-enciphers a random number using a control vector and either the master key or a key-encrypting key. If you are generating a DES asymmetric key type, the verb will multiply-encipher the random number a second time with the “opposite” key type control vector. The verb restricts the combination of control vectors used for the two encipherments and also places restrictions on the use of master-key versus EXPORTER and IMPORTER encryption key types. This is done to ensure a secure, asymmetric key distribution system.

The Key_Generate verb can also do the following:

- Generate one random number for a single-length key or one or two random numbers for a double-length key.
- Update a key token or create a key token that contains the default control vector values for the key type. If you update a key token, you can use your own control vector to add additional restrictions.

Before generating a key, you should consider how the key will be archived and recovered if unexpected events occur. Before using the Key_Generate verb, you should also consider the following aspects of key processing:

- The use of the key determines the key type and can determine whether you create a key-token with the default control vector or update the key-token with your own control vector that contains additional restrictions.

  If you update a key token, first use the Key_Token_Build verb to create the control vector and the key token, then use the Key_Generate verb to generate the key.

- Where and when the key will be used determines the form of the key, whether the verb generates one key or a key-pair, and whether the verb multiply-enciphers each key for operational, import, or export use. The verb multiply-enciphers each key under a key that is formed by exclusive-ORing the control vector in the new or updated key-token with one of the following keys:

  - The master key. This is the operational (OP) key form.
  - An IMPORTER key-encrypting key. This is the external, importable (IM) key form.
  - An EXPORTER key-encrypting key. This is the external, exportable (EX) key form.

  If a key will be used locally, it should be enciphered in the OP key form or IM key form. An IM key form can be saved on external media and imported when its use is required. Saving a key locally in the IM key form ensures that the key can be used if the master key is changed between the time the key was generated and the time it is used. This allows you to maintain the IMPORTER key-encrypting keys in operational form and to store keys that are not needed immediately on external media.

  If a key will be used remotely (sent to another node), it should be enciphered in the EX key form under a local EXPORTER key. At the other node, the key will be imported under the paired IMPORTER or IKEYXLAT key.

- Use the **SINGLE** keyword for a key that should be single-length. Use the **SINGLE-R** keyword for a double-length key that should perform as a single-length key; this is often required when such a key will be interchanged with a non-CCA system. Use the **DOUBLE** keyword for a double-length key. Since the two halves are random numbers, it is unlikely that the result of the **DOUBLE** keyword will produce two halves with the same 64-bit value.

Exporting and Importing Keys

To operate on data with the same key at two different nodes, you must transport the key securely between the nodes. To do this, a transport-key or key-encrypting key must be installed at both nodes.

A key that is enciphered under a key-encrypting key other than the master key is called an external-key. Deciphering an operational key with the master key and enciphering the key under a key-encrypting key is called a key-export operation and changes an operational key to an external key. The key-export operation is performed in the cryptographic facility so that the clear value of the key to be exported is not revealed.

Deciphering an external key with a key-encrypting key and enciphering the key under the local master key is called a key-import operation, and changes an external key to an operational key.

The control vector for the transport key-encrypting key at the source node must specify the key as an EXPORTER key. The control vector at the target node must specify the transport key-encrypting key as an IMPORTER key. The key to be transported must be multiply-enciphered under an EXPORTER key-encrypting key at the source node and multiply-deciphered under an IMPORTER key-encrypting key at the target node. Figure 5-9 shows both the key-export and key-import operations. Data operation keys, and key-encrypting keys can be transported in this manner. The control vector specifies what kind of keys can be enciphered by a key-encrypting key. For more information, see Appendix C, "CCA Control Vector Definitions and Key Encryption" on page C-1.

Use the Key_Export and the Key_Import verbs to export and import keys with key types that the control vectors associated with the EXPORTER or IMPORTER keys permit. Use can the Data_Key_Export verb and the Data_Key_Import verb to export and import DATA keys; these verbs will not import and export key-encrypting keys.

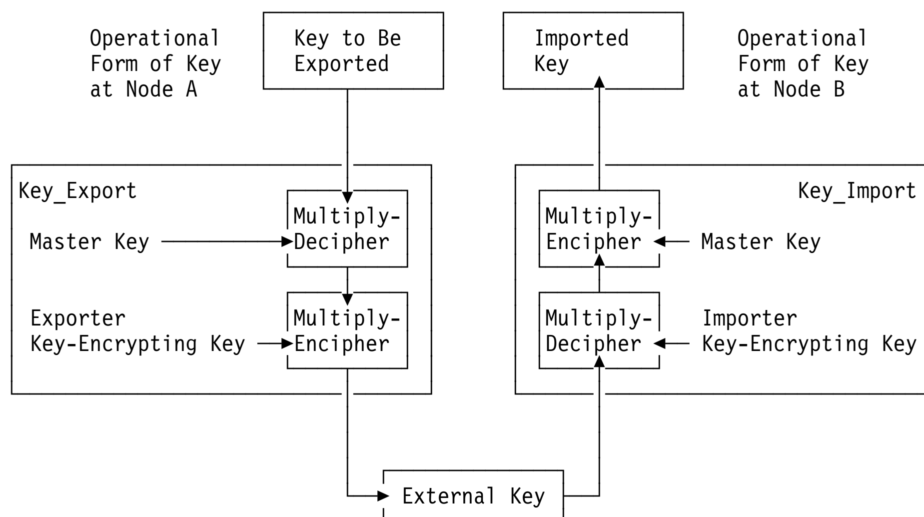

Figure 5-9. Key Exporting and Importing

Storing Keys in Key Storage

Only internal key tokens can be stored in key storage. Data operation keys and key-encrypting keys can be stored in key storage.

The verbs that you use to create, write, read, delete, and list records in key storage, and the format of the key label used to access these records, is described in Chapter 7, "Key Storage Verbs."

Note: To use key storage, the Compute_Verification_Pattern command must first be authorized. This command is used to validate that the master key used to encipher keys within the key storage file had the same value as the master key in the cryptographic facility when the key storage file is opened.

Security Precautions

In order to maintain a secure cryptographic environment, each cryptographic node must be audited in a regular basis. This audit should be aimed at preventing inadvertent and malicious breaches of security. Some of the things that should be audited are listed below:

- The same transport-key should not be used as an EXPORTER key and IMPORTER key on any given cryptographic node. This would destroy the asymmetrical properties of the transport-key.
- Enablement of the Encipher Under Master Key command should be avoided. The secure_key_import verb that employs this command can be used to import any kind of key into the system including a key-encrypting key.
- The Key_Part_Import verb should be used to enter new master keys, key-encryption keys, and data keys into the system. This verb provides for split-knowledge (dual control) of keys by ensuring that no one person knows the true value of a key. Each person enters part of a key and the actual key is not assembled until the last key part is used. Neither the key nor the partial results of the key assembly appear in the clear outside of the secure hardware.

Clear_Key_Import (CSNBCKI)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	**X**	**X**	**X**	Basic

The Clear_Key_Import verb enciphers a clear, single-length DES key under a master key. The resulting key is a DATA key because the service requires that the resulting internal key token have a DATA control vector. You can use this verb to create an internal key token from a null key token, or you can update an existing internal DATA key token with the enciphered value of the clear key. (You can create other types of DES keys from clear key information using the Key_Part_Import verb.)

If the clear-key value does not have odd parity in the low-order bit of each byte, the *reason_code* parameter presents a warning.

Restrictions

None

Format

CSNBCKI			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	*exit_data_length* bytes
clear_key	Input	Integer	8 bytes
target_key_identifier	In/Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

clear_key
The *clear_key* parameter is a pointer to a string variable containing the clear value of the DES key being imported as a DATA key. The key is to be enciphered under the master key. Although not required, the low-order bit in each byte should provide odd parity for the other bits in the byte.

target_key_identifier
The *Target_Key_Identifier* parameter is a pointer to a 64-byte string variable. If the key token in application storage or key storage is null, then a DATA key token containing the encrypted clear key replaces the null token. Otherwise, the pre-existing token must be a DATA key token and the encrypted clear key replaces the existing key value.

Required Commands

The Clear_Key_Import verb requires the Encipher Under Master Key command (command offset X'00C3') to be enabled in the hardware.

Data_Key_Export (CSNBDKX)

Platform/ Product	**OS/2**	**AIX**	**NT**	Service Group
IBM-4758	**X**	**X**	**X**	**Basic**

The Data_Key_Export verb exports an internal DATA key. The verb can export the key from an internal key token in key storage or application storage.

The verb overwrites the 64-byte target key token field with an external DES key token that contains the source key now encrypted by the exporter key-encrypting key. Only a DATA key can be exported. If the source key has a control vector valued to the default DATA control vector, the target key will be enciphered without any control vector (that is, an "all zero" control vector), otherwise the source-key control vector will also be used with the target key.

Restrictions

None

Format

CSNBDKX			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
source_key_identifier	Input	String	64 bytes
exporter_key_identifier	Input	String	64 bytes
target_key_token	Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

source_key_identifier
The *Source_Key_Identifier* parameter is a pointer to a 64 byte string variable containing the internal key token to be exported. Only a DATA key can be exported.

exporter_key_identifier
Exporter_Key_Identifier parameter is a pointer to a 64 byte string variable containing the (EXPORTER) transport key used to encipher the target key.

target_key_token
Target_Key_Token parameter is a pointer to a 64 byte string variable containing the re-encrypted source key token. The target key token will overwrite existing information.

Required Commands

If you export a key from an internal key token in application data storage or in key storage, the Data_Key_Export verb requires the Data Key Export command (command offset X'010A') to be enabled in the hardware.

Data_Key_Import (CSNBDKM)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

The Data_Key_Import verb imports an encrypted source DES DATA key and creates or updates a target internal key token with the master-key enciphered source key. The verb can import the key into an internal key token in application storage or in key storage.

Specify the following:

- An external key token containing the source key to be imported. The external key token must indicate that a control vector is present; however, the control vector is usually valued at zero.

  Alternatively, you can provide the encrypted data key at offset 16 in an otherwise all X'00' key token. The verb will process this token format as a DATA key encrypted by the importer key and a null (all zero) control vector.
- An IMPORTER key-encrypting key under which the source key is deciphered.
- An internal or null key token. The internal key token can be located in application data storage or in key storage.

The verb builds the internal key token by the following:

- Creates a default control vector for a DATA key type in the internal key token, if the control vector in the external key token is zero. If the control vector is not zero, the verb copies the control vector into the internal key token from the external key token.
- Multiply-deciphers the key under the keys formed by the exclusive-OR of the key-encrypting key (identified in the *importer_key_identifier*) and the control vector in the external key token, then multiply-enciphers the key under keys formed by the exclusive-OR of the master key and the control vector in the internal key token. The verb places the key in the internal key token.
- Calculates a token-validation value and stores it in the internal key token.

This verb does not adjust the key parity of the source key.

Restrictions

None

Format

CSNBDKM			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
source_key_token	Input	String	64 bytes
importer_key_identifier	Input	String	64 bytes
importer_key_identifier	In/Output	String	64 bytes
target_key_identifier	In/Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

source_key_token
The *source_key_token* parameter is a pointer to a 64-byte string variable containing the source key to be imported. The source key must be an external key.

importer_key_identifier
The *importer_key_identifier* parameter is a pointer to a 64-byte string variable containing the (IMPORTER) transport key used to decipher the source key.

target_key_identifier
The *target_key_identifier* parameter is a pointer to a 64-byte string variable containing a null key token, an internal key token, or the key label of an internal key token or null key token record in key storage. The key token receives the imported key.

Required Commands

If you import a key into an internal key token: The Data_Key_Import verb requires the Data Key Import command (offset X'0109') to be enabled in the hardware.

Diversified_Key_Generate (CSNBDKG)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

Use the Diversified_Key_Generate verb to generate a key based on a function of a key-generating key and data that you supply. The key-generating key key-type enables you to restrict such keys from being used in other verbs that might reveal the value of a diversified key. The keys generated with this verb are required to have the export control vector bit (bit 17) set off thereby restricting their usage to the local environment.

This verb is especially useful for creating "diversified keys" for operating with finance industry smart cards.

To use the verb, specify the following:

- A 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 key generation processes.
- The data and its length used in the process.
- The operational key used to recover the data or, for processes that employ clear data, a null key token.
- The generated-key key-token with a suitable control vector for receiving the diversified key. The specified process can restrict the type of generated key.

The verb generates the diversified key and updates the generated-key key-token with this value by the following procedure:

- Determines that it can support the process as requested by the rule array keyword
- Recovers the key-generating key and checks the control vector for the key-generating-key class and the specifed usage in this verb
- Determines that the length of the generating key is appropriate to the specified process
- Determines that the control vector in the generated-key key-token is permissible for the specified process including the prohibition of exporting the generated key
- Recovers the data-encrypting key and checks the control vector for the key appropriately for the specified process
- Decrypts the data as can be required by the specified process
- Generates the key appropriate to the specified process
- Returns the generated diversified key, multiply enciphered by the master key.

Restrictions

None

Format

CSNBDKG			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	8 bytes * rule_array_count
generating_key_identifier	Input	String	64 bytes
data_length	Input	Integer	
data	Input	String	data_length bytes
data_decrypting_key_identifier	Input	String	64 bytes
generated_key_identifier	In/Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array.

rule_array
The *rule_array* parameter is a pointer to 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 rule_array keywords are shown below:

Process Rule (one required)	
CLR8-ENC	Specifies that 8 bytes of clear (not encrypted) data shall be multiply encrypted with the generating key and returned as the generated key. The encryption process is like that shown in Figure C-3 on page C-9 for a single-length key with a control vector valued to binary zero. The *data_decrypting_key* parameter must identify a null key token. The control vector in the generated-key token can specify either a DATA, a MAC, or a MACVER key type. The control vector must not permit the key to be exported (bit 17 is zero).

generating_key_identifier
The *generating_key_identifier* parameter is a pointer to a 64-byte string variable containing the key-generating key key-token or key label of a key token. The key must be of the class key-generating key and must have bit 19 set to one.

data_length
The *data_length* parameter is a pointer to an integer variable containing the length of the data variable. The data length can be one of 8, 16, 24, or 32 as specified by the rule array keyword that you select.

data
The *data* parameter is a pointer to a string variable containing the information used in the key generation process. This can be clear or encrypted information based on the process specified in the rule array.

data_decrypting_key_identifier
The *data_decrypting_key_identifier* parameter is a pointer to a 64-byte string variable containing the data decrypting key-token or key label of a key token. The specified process dictates the class of key. If the process does not support encrypted data, point to a null key token.

generated_key_identifier
The *generated_key_identifier* parameter is a pointer to a 64-byte string variable containing the key token or the key label of the target key token. The generated key will be multiply encrypted and returned in the specified token. The control vector in the specified token must be suitable for the specified process.

Required Commands

The Diversified_Key_Generate verb requires the Generate Diversified Key command (offset X'0040') to be enabled in the hardware.

Key_Export (CSNBKEX)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

The Key_Export verb exports a source key into a target external key token. The target key token overwrites existing information. The target key is enciphered by the exporter-key exclusive-ORed with the control vector of the source key.

Specify the following:

- A keyword for the key type. In General, use the **TOKEN** key word. To remain compatible with older systems, you can explicitly name a key type, in which case it must match the key type in the control vector of the source key token.
- A source-key internal key token or the key label of an internal key token record in key storage containing the source key to be exported.
- An EXPORTER key-encrypting key under which the target key is enciphered.
- A 64-byte field to hold the target key token.

The verb builds the external key token by the following:

- Copies the control vector from the internal key token to the external key token, except when the source key has a control vector valued to the default DATA control vector; in this case the target control vector is set to zero.
- Multiply-deciphers the source key under keys formed by the exclusive-OR of the master key and the control vector in the source key token, multiply enciphers the key under keys formed by the exclusive-OR of the exporter key-encrypting key and target-key control vector, and places the result in the target key token.
- Calculates a token-validation value and stores it in the target key token.
- Places the external key token in the 64-byte field identified in the *target_key_token* parameter ignoring any preexisting data.

Restrictions

None

Format

CSNBKEX			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_type	Input	String	8 bytes
source_key_identifier	Input	String	64 Bytes
exporter_key_identifier	Input	String	64 Bytes
target_key_token	Output	String	64 Bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

key_type

key_type parameter is a pointer to an 8-byte string variable containing one of the following keywords to indicate the key type. The **TOKEN** keyword is the most commonly used.

TOKEN	EXPORTER	IMPORTER	MACVER
DATA	IKEYXLAT	MAC	OKEYXLAT

source_key_identifier

source_key_identifier parameter is a pointer to a 64-byte string variable containing the source key token or key label.

exporter_key_identifier

exporter_key_identifier parameter is a pointer to a 64-byte string variable containing the exporter key-encrypting key token or key label.

target_key_token

target_key_token parameter is a pointer to a 64-byte string variable containing the target key token field.

Required Commands

The Key_Export verb requires the Re-Encipher from Master Key command (offset X'0013') to be enabled in the hardware.

Key_Generate (CSNBKGN)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

The Key_Generate verb generates a random DES key and returns one or two enciphered copies of the key, ready to use or distribute.

A control vector associated with each copy of the key defines the type of key and any specific restrictions on the use of the key. Only certain combinations of key types are permitted when you request two copies of a key. Specify the type of key through a key-type keyword, or by providing a key token or tokens with a control vector into which the verb can place the keys. If you specify **TOKEN** as a key-type, the verb uses the pre-existing control vector from the key token. Use of the **TOKEN** keyword allows you to associate other than default control vectors with the generated keys.

Based on the *key_form* variable, the verb encrypts a copy or copies of the generated key under one or two of the following:

- the master key
- an importer key-encrypting key
- an exporter key-encrypting key.

Request two copies of a key when you intend to distribute the key to more than one node, or when you want a copy for immediate local use and the other copy available for later local import.

Specify the key length of the generated key. A DES key can be either single or double length. Certain types of CCA keys must be double length, for example the EXPORTER and IMPORTER key-encrypting keys. In certain cases you need such a key to perform as a single-length key. In these cases, specify **SINGLE-R**, "single replicated." A double-length key with equal halves performs as though the key were a single-length key.

Specify where the generated key copies should be returned, either to your program or key storage. In either case, a null key token can be overwritten by a default key token taken from your specification of key-type. If you provide an existing key token, the verb replaces the key value in the token.

Restrictions

For PIN key support (PINGEN, PINVER, OPINENC, IPINENC), the CCA
| Support Program software must be at level 1.3 or higher. For key-generating
| key support or support of double length MAC or MACVER keys, the CCA
| Support Program software must be at level &pln7beta. or higher.

Format

CSNBKGN			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_form	Input	String	4 bytes
key_length	Input	String	8 bytes
key_type_1	Input	String	8 bytes
key_type_2	Input	String	8 bytes
KEK_key_identifier_1	Input	String	64 bytes
KEK_key_identifier_2	Input	String	64 bytes
generated_key_identifier_1	In/Output	String	64 bytes
generated_key_identifier_2	In/Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_form

The *key_form* parameter is a pointer to a 4-byte string variable that defines whether one or two copies of the key will be generated, and the type of key-encrypting key used to encipher the key.

- When you want a copy of the new key to be immediately useful at the local node, ask for an operational (**OP**) key. An **OP** key is enciphered by the master key.
- When you want a copy of the new key to be imported to the local node at a later time, specify an importable (**IM**) key. An **IM** key is enciphered by an IMPORTER key type at the generating node.
- When you want to distribute the generated key to another node or nodes, specify an exportable (**EX**) key. An **EX** key is enciphered by an EXPORTER key type at the generating node and that is shared with the final destination node.

Specify one of the following key forms:

OP One key for operational use.

IM One key to be imported later to this node.

EX One key for distribution to another node.

OPOP Two copies of the generated key, normally with different control vector values.

OPIM Two copies of the generated key, normally with different control vector values; one for use now, one for later importation.

OPEX Two copies of the generated key, normally with different control vector values; one for local use and the other for use at a remote node.

IMIM Two copies of the generated key, normally with different control vector values; to be imported later to the local node.

IMEX Two copies of the generated key, normally with different control vector values; one to be imported later to the local node and the other for a remote node.

EXEX Two copies of the generated key, sometimes with different control vector values; to be sent to two different remote nodes. No copy of the generated key will be available to the local node.

key_length

The *key_length* parameter is a pointer to an 8-byte string variable containing the length of the new key. Depending on key type, you can specify a single-length key or a double-length key. A double-length key consists of two 8-byte values. Key length must contain one of the following:

| **SINGLE** or **KEYLN8**

For a single-length key.

SINGLE-R For a double-length key with equal-valued halves.

| **DOUBLE** or **KEYLN16**

For a double-length key. The key halves will be different except when the same 56-bit key would be generated twice in succession — a minuscule possibility.

8 spaces To allow the verb to determine key length based on the key's control vector.

key_type_1 and key_type_2

The *key_type_1 and key_type_2* parameters are pointers to 8-byte string variables containing keywords that specify key type for the new keys being generated.. You can also specify key type via the control vector in the pre-existing key token by using the **TOKEN** keyword. Alternatively, you can specify the key type using keywords shown in Figure 5-10 on page 5-28 and Figure 5-11 on page 5-29 This is useful when you want to create default-value key tokens and control vectors.

- Figure 5-10 on page 5-28 lists the keywords allowed when generating a single key copy (*key_forms* OP, IM, and EX). *Key_type_2* must contain a string of eight space characters.
- Figure 5-11 on page 5-29 lists the *key_type_* keyword combinations allowed when requesting two copies of a key value.

kek_key_identifier_1 and kek_key_identifier_2

The *kek_key_identifier_1* and *kek_key_identifier_2* parameters are pointers to 64-byte string variables containing the key token or key label for the key used to encipher the IM-form and EX-form keys. In general, if an OP-form key is requested, the associated KEK identifier should point to a null key token.

generated_key_identifier_1 and generated_key_identifier_2

The *generated_key_identifier_1 and generated_key_identifier_2* parameters are pointers to 64-byte string variables containing the key token or key label of the generated keys. If the parameter identifies an internal or external key token, the verb attempts to use the information in the existing key token, and simply replaces the key value. Using the **TOKEN** keyword in the *key_type_* variables requires that key tokens already exist when the verb is called, so the control vectors in those key tokens can be used. In general, unless you are using the **TOKEN** keyword, you should identify a null key token on input.

Required Commands

Depending on your specification of key form, key type, and use of the **SINGLE-R** key length control, different commands are required to enable operation of the Key_Generate verb.

- If you specify the key-form and key-type combinations shown with an X in Figure 5-10, the Key_Generate verb requires the Generate Key command (offset X'008E') to be enabled in the hardware.
- If you specify the key-form and key-type combinations shown with an X in Figure 5-11 on page 5-29, the Key_Generate verb requires the Generate Key Set command (offset X'008C') to be enabled in the hardware.
- If you specify the key-form and key-type combinations shown with an E in Figure 5-11 on page 5-29, the Key_Generate verb requires the Generate Key Set Extended command (offset X'00D7') to be enabled in the hardware.
- If you specify the **SINGLE-R** key-length keyword, the Key_Generate verb requires the Replicate Key command (offset X'00DB') to be enabled in the hardware.

Related Information

The following sections discuss the *key_type* and *key_length* parameters.

Key Type Specifications

Generated keys are returned multiply-enciphered by a key-encrypting key or by a master key exclusive-ORed with the control vector associated with that copy of the generated key.

Specify the key type of the generated key and its optional copy. If you encode the key type of the key in the control vector of its key token, you can specify **TOKEN** in the *key_form* variable. Or, you can provide a keyword for the key type if you want the default control vector associated with that keyword. One or two keywords are examined based on the *key_form* variable. Figure 5-10 shows the key types for which you can generate one copy of a key.

Figure 5-10. Key_Type and Key_Form Keywords for One Key

Key_Type_1	Key_Form OP	Key_Form IM	Key_Form EX
MAC	X	X	X
DATA	X	X	X
PINGEN	X	X	X
Generate*	X	X	X

Note: The Generate key type (key-generating key) must be requested through the specification of a proper control vector in a key token and the use of the **TOKEN** keyword.

Figure 5-11 on page 5-29 shows the key types for which you can generate two copies of a key. An 'X' indicates a permissible key type for a given key-form. An E indicates that a special (Extended) hardware command is required as those keys require special handling.

If you use the **TOKEN** keyword, the lower portions of the tables indicate key type combinations permitted by the CCA architecture but not supported through keywords.

Figure 5-11. Key_Type and Key_Form Keywords for a Key Pair

Key_Type_1	**Key_Type_2**	**Key_ Form OPOP, OPIM, IMIM**	**Key_ Form OPEX**	**Key_ Form EXEX**	**Key_ Form IMEX**
MAC	MAC	X	X	X	X
MAC	MACVER	X	X	X	X
DATA	DATA	X	X	X	X
EXPORTER	IMPORTER		X	X	X
IMPORTER	EXPORTER		X	X	X
EXPORTER	IKEYXLAT		X	X	X
IKEYXLAT	EXPORTER		X	X	X
IKEYXLAT	OKEYXLAT		X	X	X
IMPORTER	OKEYXLAT		X	X	X
OKEYXLAT	IMPORTER		X	X	X
OKEYXLAT	IKEYXLAT		X	X	X
OPINENC	IPINENC	E	X	X	X
IPINENC	OPINENC	E	X	X	X
PINGEN	PINVER		X	X	X
PINVER	PINGEN		X	X	X
OPINENC	OPINENC	X			
Generate*	Generate*	X	X	X	X

Note: The Generate key type (key-generating key) must be requested through the specification of a proper control vector in a key token and the use of the **TOKEN** keyword.

Key Length Specification

The *key_length* variable contains a keyword which specifies the length of a key, single or double. The key length specified must be consistent with the key length indicated by the control vectors associated with the generated keys. You can specify **SINGLE**, **KEYLN8**, **SINGLE-R**, **KEYLN16**, or **DOUBLE**. The **SINGLE-R** keyword (single replicated) indicates that you want a double-length key where both halves of the key are identical. Such a key performs as though the key were single length.

Figure 5-12 shows the valid key lengths for each key type. An ‘X’ indicates that a key length is permitted for a key type; a ‘D’ indicates the default key length the verb uses when you supply 8 space characters with the *key_length* parameter.

Figure 5-12. Key Lengths by Key Type

Key Type	SINGLE KEYLN8	SINGLE-R	DOUBLE KEYLN16
MAC MACVER	X, D X, D		X X
DATA	X, D		
EXPORTER IMPORTER		X X	X, D X, D
IKEYXLAT OKEYXLAT		X X	X, D X, D
IPINENC OPINENC PINGEN PINVER		X X X X	X, D X, D X, D X, D
Generate*	X	X	X

Note: The Generate key type (key-generating key) must be requested through the specification of a proper control vector in a key token and the use of the **TOKEN** keyword.

Key_Import (CSNBKIM)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

The Key_Import verb imports a source DES key enciphered by the IMPORTER key-encrypting key into a target internal key token. The imported target key is returned enciphered using the master key.

Specify the following:

- A keyword for the key type. In general, use the **TOKEN** key word. For compatibility with older systems, however, you can explicitly name a key type in which case the key type must match the key type encoded in the control vector of the source key token.
- An external key to be imported or an external key token that contains the key to be imported. When you import an enciphered key that is not in an external key token, the key must be located at offset 16 (X'10') of a null-key-token with the first byte set to X'00'.
- The key-encrypting key under which the key is deciphered.
- An internal or null key token or the key label of an internal key token or null key token in key storage.

The verb builds or updates the target key token as follows:

- If the source key is not in an external key token:
  - You must specify an explicit key type (not TOKEN).
  - The default CV for the key type is used when decrypting the source key.
  - The default CV for the key type is used when encrypting the target key.
  - The target key token must either be null or must contain valid, non-conflicting information.

  The key token is returned to the application or key storage with the imported key.
- If the source key is in an external key token:
  - When an explicit key type keyword is used, it must be consistent with the key type encoded in the source-key control vector.
  - The control vector in the source key token is used to decrypt the source key.
  - The control vector in the source key token is used to encrypt the source key under the master key.

  The key token is returned to the application or key storage with the imported key.

The Fortress product family implementations do not adjust key parity.

Restrictions

A SINGLE-R key-encrypting key (a KEK with equal clear-key halves) can not be used to encipher a DOUBLE key (a double-length key with unequal clear-key halves).

Format

CSNBKIM

return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_type	Input	String	8 bytes
source_key_token	Input	String	64 bytes
importer_key_identifier	Input	String	64 bytes
target_key_identifier	In/Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_type
The *key_type* parameter is a pointer to an 8-byte string variable containing a keyword specifying the key type of the key to be imported. In general you should use the **TOKEN** keyword.

TOKEN	EXPORTER	IMPORTER	MACVER
DATA	IKEYXLAT	MAC	OKEYXLAT

source_key_token
The *source_key_token* parameter is a pointer to a 64-byte string variable containing the source key token. Ordinarily the source key token is an external DES key token (the first byte of the key token data structure contains X'02'). However, if the first byte of the token is X'00', then the encrypted source key is taken from the data at offset 16 (X'10') in the source key token structure.

importer_key_identifier
The *importer_key_identifier* parameter is a pointer to a 64-byte string variable containing the key-token or key label for the IMPORTER key-encrypting key.

target_key_identifier
The *target_key_identifier* parameter is a pointer to a 64-byte string variable containing the target key token or key label.

Required Commands

- If you import a key into an internal key token, the key_import verb requires the Re-encipher to Master Key command (offset X'0012') to be enabled in the hardware.

Key_Part_Import (CSNBKPI)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

The Key_Part_Import verb is used to accept parts of a key and store the result as an encrypted partial key or as the final key. Before you use the Key_Part_Import verb, use the Key_Token_Build verb to create the internal key token into which the key will be imported. The control vector in the key token must have the KEY-PART bit set to one.

The first key part is stored in the key token as an encrypted partial key. Subsequent key parts are exclusive-ORed to the partial key. When the last key part is completed, the result is returned as a complete enciphered key with the KEY-PART bit in the control vector reset to zero.

If you use the Key_Part_Import verb to import a key without using key parts, you must call the verb twice. In the first call, specify a key-part of all zeros with odd parity (X'0101...') and specify the **FIRST** keyword in the rule array. In the second call, specify a key part containing the clear key and specify the **LAST** keyword in the rule array.

The returned key is multiply-enciphered by the master key and the control vector in the token pointed to by the *key_identifier* parameter. When you use the **LAST** keyword, the key-part bit is turned off in the key token to reflect that the key is now complete.

Restrictions

None

Format

CSNBKPI

return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_part	Input	String	16 bytes
key_identifier	In/Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable. The value of the *rule_array_count* must be one for this verb.

rule_array

The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 5-13. Key_Part_Import Rule_Array Keywords

Keyword	Meaning
FIRST	Specifies that an initial key part is provided.
MIDDLE	Specifies that an intermediate key part, which is neither the first key part nor the last key part, is provided.
LAST	Specifies that the last key part is provided.

key_part

The *key_part* parameter is a pointer to a 16-byte string variable containing a key part to be entered. The key part may be either 8 or 16-bytes in length; however for 8-byte keys, you must place the key part in the high-order bytes of the 16-byte key part field.

key_identifier

The *key_identifier* parameter is a pointer t o a 64-byte string variable containing the internal DES key token or a key label for a DES key token. The key token must not be null and does supply the control vector for the partial key.

Required Commands

The Key_Part_Import verb requires the following commands to be enabled in the hardware:

- The Load First Keypart command (offset X'001B') with the **FIRST** keyword.
- The Combine Key Parts command (offset X'001C') with the **MIDDLE** and **LAST** keywords.

Key_Test (CSNBKYT)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

The Key_Test verb generates or verifies a verification pattern for keys and key parts. Use this verb to verify that a clear or enciphered key or key part was entered correctly without exposing the value of the key.

Specify in the rule array whether the verb generates or verifies a verification pattern and whether it performs the task on a key or on a key part.

When the verb generates a verification pattern, the verb uses the key or key part to create and cryptographically process a random number; then the verb returns the random number and the verification pattern.

When the verb tests a verification pattern against a key or a key part, you must supply the verification data from a previous procedure call to the Key_Test verb. The verb returns the verification results in the form of a reason code.

The verb returns a return code of 4 and reason code of 1 if verification fails.

You can specify an alternative key test method with the **ENC-ZERO** keyword in the rule array.

For more information about the verification methods used with DES keys, see "Cryptographic Key Verification Techniques" on page D-1.

Restrictions

None

Format

CSNBKYT

return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_identifier	Input	String	64 bytes
random_number	In/Output	String	8 bytes
verification_pattern	In/Output	String	8 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array.

rule_array

The *rule_array* parameter is a pointer to 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 rule_array keywords are shown below:

Key or Key-Part Rule (one required)	
KEY-CLR	Requests processing for a single-length clear key or key part.
KEY-CLRD	Requests processing for a double-length clear key or key part.
KEY-ENC	Requests processing for a single-length enciphered key or key part supplied in a key token.
KEY-ENCD	Requests processing for a double-length enciphered key or key part supplied in a key token.
KEY-KM	Identifies the master key register.
KEY-NKM	Identifies the new master key register.
KEY-OKM	Identifies the old master key register.
Process Rule (one required)	
GENERATE	Generates a verification pattern.
VERIFY	Verifies a verification pattern.
Verification Process Rule (optional)	
ENC-ZERO	Specifies use of the “encrypt zeros” method. Use only with **KEY-CLR**, **KEY-CLRD**, **KEY-ENC**, or **KEY-ENCD**.
Cryptographic Hardware Rule (optional)	
ADAPTER	Specifies the Cryptographic Adapter.
DFLT-CF	Specifies the default cryptographic device or process. In a configuration with more than one cryptographic device or process, the implementation defines which device is the default device or process. This is the default keyword.

key_identifier

The *key_identifier* parameter is a pointer to a 64-byte string variable containing an internal key token, a key label that identifies an internal key token record in key storage, or a clear key.

The key token contains the key or the key part used to generate or verify the verification pattern.

When you specify the **KEY-CLR** keyword, the clear key or key part must be stored in bytes 0 to 7 of the key identifier. When you specify the **KEY-CLRD** keyword, the clear key or key part must be stored in bytes 0 to 15 of the key identifier. When you specify the **KEY-ENC** or the **KEY-ENCD** keyword, the key or key part must be in a key token in the key identifier.

random_number

The *random_number* parameter is a pointer to an 8-byte string variable containing the binary random number the verb uses 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 random number. With the **ENC-ZERO** method, the data in the random_number variable is not used.

verification_pattern

The *verification_pattern* parameter is a pointer to an 8-byte string variable containing the binary verification pattern. When you specify the **GENERATE** keyword, the verb returns the verification pattern. When you specify the **VERIFY** keyword, you must supply the verification pattern.

With the **ENC-ZERO** method, the verification data occupies the high-order four bytes while the low-order four bytes are unspecified (the data is passed between your application and the cryptographic engine but is otherwise unused). See “Cryptographic Key Verification Techniques” on page D-1.

Required Commands

The Key_Test verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the hardware.

Key_Token_Build (CSNBKTB)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

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. You cannot use this verb to change a key or to change the control vector related to a key.

Restrictions

None

Format

CSNBKTB			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_token	Output	String	64 bytes
key_type	Input	String	8 bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_value	Input	String	16 bytes
master_key_verification_pattern	Input	String	4 bytes
reserved	Input	Integer	value ignored
reserved	Input	String	8 bytes, value ignored
control_vector	Input	String	16 bytes
reserved	Input	String	8 bytes
reserved	Input	Integer	
reserved	Input	String	8 bytes
reserved	Input	String	8 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_token
The *key_token* parameter is a pointer to a 64-byte string variable where the verb returns a key token.

Note: You cannot use a key label for a key token record in key storage.

key_type
The *key_type* parameter is a pointer to an eight-byte string variable containing a keyword that defines the key type. The keyword must be 8 bytes in length, uppercase, left-justified, and padded on the right with space characters. Valid key type keywords are shown in the following list:

DATA	IKEYXLAT	MAC	OKEYXLAT
EXPORTER	IMPORTER	MACVER	USE-CV

For information about key types, see Appendix C, "CCA Control Vector Definitions and Key Encryption" on page C-1.

Specify the **USE-CV** keyword to indicate the key type should be obtained from the control vector variable.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array. ~~The value of the *rule_array_count* must be one for this verb.~~

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 5-14. Key_Token_Build Rule_Array Keywords

Keyword	Meaning
Token Type Specify one of the following (required).	
INTERNAL	Specifies an internal key token.
EXTERNAL	Specifies an external key token.
Key Status Specify one of the following (optional)	
KEY	Indicates the key token is to contain a key. The key_value variable contains the key.
NO-KEY	Indicates the key token is not to contain a key. This is the default key status.
Control Vector (CV) Status Specify one of the following (optional).	
CV	Obtain the control vector from the variable identified by the *control_vector* parameter.
NO-CV	This keyword indicates that a control vector is to be supplied based on the key type and control vector related keywords. This is the default. **Note:** If you specify the **USE-CV** keyword in the *key_type* parameter, use the **CV** keyword here.
Control Vector Keywords Specify one of the following (Optional)	
KEY-PART **XLATE**	**Note:** See Appendix C, "CCA Control Vector Definitions and Key Encryption" on page C-1 for a discussion of control vectors and the keywords you can specify to create a control vector value.

key_value
The *key_value* parameter is a pointer to a 16-byte string variable. If you use the **KEY** keyword, the string variable is incorporated into the encrypted-key portion of the key token. Single-length keys must be left-justified in the variable and padded on the right (low-order) with eight-bytes of X'00'.

master_key_verification_pattern
The *master_key_verification_pattern* parameter is a pointer to a four-byte string variable. If you use the **KEY** keyword, the two-byte master key verification pattern is taken from the third and fourth bytes of the source string. The first two bytes must be X'0000'.

control_vector
The *control_vector* parameter is a pointer to a 16-byte string variable. If you use the **CV** keyword, the variable is used as the control vector.

Reserved
Reserved parameters may contain a null address, or may point to an address in application data storage. When an address pointer is not null, you must identify data consistent with the parameter description in the Format section above.

Required Commands

The Key_Token_Build verb has no required hardware commands because it is not a cryptographic verb.

Key_Token_Change (CSNBKTC)

Platform/ Product	OS/2	AIX	NT	Service Group
IBM-4758	X	X	X	Basic

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.

Note: An application system is responsible for keeping all of its keys in a useable form. When the master key is changed, the Fortress product family 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 reenciphered under the current master key for continued use of the key. Use the Key_Token_Change verb to reencipher such a key(s).

Note: 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

Restrictions

None.

Format

CSNBKTC			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_identifier	In/Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array. The value of the *rule_array_count* must be one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 5-15. Key_Token_Change Rule_Array Keywords	
Keyword	**Meaning**
RTCMK	Re-enciphers a DES key to the current master key in an internal key token in application data 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, reason code — 0, 0). If the supplied key is enciphered under the old master key, the key will be updated to encipherment by the current master key and the verb returns a positive response (return code, reason code — 0, 0). Other cases return some form of abnormal response.

Key_Identifier

The *key_identifier* parameter is a pointer to a 64-byte string variable containing the DES internal key token or the key label of an internal key token record in key storage.

Required Commands

If you specify RTCMK keyword, the Key_Token_Change verb requires the Re-Encipher to Current Master Key command (offset X'0090') to be enabled in the hardware.

Key_Translate (CSNBKTR)

Platform/ Product	OS/2	AIX	NT	Service Subset
IBM-4758	X	X	X	Basic

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.

Specify the following key tokens to use this verb:

- The external (input) key token containing the key to be re-enciphered.
- The internal key token containing the IMPORTER or IKEYXLAT key-encrypting key. (The control vector for the IMPORTER key must have the XLATE bit set to 1.)
- The internal key token containing the EXPORTER or OKEYXLAT key-encrypting key. (The control vector for the EXPORTER key must have the XLATE bit set to 1.)
- A 64-byte field for the external (output) key token.

The verb builds the output key token as follows:

- Copies the control vector from the input key token.
- Verifies that the XLATE bit is set to 1 if an IMPORTER or EXPORTER key-encrypting key is used.
- Multiply deciphers the key under a key formed by the exclusive-OR of the key-encrypting key and the control vector in the input key token, multiply enciphers the key under a key formed by the exclusive-OR of the key-encrypting key and the control vector in the output key token; then places the key in the output key token.
- Copies other information from the input key token.
- Calculates a token-validation value and stores it in the output key token.

Restrictions

None.

Format

CSNBKTR			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
input_key_token	In/Output	String	64 bytes
input_KEK_key_identifier	Input	String	64 bytes
output_KEK_key_identifier	Input	String	64 bytes
output_key_token	Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

input_key_token
The *input_key_token* parameter is a pointer to 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
The *input_KEK_key_identifier* parameter is a pointer to a 64-byte string variable containing the internal key token or the key label of an internal key token record in key storage. 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
The *output_KEK_key_identifier* parameter is a pointer to a 64-byte string variable containing the internal key token or the key label of an internal key token record in key storage. 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
The *output_key_token* parameter is a pointer to a 64-byte string variable containing an external key token. The external key token contains the re-enciphered key.

Required Commands

The Key_Translate verb requires the Translate Key command (offset X'001F') to be enabled in the hardware.

Random_Number_Generate (CSNBRNG)

Platform/ Product	**OS/2**	**AIX**	**NT**	Service Subset
IBM-4758	**X**	**X**	**X**	Basic

The Random_Number_Generate verb generates a random number for use as an initialization vector, clear key, or clear key part.

You specify whether the random number is 64-bits or 56-bits with the low-order bit in each byte adjusted for even or odd parity. The verb returns the random number in an eight-byte binary field.

Because the Random_Number_Generate verb uses cryptographic processes, the quality of the output is better than that which higher-level language compilers typically supply.

Restrictions

None

Format

CSNBRNG			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
form	Input	String	8 bytes
random_number	Output	String	8 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

form

The *form* parameter is a pointer to an eight-byte string variable containing a keyword to select the characteristic of the random number. The keyword must be 8 bytes in length, left-justified, and padded on the right with space characters. The keywords are shown in the table below.

Figure 5-16. Key_Token_Build Form Keywords

Keyword	**Meaning**
RANDOM	Requests the generation of a 64-bit random number.
ODD	Requests the generation of a 56-bit, odd parity, random number.
EVEN	Requests the generation of a 56-bit, even parity, random number.

random_number

The *random_number* parameter is a pointer to an eight-byte string variable containing the random number.

Required Commands

The Random_Number_Generate verb requires the Generate Key command (offset X'008E') to be enabled in the hardware.

PKA_Symmetric_Key_Export (CSNDSYX)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Symmetric_Key_Export verb enciphers a symmetric DES or CDMF key using an RSA public key.

Specify the symmetric key to be exported, the exporting RSA public key, and a rule array keyword to define the key-formatting method. The control vector for the DES or CDMF key must permit the key to be exported.

PKCS-1.2 Single length DATA keys (and CDMF keys) are enciphered according to the method described in the RSA DSI PKCS #1 documentation.

Restrictions

| The RSA public key modulus size (key size) is limited by the Function Control Vector to accommodate governmental export and import regulations. The verb enforces this restriction. Generally the key size is limited to 512, 768, or 1024 bits.

| You can not export a key-encrypting key with this verb.

Format

CSNDSYX			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
source_key_identifier_length	Input	Integer	
source_key_identifier	Input	String	source_key_identifier_length bytes
RSA_public_key_token_length	Input	Integer	
RSA_public_key_token	Input	String	RSA_public_key_identifier_length bytes
RSA_enciphered_key_length	In/Output	Integer	
RSA_enciphered_key	Output	String	RSA_enciphered_key_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array. The value of the *rule_array_count* must be one for this verb.

rule_array

The *rule_array* parameter is a pointer to 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 rule_array keywords are shown below:

Keyword	Meaning
Key Formatting Method (one required)	
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02 documentation. Only single-length DES or CDMF DATA keys can be enciphered using this method.

source_key_identifier_length

The *source_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field containing the key label or key token of the key to be exported. The maximum size specified is 2500 bytes.

source_key_identifier

The *source_key_identifier* parameter is a pointer to a string variable containing either an operational key token or the key label of an operational key token to be exported. The associated control vector must permit the key to be exported.

RSA_public_key_token_length

The *RSA_public_key_token_length* parameter is a pointer to an integer variable containing the length (in bytes) of the variable containing the key token or the key label of the RSA public key used to encipher the exported DES key. The maximum size specified is 2500 bytes.

RSA_public_key_token

The *RSA_public_key_token* parameter is a pointer to a string variable containing a PKA96 RSA key token with the RSA public key of the remote node that will import the exported key.

RSA_enciphered_key_length

The *RSA_enciphered_key_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field that to receive the exported RSA-enciphered key. On output, the variable is updated with the actual length of the key field. The maximum size specified is 2500 bytes.

RSA_enciphered_key

The *RSA_enciphered_key* parameter is a pointer to a string variable to receive the exported RSA-enciphered key.

Required Commands

The PKA_Symmetric_Key_Export verb requires these commands to be enabled in the hardware for exporting various key types:

- Symmetric Key Export command (offset X'0105') for DATA keys using the **PKCS-1.2** method.

PKA_Symmetric_Key_Generate (CSNDSYG)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Symmetric_Key_Generate verb generates a random DES key-encrypting key and enciphers the key value. The key value is multiply enciphered under the master key with a control vector. The key value is also enciphered under an RSA public key for distribution to a remote node (that has the associated private key).

The control vector for the local key is taken from an internal (operational) DES key token that must be present on input in the local_enciphered_key_identifier variable or in the key token identified by the key label in that variable.

The rule array defines how the RSA-enciphered key shall be enciphered:

PKA92 Use the key encipherment technique employed in the IBM Transaction Security System (TSS) 4753 and 4755 product PKA92 implementations. See “PKA92 Key Format and Encryption Process” on page D-8. A node-identification (EID) value must have been established prior to use of this verb (use the Cryptographic_Facility_Control verb to set the EID).

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_token variable. Only a key-encrypting keys that conform to the rules for an OPEX case under the Key_Generate verb are permitted.

NL-EPP-5 Use the key encipherment technique defined by certain OEM equipment. See “Encrypting a Key_Encrypting Key in the NL-EPP-5 Format” on page D-10.

Restrictions

This verb only generates key-encrypting keys.

Format

CSNDSYG			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_encrypting_key_identifier	Input	String	64 bytes
RSA_public_key_identifier_length	Input	Integer	
RSA_public_key_identifier	Input	String	RSA_public_key_identifier_length bytes
local_enciphered_key_identifier_length	In/Output	Integer	64
local_enciphered_key_identifier	In/Output	String	
RSA_enciphered_key_token_length	In/Output	Integer	
RSA_enciphered_key_token	In/Output	String	RSA_enciphered_key_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array. The value of the *rule_array_count* must be one for this verb.

rule_array
The *rule_array* parameter is a pointer to 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 rule_array keywords are shown below:

Keyword	Meaning
Key Formatting Method (one required)	
PKA92	Specifies the PKA92 method of key encipherment.
NL-EPP-5	Specifies the NL-EPP-5 process of key encipherment. See "Encrypting a Key_Encrypting Key in the NL-EPP-5 Format" on page D-10.
Key Length (optional)	
SINGLE-R	Specifies that the generated key-encrypting key is to have equal left and right halves and thus perform as a single length key.

key_encrypting_key_identifier
The *key_encrypting_key_identifier* parameter must point to a DES null key token.

RSA_public_key_identifier_length
The *RSA_public_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the variable containing the key token or the key label of the RSA public key used to encipher the exported DES key. The maximum size specified is 2500 bytes.

RSA_public_key_identifier
The *RSA_public_key_identifier* parameter is a pointer to a string variable containing a PKA96 RSA key token with the RSA public key of the remote node that will import the exported key.

local_enciphered_key_identifier_length
The *local_enciphered_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field containing the key label or key token of the local key to be generated. The maximum size permitted is 2500; 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
The *local_enciphered_key_identifier* parameter is a pointer to a string variable containing either a key name or an internal DES 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.

RSA_enciphered_key_length
The *RSA_enciphered_key_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field that to receive the exported RSA-enciphered key. On output, the variable is updated with the actual length of the key field. The maximum size specified is 2500 bytes.

RSA_enciphered_key
The *RSA_enciphered_key* parameter is a pointer to a string variable to receive the generated RSA-enciphered key. An internal (operational) DES key token that must be present on input in the RSA_enciphered_key_token variable.

Required Commands

The PKA_Symmetric_Key_Generate verb requires these command(s) to be enabled in the hardware:

- PKA92 Symmetric Key Generate command (command offset X'010D')
- NL-EPP-5 Symmetric Key Generate command (command offset X'010E')

PKA_Symmetric_Key_Import (CSNDSYI)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Symmetric_Key_Import verb recovers a symmetric (DES or CDMF) key that is enciphered by an RSA public key. The verb deciphers the RSA-enciphered symmetric key to be imported by using an RSA private key, then multiply enciphers the symmetric key DES key using the master key and a control vector.

You specify the operational importing RSA private key, the RSA-enciphered symmetric key to be imported, and a rule array keyword to define the key-formatting method.

Several methods for recovering DES (and CDMF) keys are available. You select a method through the use of a rule array keyword:

PKCS-1.2 Single-length DATA keys (and CDMF keys) are recovered according to the method described in the RSA DSI PKCS #1 documentation. The result is enciphered as a single-length DATA key.

PKA92 Single- and double-length keys and their control vectors are deciphered using the method employed in the Transaction Security System PKA92 implementation. A node-identification (EID) value must have been established prior to use of this verb.

Note: A key-encrypting key RSA-enciphered at this node (EID) cannot be imported at this same node.

Restrictions

The RSA public key modulus size (key size) is limited by the Function Control Vector to accommodate governmental export and import regulations. The verb enforces this restriction. Generally the key size is limited to 512, 768, or 1024 bits.

The EID enciphered with a key-encrypting key can not be the same as the EID of the importing cryptographic engine.

Other IBM implementations of this verb may not support:

- Key types other than a default DATA control vector
- Use of a key label with the target key identifier.

Check the product-specific literature for restrictions.

Format

CSNDSYI			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
RSA_enciphered_key_length	Input	Integer	
RSA_enciphered_key	Input	String	RSA_enciphered_key_length bytes
RSA_private_key_identifier_length	Input	Integer	
RSA_private_key_identifier	Input	String	RSA_private_key_identifier_length bytes
target_key_identifier_length	In/Output	Integer	
target_key_identifier	In/Output	String	target_key_identifier_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array. The value of the *rule_array_count* must be one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
RSA Key Encipherment Method (one required)	
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02 documentation. Only a DATA key can be deciphered using this method.
PKA92	Specifies the PKA92 method of key encipherment. Single- and double-length DES (and single-length CDMF) keys can be imported.

RSA_enciphered_key_length
The *RSA_enciphered_key_length* parameter is a pointer to integer containing the length (in bytes) of the field containing the key being imported. The maximum size specified is 2500 bytes.

RSA_enciphered_key
The *RSA_enciphered_key* parameter is a pointer to a string variable containing the key being imported.

RSA_private_key_identifier_length
The *RSA_private_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field containing the RSA key used to decipher the RSA-enciphered key, or the key label of such a key. The maximum size specified is 2500 bytes.

RSA_private_key_identifier
The *RSA_private_key_identifier* parameter is a pointer to a string variable containing a key label or a PKA96 key token with the internal RSA private key to be used to decipher the RSA-enciphered key.

target_key_identifier_length
The *target_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field containing the target_key_identifier. On output, the variable is updated with the actual length of the key field. The maximum size specified is 2500 bytes.

target_key_identifier
The *target_key_identifier* parameter is a pointer to a string variable containing either a key label, an internal key token, or a null key token. Any identified internal key token must contain a control vector that conforms to the requirements of the key that is imported. For example, if the **PKCS-1.2** keyword is used in the rule array, the key token must contain a default-value, DATA control vector. The imported key is returned in a key token identified through this parameter.

Required Commands

The PKA_Symmetric_Key_Import verb requires these commands to be enabled in the hardware for importing various key types:

- Symmetric Key Import command (offset X'0106') for DATA keys using **PKCS-1.2** method.
- PKA92 Symmetric Key Import command (offset X'0235') when importing DATA, MAC, MACVER, or key-generating keys using the **PKA92** method.
- PKA92 PIN Key Import command (offset X'0236') when importing PINGEN, PINVER, IPINENC, or OPINENC keys using the **PKA92** method.

Chapter 6. Data Confidentiality and Data Integrity

Figure 6-1. Data Confidentiality and Data Integrity Verbs

Verb	Page	Service	Entry Point	Svc Lcn
Decipher	6-4	Deciphers data.	CSNBDEC	E
Encipher	6-7	Enciphers data.	CSNBENC	E
MAC_Generate	6-10	Generates a message authentication code (MAC).	CSNBMGN	E
MAC_Verify	6-13	Verifies a MAC.	CSNBMVR	E
Svc Lcn: Service location: E=Engine, S=Security API software				

This chapter describes the verbs that use the Data Encryption Standard (DES) algorithm to encrypt and decrypt data and to generate and verify a message authentication code (MAC).

Encryption and Message Authentication Codes

This section explains how to use the services described in this chapter to ensure the confidentiality of data through encryption, and to ensure the integrity of data through the use of Message Authentication Codes (MAC).

Note: See Chapter 4, "Hashing and Digital Signatures" on page 4-1 for information about other ways to ensure data integrity.

Ensuring Data Confidentiality

You can use the Encipher verb to convert plaintext to ciphertext, and the Decipher verb to reverse the process to convert ciphertext back to plaintext. These services use the DES data encryption algorithm. DES operates on blocks of 64 bits (8 bytes).

If you know that your data will always be a multiple of 8 bytes, you can request the use of the *cipher block chaining* mode of encryption, designated *CBC*. In this mode of encryption, the enciphered result of encrypting one block of plaintext is exclusive-ORed with the subsequent block of plaintext prior to enciphering the second block. This process is repeated through the processing of your plaintext. The process is reversed in decryption; see "Ciphering Methods" on page D-3.

Note that if some portion of the ciphertext is altered, the CBC decryption of that block and the subsequent block will not recover the original plaintext; other blocks of plaintext will be correctly recovered. CBC encryption is used to disguise patterns in your data that could be seen if each data block was encrypted by itself.

In general, data to be ciphered is not a multiple of 8 bytes. In this case you need to adopt a strategy for the *last block* of data. The Encipher and Decipher verbs also support the ANSI X9.23 mode of encryption. In X9.23 encryption, at least one byte, and up to eight bytes, of data are always added to the end of your plaintext. The last of the added bytes is a binary value equal to the

number of added bytes. In X9.23 decryption, the padding is removed from the decrypted plaintext.

Whenever the first block of plaintext has a predictable value, it is important to modify the first block of data prior to encryption to deny an adversary a known plaintext-ciphertext pair. There are two common approaches:

- Use an *initialization vector*
- Prepend your data with 8 bytes of random data, an *initial text sequence*.

An initialization vector is exclusive-ORed with the first block of plaintext prior to encrypting the result. The initialization vector is exclusive-ORed with the decryption of the first block of ciphertext to correctly recover the original plaintext. You must of course have a means of passing the value of the initialization vector from the encryption process to the decryption process; a common solution to the problem is to pass the initialization vector as an encrypted quantity during key agreement between the encrypting and decrypting processes. You specify the value of an initialization vector when you invoke the Encipher and the Decipher verbs.

If the procedure for agreeing on a key does not readily result in passing of an encrypted quantity that can serve as the initialization vector, then you can add 8 bytes of random data to the start of your plaintext. Of course the decrypting process must remove this initial text sequence as it recovers your plaintext. An initialization vector valued to binary zero is used in this case.

The key used to encrypt or decrypt your data is specified in a key token. The control vector for the key must be of the general class DATA[1].

If an invocation of the Encipher or the Decipher verb should include use of the initialization vector value, use the keyword **INITIAL**. If there is more data that is a logical extension of preceding data, you can use the keyword **CONTINUE**. In this case, the initialization vector value is not used, but the enciphered value of the last block of data from a prior ciphering verb is taken from the *chaining_vector* save area that you must provide with each use of the ciphering verbs. Each portion of your data must be a multiple of eight bytes and you must use the **CBC** encryption mode. You can use **X9.23** keyword with the final invocation of the ciphering verbs if your processes use this method to accommodate data that can be other than a multiple of eight bytes.

Ensuring Data Integrity

CCA offers three classes of services for ensuring data integrity:

- Message authentication code (MAC) techniques based on the DES algorithm
- Hashing techniques
- Digital signature techniques.

---

[1] Uppercase letters are used for DATA to distinguish the meaning from a more general sense in which the term *data* keys means keys used for ciphering and MACing. In this publication, DATA means the control-vector specified class of keys that can participate in Encipher and Decipher verbs. Note that the default value of the DATA control vector also permits DATA keys to participate in MAC_Generate and MAC_Verify operations. This is not true for all implementations of CCA.

For information on using hashing or digital signatures to ensure the integrity of data, see Chapter 4, “Hashing and Digital Signatures.” This chapter describes the MAC verbs.

The MAC_Generate and the MAC_Verify verbs support message authentication code generation and verification consistent with ANSI standard X9.9, ISO DP 8731, Part I, and ANSI X9.19 Optional Procedure 1. These methods together support both single- and double-length keys. For additional information about MAC calculation methods, see “MAC Calculation Method” on page D-7.

You can employ MAC values with four, six, or eight-byte lengths (32, 48, or 64 bits) by using the **MACLEN4**, **MACLEN6**, or **MACLEN8** keywords in the rule array. **MACLEN4** is the default.

When generating or verifying a 32-bit MAC, exchange the MAC in one of these ways:

- Binary, in four bytes (the default method)
- Eight hexadecimal characters, invoked using the **HEX-8** keyword
- Eight hexadecimal characters with a space character between the fourth and fifth hex characters invoked using the **HEX-9** keyword.

For details about MAC services, see the MAC_Generate verb on page 6-10 and the MAC_Verify verb on page 6-13.

MACing Segmented Data

The MAC services described in this chapter allow you to divide a string of data into parts, and generate or verify a MAC in a series of calls to the appropriate verb. This can be useful when it is inconvenient or impossible to bring the entire string into memory. For example, you might wish to MAC the entire contents of a data set tens or hundreds of mega-bytes in length. The length of the data in each procedure-call is restricted only by the operating environment and the particular verb. For restrictions to a verb, see the “Restriction” section of the verb descriptions later in this chapter.

In each procedure-call, a segmenting-control keyword indicates whether the call contains the first, middle, or last unit of segmented data; the *chaining_vector* parameter specifies the work area that the verb uses. (The default segmenting-control keyword **ONLY** specifies that segmenting is not used.)

Decipher (CSNBDEC)

Platform/ Product	OS/2	AIX	NT	Service Subset
IBM-4758	X	X	X	Basic

The Decipher verb uses the Data Encryption Standard (DES) or the Commercial Data Masking Facility (CDMF) algorithm and a cipher key to decipher data (ciphertext). This verb results in data called plaintext.

Performance can be enhanced if you align the start of the plaintext and ciphertext variables on a four-byte boundary.

For information about the ciphering verbs, see "Ensuring Data Confidentiality" on page 6-1.

Restrictions

The maximum text_length is restricted to 32 megabytes.

Format

CSNBDEC

return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Inp/Outp	String	exit_data_length bytes
key_identifier	Input	String	64 bytes
text_length	In/Out	Integer	
ciphertext	Input	String	text_length bytes
initialization_vector	Input	String	8 bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
chaining_vector	In/Out	String	18 bytes
plaintext	Output	String	text_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_identifier
The *key_identifier* parameter is a pointer to a string variable containing a 64-byte internal key token or a key label of an internal key token record in key storage.

text_length
The *text_length* parameter is a pointer to an integer variable containing the length of the ciphertext. If the plaintext returned is a different length because the padding was removed, the verb updates the input value to the length of the plaintext.

ciphertext
The *ciphertext* parameter is a pointer to a string variable containing the text to be deciphered.

initialization_vector
The *initialization_vector* parameter is a pointer to an eight-byte string variable containing the initialization_vector the verb uses with the input data.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. If the rule array does not specify a ciphering method, the default ciphering method is **CBC**.

For an adapter that supports both DES and CDMF, you can choose the encryption process. The rule_array keywords are shown below:

Figure 6-2. Decipher Rule_Array Keywords

Keyword	Meaning
Ciphering Method Selection	
CBC	Specifies cipher-block chaining. The data must be a multiple of eight bytes.
X9.23	Specifies cipher-block chaining with one to eight bytes of padding. This is compatible with the requirements in ANSI Standard X9.23.
ICV Selection	
INITIAL	Specifies use of the initialization-vector from the key token or the initialization-vector to which the *initialization_vector* parameter points. This is the default.
CONTINUE	Specifies use of the initialization-vector to which the *chaining_vector* parameter points. The **CONTINUE** keyword is not valid with with the **X9.23** keyword.
Cipher Algorithm	
DES	Specifies use of the DES ciphering algorithm. If an adapter does not support DES general data-decipherment, the verb is rejected. This is the default on an adapter that supports both DES and CDMF.
CDMF	Specifies use of the CDMF ciphering algorithm.

chaining_vector
The *chaining_vector* parameter is a pointer to an 18-byte string variable containing the segmented data between calls by the security server. The output chaining vector is contained in bytes zero through seven.

Note: The application program must not change the data in this string.

plaintext
The *plaintext* parameter is a pointer to a string variable to contain the plaintext the verb returns. The starting address of plaintext **cannot** begin within ciphertext.

Required Commands

The Decipher verb requires the Decipher command (offset X'000F') to be enabled in the hardware.

Encipher (CSNBENC)

Platform/ Product	OS/2	AIX	NT	Service Subset
IBM-4758	X	X	X	Basic

The Encipher verb uses the DES algorithm and a secret key to encipher data. This verb returns data called ciphertext.

Ciphertext can be as many as eight bytes longer than the plaintext due to padding. Ensure the ciphertext buffer is large enough.

Performance can be enhanced by aligning the start of the plaintext and ciphertext variables on four-byte boundaries.

For general information about the ciphering verbs, see "Ensuring Data Confidentiality" on page 6-1.

Restrictions

The maximum text_length is restricted to 32 megabytes.

Format

CSNBENC			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Inp/Outp	String	exit_data_length bytes
key_identifier	In/Out	String	64 bytes
text_length	In/Out	Integer	
plaintext	Input	String	text_length bytes
initialization_vector	Input	String	8 bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
pad_character	Input	Integer	
chaining_vector	In/Out	String	18 bytes
ciphertext	Output	String	text_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_identifier
The *key_identifier* parameter is a pointer to a 64-byte string variable containing an internal key token or the key label of an internal key token record in key storage.

text_length
The *text_length* parameter is a pointer to an integer variable containing the length of the plaintext and ciphertext. If ciphertext is longer because padding bytes were added, the verb updates the input value to be the length of the ciphertext.

plaintext
The *plaintext* parameter is a pointer to a string variable containing the text to be enciphered.

initialization_vector
The *initialization_vector* parameter is a pointer to an eight-byte string variable containing the initialization_vector the verb uses with the input data.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. If the rule array does not specify a ciphering method, the default method is **CBC**. The rule_array keywords are shown below:

Figure 6-3. Encipher Rule_Array Keywords

Keyword	Meaning
Ciphering Method Selection	
CBC	Specifies cipher-block chaining. The data must be a multiple of eight bytes.
X9.23	Specifies cipher block chaining with one to eight bytes of padding. This is compatible with the requirements in ANSI Standard X9.23.
ICV Selection	
INITIAL	Specifies use of the initialization-vector from the key token or the initialization-vector to which the *initialization_vector* parameter points. This is the default.
CONTINUE	Specifies use of the initialization-vector to which the *chaining_vector* parameter points. The **CONTINUE** keyword is not valid with the **X9.23** keyword.
Cipher Algorithm	
DES	Specifies use of the DES ciphering algorithm. If an adapter does not support DES general data encipherment, the verb is rejected. This is the default on an adapter that supports both DES and CDMF.
CDMF	Specifies use of the CDMF ciphering algorithm.

pad_character
The *pad_character* parameter is a pointer to an integer containing a value used as a padding character. The value must be in the range from 0 to 255. When you use the **X9.23** ciphering method, the security server extends the plaintext with a count byte and padding bytes as required.

chaining_vector
The *chaining_vector* parameter is a pointer to an 18-byte string variable that the security server uses as a work area to carry segmented data between procedure-calls.

Note: The application program must not change the data in this string.

ciphertext

The *ciphertext* parameter is a pointer to a string variable that receives the enciphered text. The ciphertext field might be eight bytes longer than the plaintext because padding. The starting address of ciphertext **cannot** begin within plaintext.

Required Commands

The Encipher verb requires the Encipher command (offset X'000E') to be enabled in the hardware.

MAC_Generate (CSNBMGN)

Platform/ Product	**OS/2**	**AIX**	**NT**	Service Subset
IBM-4758	**X**	**X**	**X**	Basic

The MAC_Generate verb generates a message authentication code (MAC) for a text string supplied by the application program. Both single- and double-length keys are supported.

Performance can be enhanced by aligning the start of the text variable on a four-byte boundary.

For information about using the MAC generation and verification verbs, see "Ensuring Data Integrity" on page 6-2.

Restrictions

Text length must be at least 8 bytes and less than 32 megabytes.

Format

CSNBMGN			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
key_identifier	Input	String	64 bytes
text_length	Input	Integer	
text	Input	String	text_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
chaining_vector	In/Output	String	18 bytes
MAC	Output	String	8 or 9 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_identifier
The *key_identifier* parameter is a pointer to a 64-byte string variable containing an internal key token or the key label of an internal key token record in key storage. Use either MAC or DATA key types. MAC keys can be either single- or double-length.

text_length
The *text_length* parameter is a pointer to an integer containing the length of the text.

text
The *text* parameter is a pointer to a string variable containing the text the hardware uses to calculate the MAC.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array.

| If the rule array count value is zero, the default segmenting-control is **ONLY**
| and the default MAC-length is **MACLEN4**.

rule_array

The *rule_array* parameter is a pointer to 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 rule_array keywords are shown below:

Keyword	Meaning
MAC Ciphering Methods (one, optional)	
X9.9-1	Specifies the ANSI X9.9-1 and X9.19 Basic Procedure. This is the default for a single length key.
X9.19OPT	Specifies the ANSI X9.19 Optional Procedure. This is the default for a double length key.
Segmenting Control (one, optional)	
ONLY	Specifies the application program does not use segmenting. This is the default.
FIRST	Specifies this is the first segment of data from the application program.
MIDDLE	Specifies this is an intermediate segment of data from the application program.
LAST	Specifies this is the last segment of data from the application program.
MAC Length and Presentation (one, optional)	
MACLEN4	Specifies a four-byte MAC. This is the default.
MACLEN6	Specifies a six-byte MAC.
MACLEN8	Specifies an eight-byte MAC.
HEX-8	Specifies a four-byte MAC and presents it as eight hexadecimal characters.
HEX-9	Specifies a four-byte MAC and presents it as two groups of four hexadecimal characters separated by a space character.

chaining_vector

The *chaining_vector* parameter is a pointer to an 18-byte string variable the security server uses as a work area to carry segmented data between procedure calls.

Note: The application program must not change the data in this string.

MAC

The *MAC* parameter is a pointer to a string variable that receives the resulting MAC. The value is left-justified in the field. Allocate a field large enough to receive the resulting MAC value.

Required Commands

The MAC_Generate verb requires the Generate MAC command (offset X'0010') to be enabled in the hardware.

MAC_Verify (CSNBMVR)

Platform/ Product	OS/2	AIX	NT	Service Subset
IBM-4758	X	X	X	Basic

The MAC_Verify verb verifies a message authentication code (MAC) for a text string supplies by the application program. Both single- and double-length keys are supported.

Performance can be enhanced by aligning the start of the text variable on a four-byte boundary.

For information about using the MAC generation and verification verbs, see "Ensuring Data Integrity" on page 6-2.

Restrictions

Text length must be at least eight bytes and less than 32 megabytes.

Format

CSNBMVR			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
key_identifier	Input	String	64 bytes
text_length	Input	Integer	
text	Input	String	text_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
chaining_vector	In/Output	String	18 bytes
MAC	Input	String	9 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_identifier
The *key_identifier* parameter is a pointer to a 64-byte string variable containing an internal key token or the key label of an internal key token record in key storage. Use either MAC, MACVER, or DATA key types. MAC and MACVER keys can be either single- or double-length.

text_length
The *text_length* parameter is a pointer to an integer containing the length of the text the hardware uses to calculate the MAC.

text
The *text* parameter is a pointer to a string variable containing the text the hardware uses to calculate the MAC.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule array.

If the rule array count value is zero, the default segmenting-control is **ONLY** and the default MAC-length is **MACLEN4**.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below: If the rule array does not specifies a MAC-length, the default method is **MACLEN4**.

Keyword	Meaning
MAC Ciphering Methods (Optional)	
X9.9-1	Specifies the ANSI X9.9-1 and X9.19 Basic Procedure. This is the default for a single length key.
X9.19OPT	Specifies the ANSI X9.19 Optional Procedure. This is the default for a double length key.
Segmenting Control (One, optional)	
Segmenting Control (One, optional)	
ONLY	Specifies the application program does not use segmenting. This is the default.
FIRST	Specifies this is the first segment of data from the application program.
MIDDLE	Specifies this is an intermediate segment of data from the application program.
LAST	Specifies this is the last segment of data from the application program.
MAC Length and Presentation (One, optional)	
MACLEN4	Specifies a four-byte MAC. This is the default.
MACLEN6	Specifies a six-byte MAC.
MACLEN8	Specifies an eight-byte MAC.
HEX-8	Specifies a four-byte MAC and presents it as eight hexadecimal characters.
HEX-9	Specifies a four-byte MAC and presents it as two groups of four hexadecimal characters separated by a space character.

chaining_vector
The *chaining_vector* parameter is a pointer to an 18-byte string variable the security server uses as a work area to carry segmented data between procedure-calls.

Note: The application program must not change the data in this string.

MAC
The *MAC* parameter is a pointer to a string variable containing the trial MAC. The value must be left-justified in the field. Nine bytes are sent to the security server. The verb verifies the MAC if you specify the **ONLY** or **LAST** keyword for the segmenting control. Ensure that this parameter is a pointer to a 9-byte variable in application data storage.

Required Commands

The MAC_Verify verb requires the Verify MAC command (offset X'0011') to be enabled in the hardware.

Chapter 7. Key Storage Verbs

Figure 7-1. Key Storage Record Services

Verb	Page	Service	Entry Point	Svc Lcn
DES_Key_Record_Create	7-4	Creates a key record in DES key storage.	CSNBKRC	S
DES_Key_Record_Delete	7-5	Deletes a key record or deletes the key token from a key record in DES key storage.	CSNBKRD	S
DES_Key_Record_List	7-7	Lists the key-names of the key records in DES key storage.	CSNBKRL	S
DES_Key_Record_Read	7-9	Reads a key token from DES key storage.	CSNBKRR	S
DES_Key_Record_Write	7-10	Writes a key token into DES key storage.	CSNBKRW	S
PKA_Key_Record_Create	7-11	Creates a record in the public-key key-storage.	CSNDKRC	S
PKA_Key_Record_Delete	7-13	Deletes a record or deletes the key token from a record in public-key key-storage.	CSNDKRD	S
PKA_Key_Record_List	7-15	Lists the key-names of the records in public-key key-storage.	CSNDKRL	S
PKA_Key_Record_Read	7-17	Reads a key token from public-key key-storage.	CSNDKRR	S
PKA_Key_Record_Write	7-19	Writes a key token in public-key key-storage.	CSNDKRW	S
Retained_Key_Delete	7-21	Delete a key retained within the cryptographic engine.	CSNDRKD	E
Retained_Key_List	7-22	List the public and private RSA keys retained within the cryptographic engine.	CSNDRKL	E
Svc Lcn: Service location: E=Cryptographic Engine, S=Security API software				

This chapter describes how you can use key-storage mechanisms and the associated verbs for creating, writing, reading, listing, and deleting records in key storage.

Key Labels and Key Storage Management

Use the verbs described in this chapter to manage key storage. The CCA support software manages key storage as an indexed repository of key records. Access key storage through the use of a key label.

There are several independent key storage systems to manage records for DES key records and for PKA key records. DES key storage holds internal DES key tokens. PKA key storage holds both internal and external public and private RSA key tokens.

Also, public and private RSA keys can be retained within the Coprocessor. Public RSA keys are loaded into the Coprocessor through use of the PKA_Public_Key_Hash_Register and PKA_Public_Key_Register verbs. 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 usually be retained within the Coprocessor.

© Copyright IBM Corp. 1997-98

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 Key_Record_Create verb.

Use the Key_Record_Delete verb to delete a key token from a key record, or to delete both the key token and the key record.

Use the Key_Record_List verb to determine the existence of key records in key storage. The Key_Record_List verb creates a key record list data set with information about select key records. The wild card character (*) is used to obtain information about multiple key records. The data set can be read using conventional workstation-data-management services.

Individual key tokens can be read or written using the Key_Record_Read or Key_Record_Write verbs.

Key Label Content

Use a key label to identify a record or records in key storage managed by a CCA implementation. The key label must be left-justified in the 64-byte string variable used as input to the verb. Some verbs specify use of a key label while others specify use of 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:

- If the first character is within the range X'20' through X'FE', the input is be 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 label is terminated by a space character on the right (ASCII X'20', EBCDIC X'40'). The remainder of the 64-byte field is padded with space characters.
- Construct a label with one to seven *name_tokens*, each separated by a period (“.”). The key label must not end with a period.
- A name_token consists of one-to-eight characters in the character set A...Z, 0...9, and three additional characters relating to different character symbols in the various national language character sets as listed below:

ASCII Systems	**EBCDIC Systems**	**USA Graphic (for reference)**
X'23'	X'7B'	#
X'24'	X'5B'	$
X'40'	X'7C'	@

  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.

  The first character of the key label can not be numeric (0, ..., 9).

Notes:

1. Some CCA implementations accept the characters a...z and fold these to their upper case equivalents A...Z. 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 may be "tokenized" by dropping the periods and formatting it into eight-byte groups padded with space characters.

Some verbs accept a key label containing a "wild card"; an asterisk ("*") represents the wild card (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 not valid key labels include the following:

```
A/.B (includes an unacceptable character, "/")
ABCDEFGH9  (name token too long)
11111111.2.3.4.55555  (first character numeric)
A1111111.2.3.4.55555.6.7.8  (too many name tokens)
BANKSYS.XXXXX.*43*.D  (more than one wild card in a name token).
```

DES_Key_Record_Create (CSNBKRC)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	Basic

The DES_Key_Record_Create verb adds a key record to DES key storage. It is identified by the key label specified using the *key_label* parameter.

After creating a key record, you can use any of the following verbs to add or update a key token in the key record:

- DES_Key_Record_Write
- Data_Key_Import
- Key_Import
- Key_Part_Import
- Key_Generate

To delete a key record, you must use the DES_Key_Record_Delete verb.

Restrictions

None.

Format

CSNBKRC			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_label	Input	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

key_label
The *key_label* parameter is a pointer to a 64-byte string variable containing key label of the key record to be created.

Required Commands

The DES_Key_Record_Create verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the access control system.

DES_Key_Record_Delete (CSNBKRD)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	Basic

The DES_Key_Record_Delete verb does the following:

- Replaces the token in key record with a null token.
- Deletes an entire key record, including the key label, from key storage.

Identify the task with the *rule_array* keyword, and the key record with the *key_label*. To identify multiple records, use a wild card (*) in the key label.

Restrictions

None.

Format

CSNBKRD			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_label	Input	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero or one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 7-2. Key_Token_BuildRule_Array Keywords

Keyword	Meaning
TOKEN-DL	Deletes a key token from a key record in key storage. This is the default.
LABEL-DL	Deletes an entire key record, including the key label, from key storage.

key_label

The *key_label* parameter is a pointer to a 64-byte string variable containing the key label of a key token record in key storage. In a key label, use a wild card (*) to identify multiple records in key storage.

Required Commands

| The DES_Key_Record_Delete verb requires the Compute Verification Pattern
| command (offset X'001D') to be enabled in the access control system.

DES_Key_Record_List (CSNBKRL)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	Basic

The DES_Key_Record_List verb creates a Key Record List data set containing information about specified key records in key storage. Information 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 a key_label of *, *.*, *.*.*, etc.

The verb creates the list data set and returns the name of the data set and and the length of the data set name to the calling application. This data set has a header record, followed by 0 to *n* detail records, where *n* is the number of key records with matching key labels. For information about the header and detail records, see "Key Record List Data Set" on page B-16.

Restrictions

None.

Format

CSNBKRL			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_label	Input	String	64 bytes
data_set_name_length	Output	Integer	
data_set_name	Output	String	data_set_name_length bytes
security_server_name	Output	String	8 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_label
The *key_label* parameter is a pointer to a 64-byte 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.

data_set_name_length
The *data_set_name_length* parameter is a pointer to an integer variable containing the length of the name returned in the *data_set_name* variable.

data_set_name

The *data_set_name* parameter is a pointer to a 64-byte string variable where the verb returns the name of the data set containing the key record information. The data_set_name is left justified in the field.

The verb returns the data_set_name as a fully qualified file specification (for example, *C:\PKADIR\KYRLTnnn.LST* in the OS/2 environment), where *nnn* is the numeric portion of the name. This value increases by one every time you use this verb; when it reaches 999, the value is reset to 001.

Note: When the verb stores a key_Record_List data set, it overlays any older data set ith the same nnn value in its name.

security_server_name

The *security_server_name* parameter is a pointer to an eight-byte string variable. The information in this variable will not be used, but you must identify the variable.

Required Commands

The DES_Key_Record_List verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the access control system.

DES_Key_Record_Read (CSNBKRR)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	Basic

The DES_Key_Record_Read verb copies a key token from key storage to application data storage. The returned key token can be null.

Restrictions

This service does not have any restrictions.

Format

CSNBKRR			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_label	Input	String	64 bytes
key_token	Output	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

key_label
The *key_label* parameter is a pointer to a 64-byte string variable containing the key label of the record to be read from key storage.

key_token
The *key_token* parameter is a pointer to a 64-byte string variable to contain the token read from key storage.

Required Commands

| The DES_Key_Record_Read verb requires the Compute Verification Pattern
| command (offset X'001D') to be enabled in the access control system.

DES_Key_Record_Write (CSNBKRW)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	Basic

The DES_Key_Record_Write verb copies an internal DES key token from application data storage into DES key storage.

Before you use the DES_Key_Record_Write verb, use DES_Key_Record_Create to create a key record.

Restrictions

None.

Format

CSNBKRW			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_token	Output	String	
key_label	Input	String	64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

key_token
The *key_token* parameter is a pointer to a string variable containing the DES internal key token to be written into key storage.

key_label
The *key_label* parameter is a pointer to a 64-byte string variable containing the key label that identifies the record in key storage where the key token is to be written.

Required Commands

The DES_Key_Record_Write verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the access control system.

PKA_Key_Record_Create (CSNDKRC)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Key_Record_Create service adds a key record to PKA key storage. The new key record may be a null key token or a valid PKA internal or external token. It is identified by the key label specified with the *key_label* parameter.

After creating a key record, you can use any of the following verbs to add or update a key token in the record:

- PKA_Key_Import
- PKA_Key_Generate

To delete a key record, you must use the PKA_Key_Record_Delete verb.

Restrictions

None.

Format

CSNDKRC			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_label	Input	String	64 bytes
key_token_length	Input	Integer	
key_token	Input	String	key_token_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. Currently this verb does not require keywords and this field is ignored.

key_label
The *key_label* parameter is a pointer to a 64-byte string variable containing the key label of the key record to be created.

key_token_length
The *key_token_length* parameter is a pointer to an integer variable containing the length (in bytes) of the key_token to be written to key storage. If key_token_length contains zero, a record with a null PKA key token is created.

key_token
The *key_token* parameter is a pointer to a 64-byte string variable containing the the key token being written to key storage.

Required Commands

| The PKA_Key_Record_Create verb requires the Compute Verification Pattern
| command (offset X'001D') to be enabled in the access control system.

PKA_Key_Record_Delete (CSNDKRD)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Key_Record_Delete verb does the following:

- Replaces the token in key record with a null token.
- Deletes an entire key record, including the key label, from key storage.

Identify the task with the *rule_array*, and the key record with the *key_label*. To identify multiple records, use a wild card (*) in a key label.

Restrictions

None.

Format

CSNDKRD			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_label	Input	String	

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* may be zero or one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 7-3. Key_Token_BuildRule_Array Keywords

Keyword	Meaning
TOKEN-DL	Deletes a key token from a key record in key storage. This is the default.
LABEL-DL	Deletes an entire key record, including the key label, from key storage.

key_label
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, use a wild card (*) to identify multiple records in key storage.

Required Commands

The PKA_Key_Record_Delete verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the access control system.

PKA_Key_Record_List (CSNDKRL)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Key_Record_List verb creates a Key Record List data set containing information about specified key records in key storage. Information includes whether record validation is correct, the type of key, and the date and time when 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 a key label.

Note: To list all the labels in key storage, speicfy a key_label of *, *.*, *.*.*, etc.

The verb creates the list data set and returns the name of the data set and the length of the data set name to the calling application. The verb also returns the name of the security server where the data set is stored. The PKA_Key_Record_List data set has a header record, followed by 0 to *n* detail records, where *n* is the number of key records with matching key labels. For information about the header and detail records, see "Key Record List Data Set" on page B-16.

Restrictions

None.

Format

CSNDKRL

return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_label	Input	String	64 bytes
data_set_name_length	Output	Integer	
data_set_name	Output	String	data_set_name_length bytes
security_server_name	Output	String	8 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified,

and padded on the right with space characters. Currently this verb does not require keywords and this field is ignored.

key_label
The *key_label* parameter is a pointer to a 64-byte string variable containing a key record in key storage. You can use a wild card (*) to identify multiple records in key storage.

data_set_name_length
The *data_set_name_length* parameter is a pointer to an integer variable containing the length of the name returned in the data_set_name variable.

data_set_name
The *data_set_name* parameter is a pointer to 64-byte string variable where the verb returns the name of the data set containing the key record information. The data_set_name is left justified in the field.

The verb returns the data_set_name as a fully qualified file specification (for example, *C:\PKADIR\KYRLTnnn.LST* in the OS/2 environment), where *nnn* is the numeric portion of the name. This value increases by one every time you use this verb; when it reaches 999, the value is reset to 001.

Note: When the verb stores a key_Record_List data set, it overlays any older data set ith the same nnn value in its name.

security_server_name
The *security_server_name* parameter is a pointer to an eight-byte string variable. The information in this variable is not used, but it must be identified.

Required Commands

| The PKA_Key_Record_List verb requires the Compute Verification Pattern
| command (offset X'001D') to be enabled in the access control system.

PKA_Key_Record_Read (CSNDKRR)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Key_Record_Read verb copies a key token from key storage to application data storage.

The returned key token may be null. In this event, the key_length variable contains a value of eight and the key token variable contains eight bytes of X'00' beginning at offset zero (see "Null Key Token" on page B-2).

Restrictions

None.

Format

CSNDKRR			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_label	Input	String	64 bytes
key_token_length	In/Out	Integer	
key_token	Output	String	key_token_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. Currently this verb does not require keywords and this field is ignored.

key_label
The *key_label* parameter is a pointer to a 64-byte string variable containing the key label of the the record to be read from key storage.

key_token_length
The *key_token_length* parameter is a pointer to an integer variable containing the length (in bytes) of the key_token variable. This variable must be large enough to hold the key token beinf read. On successful completion, key_token_length contains the actual length of the token being returned. The maximum size is 2500 bytes.

key_token
The *key_token* parameter is a pointer to a string variable where the PKA token being read from key storage is to be returned.

Required Commands

None.

The PKA_Key_Record_Read verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the access control system.

PKA_Key_Record_Write (CSNDKRW)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The PKA_Key_Record_Write verb copies an internal or external PKA key token from application data storage into key storage.

There are two processing options:

- Write the new token only if the old token was null.
- Write the new token regardless of content of the old token.

Before you use the PKA_Key_Record_Write verb, use the PKA_Key_Record_Create to create a key record.

Restrictions

None.

Format

CSNDKRW			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
key_label	Input	String	6a4 bytes
key_token_length	Input	Integer	
key_token	Input	String	key_token_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero or one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight-bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Figure 7-4. Key_Token_BuildRule_Array Keywords	
Keyword	**Meaning**
CHECK	Specifies that the record will be written only if a record of the same label in key storage contains a null token. This is the default.
OVERLAY	Specifies that the record will be overwritten regardless of the current content of the record.

key_label
The *key_label* parameter is a pointer to a 64-byte string variable containing the ley label that identifies the key record in key storage where the key token is to be written.

key_token_length
The *key_token_length* parameter is a pointer to an integer variable containing the size (in bytes) of the key_token.

key_token
The *key_token* parameter is a pointer to a string variable containing the the PKA key token to be written into key storage.

Required Commands

| The PKA_Key_Record_Write verb requires the Compute Verification Pattern
| command (offset X'001D') to be enabled in the access control system.

Retained_Key_Delete (CSNDRKD)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The Retained_Key_Delete verb deletes a key that has been retained within the Coprocessor.

You can retain both public and private keys within the Coprocessor through the use of verbs such as PKA_Key_Generate and PKA_Public_Key_Register. A list of retained keys can be obtained with the use of the Retained_Key_List verb.

Restrictions

None.

Format

CSNDRKD			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0
rule_array	Input	String array	rule_array_count * 8 bytes
key_label	Input	String	

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero for this verb.

rule_array
The *rule_array* parameter should be a null address pointer.

key_label
The *key_label* parameter points to a 64-byte string variable containing the key label of a key that has been retained within the Coprocessor.

Required Commands

The Retained_Key_Delete verb requires the Delete Retained Key command (offset X'0203') to be enabled in the hardware.

Retained_Key_List (CSNDRKL)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PKA96

The Retained_Key_List verb lists the key labels of those keys that have been retained within the Coprocessor.

You can filter the set of key labels returned to your application through the use of the key label mask input variable. Specify the keys to be listed using the key_label_mask variable; to identify multiple keys, use the wild card (*) in a mask. To list all retained key labels, specify a mask of *, *.*, *.*.*, etc. **(see restriction below).**

You can retain both public and private keys within the Coprocessor through the use of verbs such as PKA_Key_Generate and PKA_Public_Key_Register. You can delete retained keys with the use of the Retained_Key_Delete verb.

Restrictions

With the initial implementation of this verb, the key_name_mask variable must be a single asterisk (*) followed by 63 space characters. Filtering of returned key labels is not supported at this time.

Format

CSNDRKL

return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0
rule_array	Input	String array	rule_array_count * 8 bytes
key_label_mask	Input	String	64 bytes, null pointer
retained_keys_count	Output	Integer	
key_labels_count	In/Output	Integer	
key_labels	Output	String	key_labels_count * 64 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule array. The value of the *rule_array_count* must be zero for this verb.

rule_array
The *rule_array* parameter should be a null address pointer.

key_label_mask
The *key_label_mask* parameter points to a 64-byte string variable containing a key label mask that is used to filter the list of key names returned by the

verb. You can use a wild card (*) to identify multiple keys retained within the Coprocessor.

Note: The initial implementation requires this variable to be a single asterisk (*) followed by 63 space characters.

retained_keys_count
The *retained_keys_count* parameter points to an integer variable to receive the number of retained keys stored within the Coprocessor.

key_labels_count
The *key_labels_count* parameter points 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. The value returned in the registered_keys_count variable can be larger if you have not provided for the return of a sufficiently large number number of key_labels.

key_labels
The *key_labels* parameter points to a string array variable. The Coprocessor returns zero or more 64-byte entries that each contain a key label of a key retained within the Coprocessor.

Required Commands

The Retained_Key_List verb requires the List Retained Key command (offset X'0230') to be enabled in the hardware.

Chapter 8. Financial Services Support Verbs

Figure 8-1. Financial Services Support Verbs

Verb	Page	Service	Entry Point	Svc Lcn
Clear_PIN_Encrypt	8-12	This verb formats a PIN into a PIN-block and outputs the PIN-block as an encrypted quantity. The keyword **RANDOM** represents an extension to the support available with other CCA implementations. to generate random PINs that are output in encrypted PIN blocks.	CSNBCPE	E
Clear_PIN_Generate	8-15	This verb generates a clear PIN, or a PIN offset.	CSNBPGN	E
Clear_PIN_Generate_Alternate	8-18	This verb extracts a customer-selected PIN or instution-assigned PIN from an encrypted PIN-block and generates a PIN offset.	CSNBCPA	E
Encrypted_PIN_Generate	8-24	This verb generates a PIN from an account number and other information and returns the result in an encrypted PIN block.	CSNBEPG	E
Encrypted_PIN_Translate	8-29	This verb operates in two modes. Translate mode re-encrypts a PIN block under a different key. Reformat mode does one or more of the following: • Reformats a PIN from one PIN block format into another PIN block format • Changes selected non-PIN digits in a PIN block • Re-encrypts a PIN block.	CSNBPTR	E
Encrypted_PIN_Verify	8-34	This verb extracts and verifies a PIN by using the specified PIN calculation method.	CSNBPVR	E
SET_Block_Compose	8-40	Creates a SET-protocol RSA-OAEP block and DES encrypts the data block in support of the SET protocols.	CSNDSBC	
SET_Block_Decompose	8-44	Decomposes the RSA-OAEP block and DES decrypts the data block in support of the SET protocols.	CSNDSBD	
Svc Lcn: Service location: E=Cryptographic Engine, S=Security API software				

There are two classes of verbs described in this chapter:

- Finance industry PIN processing verbs. Information common to those verbs is described in the next section.
- SET-related verbs; these verbs support cryptographic operations as defined in the Secure Electronic Transaction (SET) protocol as defined by VISA International and Mastercard; see their Web pages for a reference to the SET protocol.

Processing Financial PINs

© Copyright IBM Corp. 1997-98

This section describes how the financial personal identification number (PIN) verbs allow you to process financial PINs. A financial PIN is used to authorize personal financial transactions for a customer who uses an automated teller machine.[1] A financial PIN is similar to a password except that a financial PIN consists of decimal digits and is normally a cryptographic function of an associated account number. The financial PIN verbs support PINs that range from 4 to 16 digits in length. (A financial PIN is usually 4 to 6 digits in length.)

The financial PIN verbs form a complete set of verbs that you can use in various combinations to process financial PINs. You use these verbs, whose relationships and primary inputs and outputs are depicted in Figure 8-2 on page 8-3, to do the following:

- Provide security for the PINs by supporting encrypted PIN-blocks with these capabilities:
  - Encryption of a clear PIN in various PIN-block formats
  - Generation of random PIN values and encryption of these in various PIN-block formats
  - Verification of a PIN, the PIN-block is decrypted as part of the verification service
  - Re-encrypting of a PIN-block under another key with optional, integral changing of the PIN-block format.
- Support multiple PIN calculation methods
- Support multiple PIN-block formats and PIN extraction methods
- Provide the following services:
  - Create encrypted PIN blocks for transmission
  - Generate institution-assigned PINs
  - Generate an offset or a VISA PIN-validation value (PVV)
  - Create encrypted PIN blocks for a PIN-verification database
  - Change the PIN block encrypting key or the PIN-block format
  - Verify PINs.

Normally, a customer inserts a magnetic-stripe card and enters a PIN (a *trial PIN*) into an automated teller machine to identify himself. The automated teller machine does the following:

- Obtains account information and other information from the magnetic stripe on the card.
- Formats the trial PIN into a *PIN block* and encrypts the PIN block.
- Sends the information from the card, the encrypted PIN block, and other data in a message to a host program for verification.

To verify a PIN, a program normally uses one of the following two methods:

- PIN calculation method. In this method, the program calls the PIN verification verb that decrypts the trial PIN block, extracts the trial PIN from the PIN block, re-calculates the account-number based PIN, adjusts this

[1] In this chapter, automated teller machine (ATM) can also mean a point-of-sale device, an enhanced teller terminal, or a programmable workstation, unless noted otherwise.

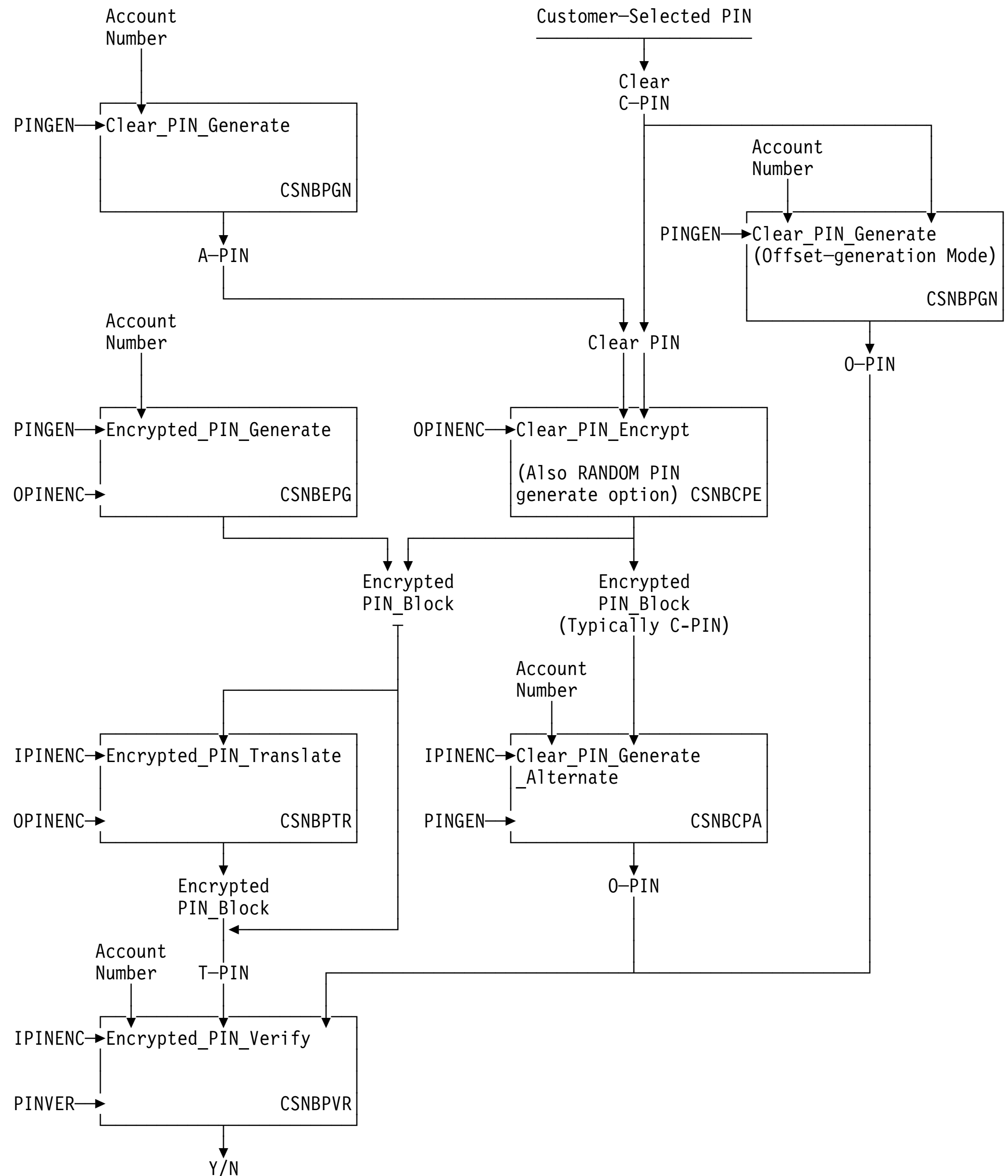

Figure 8-2. Financial PIN Verbs

value with any *offset*, compares the resulting value to the trial PIN, and returns the results of the comparison.

- PIN database method. In this method, the encrypted PIN block that contains the correct customer-PIN is stored in a PIN-verification database. Upon receipt of an encrypted trial-PIN block, the program calls a verb to translate (decipher, then encipher) the trial PIN block to the format and key used for the encrypted PIN block in the PIN-verification database. The two encrypted PIN blocks can then be compared for equality.

In general, a PIN can be assigned by an institution or selected by a customer. Some PIN calculation methods use the institution-assigned or customer-selected PIN to calculate another value that is stored on the magnetic stripe of the account-holder's card or in a data base and that is used in the PIN-verification process.

The following terms are used for the various “PIN” values:

A-PIN The quantity derived from a function of the account number, and PIN-generating key, and other inputs such as a *decimalization table*.

C-PIN The quantity that a customer *should use* to identify himself; in general, this can be a customer-selected or institution-assigned quantity.

O-PIN A quantity, sometimes called an *offset*, that relates the A-PIN to the C-PIN as permitted by certain calculation methods.

T-PIN The *trial* PIN presented for verification.

PIN Verb Summary

The *Clear_PIN_Generate* verb (CSNBPGN) uses a PIN-generating key and an account number to create an A-PIN according to the calculation method selected through a rule-array keyword. See “PIN Calculation Methods” on page E-1. Certain calculation methods also accept a C-PIN value and return an O-PIN calculated from the Coprocessor generated and retained A-PIN value.

The *Encrypted_PIN_Generate* verb (CSNBEPG) uses a PIN-generating key and an account number to create an A-PIN according to the calculation method selected through a rule-array keyword. The verb formats the A-PIN value into a PIN block as specified in the input control information. The PIN block is returned encrypted by the supplied OPINENC-type key.

The *Clear_PIN_Encrypt* verb (CSNBCPE) accepts a PIN value and formats the input into a PIN block. The result is encrypted and returned. This verb can also randomly generate PIN values and return these as encrypted PIN blocks; this function is useful when an institution wishes to distribute (initial) PIN values to its customers.

The *Clear_PIN_Generate_Alternate* verb (CSNBCPA) accepts an encrypted PIN block that would normally contain a customer-selected C-PIN value. The verb calculates the A-PIN from the account number and PIN-generating key and then derives the O-PIN as a function of the A-PIN and the C-PIN; the O-PIN is returned in the clear.

The *Encrypted_PIN_Verify* verb (CSNBPVR) accepts an account number and PIN-verifying or PIN-generating key to internally produce an A-PIN. For certain methods, the verb also accepts an O-PIN so that it can produce the correct value that a customer should enter to access his account. The final input, an encrypted T-PIN block, is decrypted, the customer-entered trial PIN is extracted from the block and compared to the calculated value; equality or inequality is indicated by the return code values (and reason code values). Return code 0 indicates the PIN is validated while code 4 indicates that the trial PIN failed validation.

The *Encrypted_PIN_Translate* verb (CSNBPTR) is used to change the key used later to decrypt or compare the PIN block. The verb can also extract the PIN from one PIN block format and insert the PIN into another PIN block format before re-encryption. This service is useful when transferring PIN blocks from one domain to another.

PIN Calculation Method and PIN Block Format Summary

As described in the following sections, you can use a variety of PIN calcualtion methods and a variety of PIN block formats with the various PIN processing verbs. Figure 8-3 provides a summary of the supported combinations.

Figure 8-3. PIN Verb, PIN Calculation Method, and PIN-block Format Support Summary

Verb / Calcualtion Method, PIN Block	**Entry Point**	**IBM-PIN, IBM-PINO**	**VISA-PVV**	**GBP-PIN**	**INBK-PIN**	**NL-PIN-1**	**3624**	**ISO-0**	**ISO-1**	**ISO-2**
Clear_PIN_Encrypt	CSNBCPE						√	√	√	√
Clear_PIN_Generate	CSNBPGN	√								
Clear_PIN_Generate_Alternate	CSNBCPA	√	√			√	√	√	√	
Encrypted_PIN_Generate	CSNBEPG			√	√	√	√	√	√	√
Encrypted_PIN_Translate	CSNBPTR						√	√	√	√
Encrypted_PIN_Verify	CSNBPVR	√	√	√	√		√	√	√	√

Providing Security for PINs

It is important to maintain the security of PINs. Unauthorized knowledge of a PIN and its associated account number can result in fraudulent transactions. One method of maintaining the security of a PIN is to store the PIN in a *PIN block*, encrypt the PIN block, and only send or store a PIN in this form. A PIN block is 64 bits in length, which is the length of data on which the DES algorithm operates. A PIN block consists of both PIN *digits* and non-PIN digits. The non-PIN digits pad the PIN digits to a length of 64 bits. When discussing PINs, the term *digit* refers to a 4-bit quantity that can be valued to the decimal values 0...9 and in some cases also to the hexadecimal values A...F. Several different PIN block formats are supported. See “PIN Block Formats” on page E-8.

The non-PIN digits can also add variability to a PIN block. Varying the value of the non-PIN digits in a PIN block is a security measure used to create a large number of different encrypted PIN blocks, even though there are typically only 10,000 PIN values in use. To enhance the security of a clear PIN during PIN processing, the verbs generally operate with encrypted PIN blocks. The PIN verbs provide high-level services that typically insert or extract PIN values to or from a PIN block internal to the verb.

The following verbs receive clear PINs from your application program or return clear PINs to your program; none of the other PIN verbs reveal a clear PIN:

- Clear_PIN_Generate
- Clear_PIN_Encrypt.

When your application program supplies a clear PIN to a verb or receives a clear PIN from a verb, ensure that adequate access controls and auditing are provided to protect this sensitive data. Also recognize that exhaustive use of certain verbs such as Encrypted_PIN_Verify and Clear_PIN_Generate_Alternate can reveal the value of a PIN; therefore if production level keys are available in a system, be sure that you have usage controls and auditing in effect to detect inappropriate usage of these verbs.

Using Specific Key Types and Key-Usage Bits to Help Ensure PIN Security

The control vectors (see Appendix C, "CCA Control Vector Definitions and Key Encryption" on page C-1) associated with obtaining and verifying PINs enable you to minimize certain security exposures. The class of keys designated *PINGEN* operates in the verbs that create and validate PIN values, whereas the *PINVER* class operates only in those verbs that validate a trial PIN. Reduce your exposure to fraud by limiting the availability of the PINGEN keys to those applications and times when it is legitimate to create new PIN values. Use the PINVER key class to validate trial PINs. You can also further restrict those verbs in which a PINGEN key will perform by selectively turning off bits in the default PINGEN control vector.

Those verbs that encrypt a PIN block require the encrypting key to be of the class *OPINENC*, output PIN (block) encrypting key. Those verbs that decrypt a PIN block require the encrypting key to be of the class *IPINENC*, input PIN (block) encrypting key. The actual input and output key values are the same, but the use of two different types of control vectors aids in defeating certain *insider attacks* that might enable redirection of encrypted PIN values to an unintended service to the attacker's benefit. You can also turn off selected bits in the default OPINENC and IPINENC control vectors to limit those verbs in which a given key can operate to further reduce exposure to insider fraud.

In summary, the PIN verbs use these key types:

PINGEN (PIN-generating) key type

The PIN verbs that generate and verify a PIN require the PIN-generating key to have a control vector that specifies a PINGEN key type.

The Encrypted_PIN_Verify verb can also use a key with a PINGEN key type if bit 22 is set to 1 to specify that the key can be used to verify a PIN.

PINVER (PIN-verifying) key type

The Encrypted_PIN_Verify verb, which verifies an encrypted PIN by using the PIN calculation method, requires the PIN-generating key to have a control vector that specifies the PINVER key type, or a control vector that specifies the PINGEN key type and has bit 22 set to 1. Note that the PINVER key type can not be used to create a PIN value, and therefore is the preferred key type in a system that only needs to validate PINs.

IPINENC (input PIN-block encrypting) key type

The PIN verbs that decrypt a PIN block require the decrypting key to have a control vector that specifies an IPINENC key type.

OPINENC (output PIN-block encrypting) key type

The PIN verbs that encrypt a PIN block require the encrypting key to have a control vector that specifies an OPINENC key type.

Supporting Multiple PIN Calculation Methods

The PIN verbs support multiple PIN calculation methods. You use a *data_array* variable to supply information that a PIN calculation method requires.

PIN Calculation Methods

A PIN calculation method determines the value of an A-PIN in relationship to an account number; the methods are described in “PIN Calculation Methods” on page E-1. The PIN verbs support the following PIN calculation methods, which you specify with a keyword in the *rule_array* variable for a verb:

PIN Calculation Method	Keyword
IBM 3624 PIN	IBM-PIN
IBM 3624 PIN Offset	IBM-PINO
Netherlands PIN-1	NL-PIN-1
IBM German Bank Pool Institution PIN	GBP-PIN
VISA PIN-Validation Value (PVV)	VISA-PVV
Interbank PIN	INBK-PIN

Data_Array

To supply the information that a PIN calculation method requires, the PIN verbs use a *data_array* variable. Depending on the calculation method and the verb, the data array elements can include a decimalization table, validation data, an offset or clear PIN, or transaction security data.

The data array is a 48-byte string made up of three consecutive 16-byte character strings. Each element must be 16 bytes in length, uppercase, left-justified, and padded on the right with space characters. Some PIN calculation methods and verbs do not require all three elements; however, all three elements must be declared.

Data Array with IBM-PIN, IBM-PINO, NL-PIN-1, GBP-PIN: When using the IBM-PIN, the IBM-PINO, the NL-PIN-1, or the IBM German Bank Pool PIN method, the data array contains elements for a decimalization table, validation data, and for certain verbs, a clear PIN or an offset.

- **Decimalization_Table**

  The first element in the data array for a PIN calculation method points to the decimalization table of 16 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').

  Note: To avoid errors when using the IBM 3624 PIN block format, you should not include in the decimalization table a decimal digit that is also used as a pad digit. For information about a pad digit, see “PIN Profile” on page 8-9.

- **Validation_Data**

  The second element in the data array for a PIN calculation method points to 1 to 16 characters of account data, which can be the customer's account number or other identifying number. If necessary, the application program must left-justify the validation data and pad on the right with space characters to a length of 16 bytes.

- **Clear_PIN**, **Offset_Data**, or **Reserved**

  The third element in the data array contains an O-PIN value. If an O-PIN is not used in the verb or method, then 16 space characters.

Data Array with the VISA PVV Calculation Method: When using the VISA PVV calculation method, the data array consists of the transaction_security_parameter, the PVV, and one reserved element.

- **Transaction_Security_Parameter**

  The first element in the data array for the VISA PVV calculation method points to transaction security data. Specify 16 characters that include the following:

  – Eleven (rightmost) digits of personal account number (PAN) data, excluding the check digit. For information about a PAN, see "Personal Account Number (PAN)" on page 8-11.
  – One digit of key index from 1 to 6.
  – Four space characters.

- **Referenced PVV**

  When using the Encrypted_PIN_Verify verb, the second element in the data array for the VISA PVV calculation method contains 4 numeric characters, which are the PVV value for the account and derived from a customer-selected PIN value. This value is followed by 12 space characters.

- **Reserved**

  The second element (when not using the Encrypted_PIN_Verify verb) and the third element in the data array for the VISA PVV calculation method are reserved. These elements point to 16-byte variables in application data storage. The information in these elements will be ignored, but the elements must be declared.

Data Array for the Interbank Calculation Method: When using the Interbank PIN calculation method with certain verbs, the data array consists of one element, the transaction_security_parameter, for transaction security data. The other two elements are reserved.

- **Transaction_Security_Parameter**

  The first element in the data array for the Interbank calculation method points to transaction security data. Specify 16 numeric characters that include the following:

  – Eleven (rightmost) digits of PAN data, excluding the check digit. For information about a PAN, see "Personal Account Number (PAN)" on page 8-11.
  – A constant, 6.
  – A 1-digit key index selector from 1 to 6.

– Three numeric characters of validation data.

- **Reserved**

  The second and third elements in the data array for the Interbank calculation method are reserved. The elements point to 16-byte variables in application data storage. The information in these elements will be ignored, but the elements must be declared.

Supporting Multiple PIN-Block Formats and PIN Extraction Methods

The PIN verbs support multiple PIN-block formats, which you specify in a *PIN_profile* variable. The supported PIN block formats are described in "PIN Block Formats" on page E-8. Multiple methods for *extracting* the PIN value from the PIN block exist for certain PIN block formats. Depending on the PIN-block format, the verbs also require a pad digit, a personal account number (PAN), and/or a sequence number.

This section describes the following:

- The PIN profile variable
- The PIN extraction methods
- The Personal Account Number (PAN).

PIN Profile

A PIN profile variable consists of three elements. The elements identify the PIN-block format, the level of format control, and any pad digit. Generally you can code a PIN profile as a constant in your application. Each element is an 8-byte character string in an array, which is the equivalent of a single 24-byte string that is organized as three 8-byte fields. The elements must be 8 bytes in length, uppercase, and, depending on the element, either left- or right-justified and padded with space characters. Depending on the verb and the PIN-block format, all three elements might not be used; however, you must declare all three elements (24 bytes).

PIN-Block Format: The PIN-block format is the first element in a PIN profile variable. You specify the format through the use of one of these keywords:

PIN-Block Format	Keyword
IBM 3624	3624
ISO-0 (equivalent to ANSI X9.8, VISA format 1, and ECI-1 formats)	ISO-0
ISO-1 (same as the ECI-4 format)	ISO-1
ISO-2	ISO-2

Format Control Enforcement: The format-control level is the second element in a PIN profile. For the IBM 4758 implementation, this element must be set to **NONE** followed by four space characters.

Pad Digit: The pad digit is the third element in a PIN profile. Certain PIN-block formats require a pad digit when a PIN is formatted or extracted, or both, as shown in Figure 8-4. The *PIN Formatting* column indicates the values that the verb uses when it creates a PIN block. The *PIN Extraction* column indicates the values that the verb uses when it extracts a PIN from a PIN block.

When required, specify the pad digit as a character from the character set 0 through 9 and A through F. The pad digit must be uppercase, right-justified in the 8-byte element, with 7 preceding space characters. When a pad digit is not required, specify eight space characters.

Note: For the IBM 3624 PIN-block format, the pad digit should be a non-decimal character (in the range from C'A' to C'F'). The 3624 PIN-block format depends on the fact that the pad digit is not the same as a PIN digit. If they are the same, unpredictable results can occur. For this reason, it is strongly recommended that you do not use a decimal digit for the pad digit. (If you use a decimal digit for the pad digit, you also limit the range of possible PINs.)

If you use a decimal digit for the pad digit, ensure that you do not include the decimal digit in the decimalization table. For information about the decimalization table, see "Data_Array" on page 8-7.

Figure 8-4. Pad-Digit Specification by PIN-Block Format

PIN-Block Format Keyword	Pad Digit for PIN Formatting	Pad Digit for PIN Extraction
3624	0 through F	0 through F
ISO-0	F	The pad-digit specification will be ignored.
ISO-1	The pad-digit specification will be ignored.	The pad-digit specification will be ignored.
ISO-2	The pad-digit specification will be ignored.	The pad-digit specification will be ignored.

PIN Extraction Methods

Before a verb can process a formatted and encrypted PIN, the verb must decrypt the PIN block and extract the PIN from the PIN block. The PIN verbs support multiple PIN extraction methods. The valid PIN extraction methods depend on the PIN-block format.

You can specify a PIN extraction method or use the default method for the PIN-block format. To specify a PIN extraction method, you use a keyword in the *rule_array* parameter for the verb.

Figure 8-5 on page 8-11 shows the keywords for the PIN extraction methods that are valid for each PIN-block format. When only one PIN extraction method is valid, the keyword is the default value. When more than one method is valid, the first keyword is the default value.

Figure 8-5. PIN Extraction Method Keywords by PIN-Block Format

PIN-Block Format	PIN Extraction Method Keywords (Used in the Rule Array)
3624	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN16, PADEXIST
ISO-0	PINBLOCK
ISO-1	PINBLOCK
ISO-2	PINBLOCK

The PIN extraction methods operate as described:

PINBLOCK Depending on the contents of the PIN block, this keyword specifies that the verb use one of the following items to identify the PIN:

- 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 This keyword specifies that the verb use the pad value in the PIN profile to identify the end of the PIN.

HEXDIGIT This keyword specifies that the verb use the first occurrence of a digit in the range from X'A' to X'F' as the pad value to determine the PIN length.

PINLEN*xx* This keyword specifies that the verb use the length specified in the keyword, where *xx* can range from 04 to 16 digits, to identify the PIN.

PADEXIST This keyword specifies that the verb use the character in the 16th position of the PIN block as the value of the pad value.

Personal Account Number (PAN)

A personal account number (PAN) identifies an individual and relates that individual to an account at the financial institution. The PAN consists of the following:

- Issuer identification number
- Customer account number
- One check digit.

For the ISO-0 PIN-block format, the PIN verbs use a PAN to format and extract a PIN. You specify the PAN with a *PAN_data* parameter for the verb. You must specify the PAN in character format in a 12-byte field. Each digit in the PAN must be in the range from 0 to 9. The actual PAN might be more than 12 digits, but the PIN verbs use only 12 digits for the PAN. Depending on the PIN-block format, the verbs use the rightmost 12 digits or the leftmost 12 digits.

- When using the ISO-0 PIN-block format, use the rightmost 12 digits of the PAN, excluding the check digit.

Clear_PIN_Encrypt (CSNBCPE)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PINS

The Clear_PIN_Encrypt verb formats a PIN into one of the following PIN block formats and encrypts the results (see “PIN Block Formats” on page E-8):

- 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

You can use the Clear_PIN_Encrypt verb to create an encrypted PIN block for transmission. With the **RANDOM** keyword, you can also have the verb generate random PIN numbers. This can be useful when you supply PIN numbers to a bank-card manufacturer.

Note: A clear PIN is a sensitive piece of information. Ensure that your application program and system design provide adequate protection for any clear PIN value.

To use this verb, specify the following:

- A key used to encrypt the PIN block
- A clear PIN. When you generate random PINs, the clear PIN variable specifies the length of the generated PIN value by the number of numeral zero characters; the remainder of the variable must be padded with space characters.
- A PIN profile that specifies the format of the PIN block to be created, and any pad digit; see “PIN Profile” on page 8-9.
- When using the ISO-0 PIN block format, the *PAN_data* variable provides the account number that is exclusive-ORed with the PIN information.
- The sequence number for use in certain PIN block formats; for those PIN block formats that do not employ a sequence number, specify a value of 99999 in the integer variable.

The verb does the following:

- Formats the PIN into the specified PIN block format.
- Checks the control vector for the OPINENC key by doing the following:
  - Verifies that the CPINENC bit is 1.
- Encrypts the PIN block in ECB mode.
- Returns the encrypted PIN block in the encrypted_PIN_block variable.

Restrictions

The software must include support for the PINS function set.

Format

CSNBCPE			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
PIN_encrypting_key_identifier	Input	String	64 bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
clear_PIN	Input	String	16 bytes
PIN_profile	Input	String array	3 * 8 bytes
PAN_data	Input	String	12 bytes
sequence_number	Input	Integer	
encrypted_PIN_block	Output	String	8 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

PIN_encrypting_key_identifier
The *PIN_encrypting_key_identifier* parameter points to 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 bit set to 1.

rule_array_count
The *rule_array_count* parameter points to an integer containing the number of elements in the rule array.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
ENCRYPT	This is the default operation of the verb, use of the keyword is optional.
RANDOM	Causes the verb to generate a random PIN value. The length of the PIN is based on the value in the clear_PIN variable. Value 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.

clear_PIN
The *clear_PIN* parameter points to a 16-byte character string with the clear PIN. The values in this variable must be left-justified and padded on the right with space characters.

PIN_profile
The *PIN_profile* parameter points to a 24-byte string containing three 8-byte elements with: a PIN block format keyword, a format control keyword (**NONE**), and a pad digit as required by certain formats. See "PIN Profile" on page 8-9.

PAN_data
The *PAN_data* parameter points to a 12-byte PAN in character format. The verb uses this parameter if the PIN profile specifies the **ISO-0** keyword for the PIN block format. Otherwise, ensure that this parameter points to a 12-byte variable in application data storage. The information in this variable will be ignored, but the variable must be declared.

sequence_number
The *sequence_number* parameter points to a 4-byte character integer. The verb currently ignores the value in this variable. For future compatibility, the suggested value is `'99999'`.

encrypted_PIN_block
The *encrypted_PIN_block* parameter points to the variable to receive the 8-byte encrypted PIN block.

Required Commands

The Clear_PIN_Encrypt verb requires the Format and Encrypt PIN command (command offset `X'00AF'`) to be enabled in the hardware.

Clear_PIN_Generate (CSNBPGN)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PINS

The Clear_PIN_Generate verb verb generates an A-PIN or an O-PIN by using one of the following calculation methods that you specify with a rule array keyword (see "PIN Calculation Methods" on page E-1):

- IBM 3624 PIN (IBM-PIN)
- IBM 3624 PIN Offset (IBM-PINO).

You can use this verb to do the following:

- Generate a clear PIN for immediate use; for example, generate a clear A-PIN as part of PIN mailer processing
- Generate an offset (O-PIN) for use on a customer account magnetic stripe card.

Notes:

1. A clear PIN is a sensitive piece of information. Ensure that your application program and system design provide adequate protection for the clear PIN.
2. To format and *encrypt* a PIN, use the Clear_PIN_Encrypt verb.

To use this verb, specify:

- A PIN-generating key
- The number of rule array elements
- The PIN calculation method
- The length of the PIN
- For certain PIN calculation methods, an additional PIN length value with the PIN_check_length variable to determine the length of the O-PIN value
- A decimalization table, validation data (e.g. account number information) and, based on the PIN calculation method, the C-PIN value, in a character array
- A 16-byte variable to receive the clear PIN.

The verb does the following:

- Verifies that the CPINGEN bits are set to 1 in the control vector for the PINGEN key.
- Calculates the A-PIN, and optionally uses the C-PIN and the A-PIN to compute the O-PIN value. See "PIN Calculation Methods" on page E-1.
- Uses the specified PIN length to determine the length of the PIN.
- Returns the clear A-PIN or O-PIN in the variable identified by the *returned_result* parameter.

Restrictions

The software must include support for the PINS function set.

Format

CSNBPGN			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
PIN_generating_key_identifier	Input	String	64 bytes
rule_array_count	Input	Integer	
rule_array	Input	String	8 bytes
PIN_length	Input	Integer	
PIN_check_length	Input	Integer	
data_array	Input	String array	16 bytes * 3
returned_result	Output	String	16 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

PIN_generating_key_identifier
The *PIN_generating_key_identifier* parameter points to a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the PIN-generation key and must contain a control vector that specifies the PINGEN key type and has the CPINGEN bits set to 1.

rule_array_count
The *rule_array_count* parameter points to an integer for the number of the rule array elements. This value must be 1.

rule_array
The *rule_array* parameter points to a keyword that defines the PIN calculation method to use to generate the A-PIN or the O-PIN.

The keyword in the rule array must be 8 bytes in length, uppercase, left-justified, and padded on the right with space characters.

Keyword	Meaning
IBM-PIN	This keyword specifies the IBM 3624 PIN calculation method to be used to generate a PIN.
IBM-PINO	This keyword specifies the IBM 3624 PIN offset calculation method to be used to generate a PIN offset.

PIN_length
The *PIN_length* parameter points to an integer in the range from 4 to 16 for the length of the PIN. The verb uses the PIN length if you specify the **IBM-PIN** or the **IBM-PINO** keyword for the calculation method. Otherwise, ensure that this parameter points to a 4-byte variable in application data storage.

PIN_check_length

The *PIN_check_length* parameter points to an integer in the range from 4 to 16 for the length of the PIN offset. The verb uses the PIN check length if you specify the **IBM-PINO** keyword for the calculation method. Otherwise, ensure that this parameter points to a 4-byte variable in application data storage. The information in this variable will be ignored, but this variable must be declared.

Note: The PIN check length must be less than or equal to the PIN length.

data_array

The *data_array* parameter points to three 16-byte numeric 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. The verb converts the space characters to zeroes.

When using the **IBM-PIN** or the **IBM-PINO** keyword, identify the following elements in the data array.

Element	Description
decimalization_table	This element contains the decimalization table of 16 characters (0 to 9) that are used to convert the hexadecimal digits (X'0' to X'F') of the encrypted validation data to decimal digits (X'0' to X'9').
validation_data	This element contains 1 to 16 characters of account data. The data must be left-justified and padded on the right with spaces.
clear_PIN	When using the **IBM-PINO** keyword, this element contains the clear customer-selected PIN. This value must be left-justified and padded with spaces. When using the **IBM-PIN** keyword, this element is ignored but must be declared.

returned_result

The *returned_result* parameter points to the generated output 16-byte character string variable. The result will be left-justified and padded with space characters.

Required Commands

The Clear_PIN_Generate verb requires the following commands to be enabled in the hardware based on the keyword specified for the PIN calculation method.

PIN Calculation Method	Command Offset	Command
IBM-PIN IBM-PINO	X'00A0'	Generate Clear 3624 PIN

Clear_PIN_Generate_Alternate (CSNBCPA)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PINS

The Clear_PIN_Generate_Alternate verb is used to obtain a value,the "O-PIN" (offset or VISA-PVV) that will relate the institution-assigned PIN to the customer-known PIN. You supply the "customer PIN" (C-PIN) as an encrypted PIN block. The verb:

- Decrypts a PIN block
- Extracts a customer-selected or institution-assigned PIN (C-PIN)
- Generates an A-PIN from the input account number, PIN-generating key, etc.
- Computes an O-PIN from the C-PIN and the A-PIN; the O-PIN is returned in the clear.

Note: To generate an O-PIN from a *clear* C-PIN, see the Clear_PIN_Generate verb.

To use this verb, specify:

- An input PIN block encrypting key used to decrypt the PIN block
- A PIN-generating key used to calculate the A-PIN
- A PIN profile that describes the PIN block that contains the C-PIN
- When using the ISO-0 PIN block format, personal account number (PAN) data to be used in extracting the PIN
- The encrypted PIN block that contains the C-PIN
- A calculation method and optionally a PIN extraction method
- The length of the O-PIN offset. (The verb determines the length of the C-PIN from the length of the extracted PIN.)
- A decimalization table and account validation data
- A 16-byte variable for the O-PIN.

The verb does the following:

- Checks the control vector of the IPINENC key to ensure that the CPINGENA bit is 1.
- Decrypts the PIN block in ECB mode.
- Extracts the PIN. The verb uses the PIN extraction method specified with the *rule_array* parameter or the default extraction method for the PIN block format. The verb also uses the PIN_check_length variable. Depending on the PIN block format specified in the PIN profile, the verb also uses the pad digit specified in the PIN_profile variable or the PAN specified in the PAN_data variable.
- Verifies that the CPINGENA bit is 1 in the control vector for the PINGEN key.
- Calculates the A-PIN; the verb uses the specified calculation method, the data_array variable, and the PIN_check_length variable to calculate the PIN.

- Calculates the O-PIN.
- Returns the clear O-PIN in the variable identified by the *returned_result* parameter.

Restrictions

The software must include support for the PINS function set.

Format

CSNBCPA			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	Integer	exit_data_length bytes
inbound_PIN_encrypting_key_identifier	Input	String	64 bytes
PIN_generating_key_identifier	Input	String	64 bytes
input_PIN_profile	Input	String array	8 bytes * 3
PAN_data	Input	String	12 bytes
encrypted_PIN_block	Input	String	8 bytes
rule_array_count	Input	Integer	
rule_array	Input	String	8 bytes * rule_array_count
PIN_check_length	Input	Integer	
data_array	Input	String array	16 bytes * 3
returned_result	Output	String	16 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

inbound_PIN_encrypting_key_identifier
The *inbound_PIN_encrypting_key_identifier* parameter points to a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the key that decrypts the PIN block C-PIN. The control vector in the key token must specify the IPINENC key type and have the CPINGENA bit set to 1.

PIN_generating_key_identifier
The *PIN_generating_key_identifier* parameter points to a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the PIN-generation key and must contain a control vector that specifies the PINGEN key type and has the CPINGENA bit set to 1.

input_PIN_profile
The *input_PIN_profile* parameter points to a character array with: the PIN block format keyword, the format control (**NONE**), a pad digit (if needed); see “PIN Profile” on page 8-9.

PAN_data
The *PAN_data* parameter points to a 12-byte field of PAN data. If the PIN profile specifies the **ISO-0**, the verb uses the PAN data to recover the C-PIN from the PIN block.

Note: When using the ISO-0 format, use the 12 rightmost PAN digits, excluding the check digit.

encrypted_PIN_block
The *encrypted_PIN_block* parameter points the 8-byte, encrypted PIN block that contains the (customer-selected) C-PIN value.

rule_array_count
The *rule_array_count* parameter points to an integer for the number of rule array elements. The rule array count value must be 1 or 2. If you use the default extraction method for the PIN block format, the rule array count value is 1.

rule_array
The *rule_array* parameter points to an array of one or two 8-byte elements. Each keyword must be uppercase, left-justified, and padded on the right with space characters.

Element Number	Function of Keyword
1	PIN calculation method
2	PIN extraction method

The first element in the rule array must specify one of the keywords that indicate the PIN calculation method, as shown in Figure 8-6.

Figure 8-6. Clear_PIN_Generate_Alternate Rule_Array Keywords (First Element)

PIN Calculation Method Keyword	Meaning
IBM-PINO	This keyword specifies use of the IBM 3624 PIN Offset calculation method.
NL-PIN-1	This keyword specifies use of the Netherlands PIN-1 calculation method.
VISA-PVV	This keyword specifies that the VISA PVV calculation method is to be used.

The second element in the rule array must specify one of the keywords that indicate a PIN extraction method, as shown in Figure 8-7. For more information about extraction methods, see “PIN Extraction Methods.”

Notes:

1. In the table, the PIN block format keyword is the keyword that you specify in the *input_PIN_profile* parameter.
2. If the PIN block format allows you to choose the PIN extraction method, and if you specify a rule array count of 1, the keyword that is listed first in the following table is the default keyword.

Figure 8-7. Clear_PIN_Generate_Alternate Rule_Array Keywords (Second Element)

PIN Block Format Keyword	PIN Extraction Method Keyword	Meaning
3624	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN16, PADEXIST	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.
ISO-0	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-0 PIN block format.
ISO-1	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-1 PIN block format.

PIN_check_length

The *PIN_check_length* parameter points to an integer in the range from 4 to 16 for the number of digits of PIN information that the verb should check. The verb uses the *PIN_check_length* parameter if you specify the **IBM-PINO** keyword for the calculation method. Otherwise, ensure that this parameter points to a 4-byte variable in application data storage. The information in this variable will be ignored, but this variable must be declared.

Note: The PIN check length must be less than or equal to the PIN length.

The length of the PIN offset in the returned result will be determined by the value that the *PIN_check_length* parameter identifies. The security server shortens the PIN offset.

data_array

The *data_array* parameter points to three 16-byte character strings, which are equivalent to a single 48-byte string. The values in the data array depend on the PIN calculation method. Each element is not always used, but you must always declare a complete data array.

When using the **IBM-PINO** keyword, identify the following elements in the data array:

Element	Description
decimalization_table	This element contains the decimalization table of 16 characters (0 to 9) that are used to convert the hexadecimal digits (X'0' to X'F') of the enciphered validation data to decimal digits (X'0' to X'9').
validation_data	This element contains one to 16 characters of account data. The data must be left-justified and padded on the right with space characters.
reserved_3	The information in this element will be ignored, but the element must be declared.

When using the **NL-PIN-1** keyword, identify the following elements in the data array:

Element	Description
decimalization_table	This 16-character string should contain the characters 0, 1, ...9, A, ...F.
validation_data	This element contains one to 16 characters of account data. The data must be left-justified and padded on the right with space characters.
reserved_3	The information in this element will be ignored, but the element must be declared.

When using the **VISA-PVV** keyword, identify the following elements in the data array. For more information about transaction security data for the VISA PVV calculation method in the *IBM 4758 CCA Basic Services*, SC31-8609..

Element	Description
transaction_security_parameter	This element contains 16 numeric characters that include the following: • Eleven (rightmost) digits of PAN data • One digit of key index from 1 to 6 • Four space characters for padding.
reserved_2	The information in this element will be ignored, but the element must be declared.
reserved_3	The information in this element will be ignored, but the element must be declared.

returned_result

The *returned_result* parameter points to the clear O-PIN as a 16-byte character string. The result will be left-justified and padded with space characters.

The length of the PIN offset in the returned result will be determined by the value that the *PIN_check_length* parameter specifies.

Required Commands

The Clear_PIN_Generate_Alternate verb requires the following commands to be enabled in the hardware based on the keyword specified for the PIN calculation methods.

PIN Calculation Method	Command Offset	Command
IBM-PINO	X'00A4'	Generate Clear 3624 PIN Offset
NL-PIN-1	X'0231'	Generate Clear NL-PIN-1 Offset
VISA-PVV	X'00BB'	Generate Clear VISA PVV Alternate

Encrypted_PIN_Generate (CSNBEPG)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PINS

The Encrypted_PIN_Generate verb generates and formats a PIN and encrypts the PIN block. To generate the PIN, the verb uses one of the following PIN calculation methods:

- IBM German Bank Pool Instution PIN
- Interbank PIN.

To format the PIN, the verb uses one of the following PIN block formats:

- IBM 3624 format
- ISO-0 format (same as ANSI X9.8, VISA-1, and ECI-1 formats)
- ISO-1 format (same as the ECI-4 format)
- ISO-2

You can use the Encrypted_PIN_Generate verb to generate a PIN and create an encrypted PIN block for transmission or for later use in a PIN verification database.

Note: To generate a clear PIN, use the Clear_PIN_Generate verb.

To generate and format a PIN and encrypt the PIN block, specify the following:

- An internal key token or a key label of an internal key token record that contains the PIN-generating key with the *PIN_generating_key_identifier* parameter. The control vector in the key token must specify the PINGEN key type and have the EPINGEN bit set to 1.
- An internal key token or a key label of an internal key token record that contains the key to be used to encrypt the PIN block with the *outbound_PIN_encrypting_key_identifier* parameter. The control vector in the key token must specify the OPINENC key type and have the EPINGEN bit set to 1.
- One for the number of rule_array elements with the rule_array_count variable.
- The PIN calculation method with a keyword in the rule_array variable.
- Zero for the PIN_length variable as the supported methods have a pre-defined PIN length.
- A decimalization table and account validation data with the *data_array* parameter. For information about a decimalization table and calculation methods, see “PIN Calculation Methods” on page E-1. For information about the data array variable, see “Data_Array” on page 8-7.
- A PIN profile that specifies the format of the PIN block to be created, the level of format control, and any pad digit with the *output_PIN_profile* parameter. For more information about the PIN profile, see “PIN Block Formats” on page E-8.
- One of the following with the *PAN_data* parameter:

- When using the ISO-0 PIN block format, specify a PAN. For information about a personal account number (PAN), see “Personal Account Number (PAN)” on page 8-11.
- When using another PIN block format, specify a 12-byte variable in application data storage. The information in the variable will not be used, but the variable must be declared.

- With the sequence_number variable specify a 4-byte integer variable valued to 99999.
- An 8-byte variable for the encrypted PIN with the *encrypted_PIN_block* parameter.

The verb does the following:

- Verifies that the EPINGEN bit is 1 in the control vector for the PIN-generating key.
- Uses the specified PIN calculation method and account validation data to calculate the PIN.
- Optionally uses the specified PIN length to determine the length of the PIN.
- Formats the PIN into the specified PIN block format. The verb includes the clear PIN and, depending on the PIN block format, the pad digit, the PAN, and the sequence number. For a description of the formats, see “PIN Block Formats” on page E-8.
- Checks the control vector for the OPINENC key by verifying that the EPINGEN bit is 1.
- Encrypts the PIN block in ECB mode according to the format-control keyword specified in the PIN profile.

Restrictions

The software must include the support for the PINS function set.

Format

CSNBEPG

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
PIN_generating_key_identifier	Input	String	64 bytes
outbound_PIN_encrypting_key_identifier	Input	String	64 bytes
rule_array_count	Input	Integer	
rule_array	Input	String	8 bytes
PIN_length	Input	Integer	
data_array	Input	String	16 bytes * 3
PIN_profile	Input	String array	8 bytes * 3
PAN_data	Input	String	**??** bytes
sequence_number	Input	Integer	
encrypted_PIN_block	Output	String	8 bytes

Parameters

For the definitions of the first four parameters, see "Parameters Common to All Verbs."

PIN_generating_key_identifier
The *PIN_generating_key_identifier* parameter points to the place in application data storage that contains a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the PIN-generating key and must contain a control vector that specifies a PINGEN key type and has the EPINGEN bit set to 1.

outbound_PIN_encrypting_key_identifier
The *outbound_PIN_encrypting_key_identifier* parameter points to the place in application data storage that contains a 64-byte internal key token or a key label of an internal key token record in key storage. 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 bit set to 1.

rule_array_count
The *rule_array_count* parameter points to the place in application data storage that contains an integer for the number of the rule array elements. This value must be 1.

rule_array
The *rule_array* parameter points to the place in application data storage that contains a keyword that defines the calculation method to be used.

The keywords in the rule array must be 8 bytes in length, uppercase, left-justified, and padded on the right with space characters, as shown in Figure 8-8.

Figure 8-8. Encrypted_PIN_Generate Rule_Array Keywords

Keyword	Meaning
GBP-PIN	This keyword specifies the IBM German Bank Pool Instution PIN calculation method to be used to generate a PIN.
INBK-PIN	This keyword specifies the Interbank PIN calculation method to be used to generate a PIN.

PIN_length
The *PIN_length* parameter points to the place in application data storage that contains an integer to define the PIN length for those PIN calculation methods with variable length PINs, otherwise the variable should be valued to zero.

data_array
The *data_array* parameter points to the place in application data storage that contains 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. The verb converts the space characters to zeros.

When using the **INBK-PIN** keyword, identify the following elements in the data array. For more information about these elements and transaction security data for the Interbank calculation method, see "Data_Array" on page 8-7.

Element	Description
transaction_security_parameter	This element contains 16 numeric characters that include the following: • Eleven (rightmost) digits of PAN data • A constant of 6 • A 1-digit key index selector from 1 to 6 • Three numeric characters of validation data.
reserved_2	The information in this element will be ignored, but the element must be declared.
reserved_3	The information in this element will be ignored, but the element must be declared.

PIN_profile
The *PIN_profile* parameter points to the place in application data storage that contains the PIN profile including the PIN block format, see "PIN Profile" on page 8-9.

PAN_data
The *PAN_data* parameter points to the place in application data storage that contains 12 digits of Personal Account Number (PAN) data. The verb uses this parameter if the PIN profile specifies the **ISO-0** for the PIN block format. Otherwise, ensure that this parameter points to a 4-byte variable in application data storage. The information in this variable is ignored, but this variable must be declared.

Note: When using the **ISO-0** keyword, use the 12 rightmost digits of the PAN data, excluding the check digit.

sequence_number
The *sequence_number* parameter points to the place in application data storage that contains the sequence number used by certain PIN block formats. Ensure that this parameter points to a 4-byte variable in application data storage.

encrypted_PIN_nlock
The *encrypted_PIN_block* parameter points to the place in application data storage where the verb will return the 8-byte encrypted PIN.

Required Commands

The Encrypted_PIN_Generate verb requires the following commands to be enabled in the cryptographic engine based on the keyword specified for the PIN calculation methods.

PIN Calculation Method	Command Offset	Command
GBP-PIN	X'00B1'	Generate Formatted and Encrypted Clear German Bank Pool PIN
INBK-PIN	X'00B2'	Generate Formatted and Encrypted Interbank PIN

Encrypted_PIN_Translate (CSNBPTR)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PINS

The Encrypted_PIN_Translate verb can re-encipher a PIN block, and optionally format a PIN into a different PIN block format.

This verb can be used to convert the encryption and format of a PIN block in an interchange network, or to have the PIN block conform to the format and encryption key used in a PIN verification database. You can also use this verb to change the PAN and/or the pad digit.

The input and output PIN blocks can be in the following formats:

- IBM 3624
- ISO-0 (equivalent to ANSI X9.8, VISA-1, and ECI-1 formats).
- ISO-1 (same as the ECI-4 format)
- ISO-2

The verb can operate in one of two modes based on a keyword in the rule array:

- Translate mode. You specify this mode with the **TRANSLAT** keyword. In this mode, the verb re-encrypts a PIN block from encryption under one PIN-block encrypting key to encryption under another PIN-block encrypting key.
- Reformat mode. You specify this mode with the **REFORMAT** keyword. In this mode, the verb decrypts the input PIN block, extracts the PIN, formats the PIN into the output PIN block, and encrypts the output PIN block.

To use this verb, specify:

- The mode of operation, translation or translation-and-reformatting with a rule array keyword
- Optionally specify the method of PIN extraction from the input PIN block with another rule array keyword
- Input and output PIN-block encrypting keys
- Input and output PIN profiles, see “PIN Profile” on page 8-9
- Input and output PAN data as required by the selected PIN block format
- An output PIN block sequence number as required by the selected PIN block format, or specify a value of 99999.

The verb does the following:

- Decrypts the input PIN block in ECB mode using a key with an IPINENC control vector; the control vector must have the TRANSLAT bit set to 1, and if reformatting is selected, the REFORMAT bit must also be set to 1.
- In reformat mode these additional steps are performed:

  Extracts the PIN from the specified PIN block format using either the default extraction method associated with the declared PIN block format

or the method specified by a the rule array keyword. As required by the PIN block format, PAN data will be used in the extraction process.

– Formats the extracted-PIN into the format declared for the output PIN block. As required by the PIN block format, the verb incorporates PAN data, sequence number, and pad character information in formatting the output.

- The PIN block is encrypted using the outbound key provided the OPINENC control vector has the TRANSLAT bit set to 1, and if reformatting is selected, the REFORMAT bit also set to 1.

Restrictions

The software must include support for the PINS function set.

Format

CSNBPTR			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
input_PIN_encrypting_key_identifier	Input	String	64 bytes
output_PIN_encrypting_key_identifier	Input	String	64 bytes
input_PIN_profile	Input	String array	8 bytes * 3
input_PAN_data	Input	String	12 bytes
input_PIN_block	Input	String	8 bytes
rule_array_count	Input	Integer	
rule_array	Input	String	8 bytes * rule_array_count
output_PIN_profile	Input	String array	8 bytes * 3
output_PAN_data	Input	String	12 bytes
sequence_number	Input	Integer	
output_PIN_block	Output	String	8 bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

input_PIN_encrypting_key_identifier

The *input_PIN_encrypting_key_identifier* parameter points to a 64-byte internal key token or a key label of an internal key token record in key storage.

The internal key token must contain the input PIN-block encrypting key to be used to decrypt the input PIN block. The control vector in the key token must specify the IPINENC key type with the TRANSLAT bit set to 1. If the **REFORMAT** keyword is used, both the TRANSLAT bit and the REFORMAT bit must be 1 in the control vector.

output_PIN_encrypting_kKey_identifier

The *output_PIN_encrypting_key_identifier* parameter points to the place in application data storage that contains a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the output PIN-block encrypting key to be used to encrypt the output PIN block. The control vector in the key token must specify the IPINENC key type with the TRANSLAT bit set to 1. If the **REFORMAT** keyword is used, both the TRANSLAT bit and the REFORMAT bit must be 1 in the control vector.

input_PIN_profile
The *input_PIN_profile* parameter points to three 8-byte character strings, which are equivalent to a 24-byte string. For more information about a PIN profile, see "PIN Profile" on page 8-9. Be sure to specify the second element as "NONE."

input_PAN_data
The *input_PAN_data* parameter points to a 12-byte field of PAN data. The verb uses this data to recover the PIN from the PIN block if you specify the **REFORMAT** keyword and the input PIN profile specifies the **ISO-0** keyword for the PIN block format.

Note: When using the ISO-0 format, use the 12 rightmost digits of PAN, excluding the check digit.

input_PIN_block
The *PIN_block_in* parameter points to the 8-byte input encrypted PIN block.

rule_array_count
The *rule_array_count* parameter points to an integer for the number of rule array elements.

rule_array
The *rule_array* parameter points to an array of one or two 8-byte elements each holding a keyword. Each keyword must be uppercase, left-justified, and padded on the right with space characters; in this verb the order of the keywords is important.

Element Number	Function of Keyword
1	Mode
2	PIN extraction method

The first element in the rule array must specify the mode, as shown in Figure 8-9.

Figure 8-9. Encrypted_PIN_Translate Rule_Array Keywords (First Element)

Mode Keyword	Meaning
TRANSLAT	This keyword specifies that only the PIN encrypting key is to be changed. The PIN block format and the contents of the PIN block are not changed. The format control specifications in the input PIN profile and in the output PIN profile are used. **Note:** The PIN block can have a PIN of any length (in the range from 4 to 16 bytes).
REFORMAT	This keyword specifies that one or more of the following are to be changed: the PIN block format, the contents of the PIN block, or the PIN encrypting key.

If you use the reformat mode, the second element in the rule array must specify one of the keywords that indicate a PIN extraction method, as shown

in Figure 8-10. For more information about extraction methods, see “PIN Extraction Methods.”

Notes:

1. In the table, the PIN block format keyword is the keyword that you specify in the *input_PIN_profile* parameter or in the *output_PIN_profile* parameter.
2. If the PIN block format allows you to choose the PIN extraction method, and if you specify a rule array count value of 1, the keyword that is listed first in the following table is the default keyword.

Figure 8-10. Encrypted_PIN_Translate Rule_Array Keywords (Second Element)

PIN Block Format Keyword	PIN Extraction Method Keyword	Meaning
3624	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN16, PADEXIST	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 3624 PIN block format.
ISO-0	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-0 PIN block format.
ISO-1	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-1 PIN block format.
ISO-2	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-2 PIN block format.

output_PIN_profile
The *output_PIN_profile* parameter points to three 8-byte character strings, which are equivalent to a 24-byte string; see “PIN Profile” on page 8-9. Be sure to specify the second element as “NONE.”

output_PAN_data
The *output_PAN_data* parameter points to a 12-byte field of PAN data. If you specify the **REFORMAT** keyword, and if the output PIN profile specifies the **ISO-0** keyword for the PIN block format, the verb uses this data to format the output PIN block. In any case, ensure that this parameter points to a 12-byte variable in application data storage.

Note: When using the ISO-0 format, use the 12 rightmost digits of PAN, excluding the check digit.

sequence_number
The *sequence_number* parameter points to the sequence number integer variable. Ensure that this parameter points to an integer variable valued to 99999 in application data storage.

output_PIN_block
The *PIN_block_out* parameter points to the re-encrypted output PIN block.

Required Commands

The Encrypted_PIN_Translate verb requires the commands shown in Figure 8-11 to be enabled in the active hardware based on the keyword specified for translation or reformatting and the format control in the PIN profile. You should enable only those commands that are required.

Figure 8-11. Encrypted_PIN_Translate Required Hardware Commands

TRANSLAT or REFORMAT Keyword	Input Profile Format Control Keyword	Output Profile Format Control Keyword	Command Offset	Command
TRANSLAT	NONE	NONE	X'00B3'	Translate PIN with No Format-Control to No Format-Control
REFORMAT	NONE	NONE	X'00B7'	Reformat PIN with No Format-Control to No Format-Control

Encrypted_PIN_Verify (CSNBPVR)

Platform/ Product	OS/2	AIX	NT	Verb Subset
IBM-4758	X	X	X	PINS

The Encrypted_PIN_Verify verb extracts a trial PIN (T-PIN) from an encrypted PIN block and verifies this value by comparing it to an A-PIN calculated by using the specified PIN calculation method. Certain PIN calculation methods modify the value of the A-PIN with the clear O-PIN (offset) value prior to the comparison.

The verb can extract a T-PIN from one of the following PIN block formats:

- IBM 3624
- ISO-0 (equivalent to ANSI X9.8, VISA-1, and ECI-1 formats).
- ISO-1 (same as the ECI-4 format)
- ISO-2

To calculate the PIN, the verb can use one of the following PIN calculation methods:

- IBM 3624 PIN
- IBM 3624 PIN Offset
- NL-PIN-1.
- IBM German Bank Pool Instution PIN
- VISA PVV
- Interbank PIN.

The input PIN block is deciphered by an IPINENC key.

To use the verb, specify the following:

- An input PIN-block encrypting key to decrypt the input PIN block
- A PIN-verifying key to be used to calculate the PIN
- A PIN profile for the input PIN block. The profile specifies the format of the PIN block, any format control, and any pad digit; see "PIN Profile" on page 8-9.
- When using the ISO-0 block format, a PAN to be used in extracting the PIN; see "Personal Account Number (PAN)" on page 8-11.
- The PIN block that contains the PIN to be verified
- A rule array keyword to select the PIN calculation method, and optionally a PIN extraction method keyword
- The length of the PIN
- A decimalization table, account validation data, and for certain calculation methods (e.g. IBM-PINO and NL-PIN-1) with the *data_array* parameter. You must also supply the offset data (O-PIN) in the third element for certain calculation methods (e.g. IBM-PINO, NL-PIN-1).

The verb does the following:

- Checks the control vector for the IPINENC key to ensure that the EPINVER bit is 1.
- Decrypts the PIN block in ECB mode.
- Extracts the T-PIN according to the format-control keyword in the PIN profile. The verb uses the PIN extraction method specified with the rule_array variable or the default extraction method for the PIN block format. Depending on the PIN block format, the verb also uses the pad digit specified in the input_PIN_profile variable and/or the PAN specified in the PAN_data variable.
- For a PINVER key, verifies that the EPINVER bit is 1 in the control vector. For a PINGEN key, verifies that both the EPINVER bit and bit 22 are 1 in the control vector.
- Calculates the A-PIN.
- For methods that employ an offset, modifies the A-PIN value with the O-PIN value entered in the third element of the data_array variable.
- Compares the extracted T-PIN with the (modified) A-PIN and reports the results in the return_code variable.

Restrictions

The software must include support for the PINS function set.

Format

CSNBPVR			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	Input	String	exit_data_length bytes
PIN_encrypting_key_identifier	Input	String	64 bytes
PIN_verifying_key_identifier	Input	String	64 bytes
PIN_profile	Input	String array	8 bytes * 3
PAN_data	Input	String	12 bytes
encrypted_PIN_block	Input	String	8 bytes
rule_array_count	Input	Integer	
rule_array	Input	String	8 bytes * rule_array_count
PIN_check_length	Input	Integer	
data_array	Input	String array	8 Bytes * 3

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

PIN_encrypting_key_identifier

The *PIN_encrypting_key_identifier* parameter points to a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token must contain the input PIN-block encrypting key to be used to decrypt the encrypted PIN block. The control vector in the internal key token must specify an IPINENC key type and have the EPINVER bit set to 1.

PIN_verifying_key_identifier
The *PIN_verifying_key_identifier* parameter points to a 64-byte internal key token or a key label of an internal key token record in key storage. The internal key token contains the key that verifies the PIN. The control vector in the internal key token must specify a PINVER or PINGEN key type. For a PINVER key, the EPINVER bit must be 1. For a PINGEN key, both the EPINVER bit and bit 22 must be 1.

PIN_profile
The *PIN_profile* parameter points to three 8-byte character strings, which are equivalent to a 24-byte string. These character strings contain the following information about a formatted PIN:

- PIN block format
- Format control
- Pad digit (if needed).

For more information about this parameter, see "PIN Profile" on page 8-9.

PAN_data
The *PAN_data* parameter specifies an address that points to the place in application data storage that contains a 12-byte field of PAN data. The verb uses the PAN data to recover the PIN from the PIN block if the PIN profile specifies the **ISO-0** keyword for the PIN block format. Otherwise, ensure that this parameter points to a 12-byte variable in application data storage.

Note: When using the ISO-0 format, use the 12 rightmost PAN digits, excluding the check digit.

encrypted_PIN_block
The *encrypted_PIN_block* parameter specifies an address that points to the place in application data storage that contains the 8-byte encrypted PIN block.

rule_array_count
The *rule_array_count* parameter points to an integer for the number of rule array elements. The rule array count value must be 1 or 2. If you use the default extraction method for the PIN block format, the rule array count value is 1.

rule_array
The *rule_array* parameter points to an array of 8-byte elements that contain a keyword. Each keyword must be uppercase, left-justified, and padded on the right with space characters.

Element Number	Function of Keyword
1	PIN calculation method
2	PIN extraction method

The first element in the rule array must specify one of the keywords that indicate the PIN calculation method, as shown in Figure 8-12.

Figure 8-12. Encrypted_PIN_Verify Rule_Array Keywords (First Element)

PIN Calculation Method Keyword	Meaning
IBM-PIN	This keyword specifies that the IBM 3624 PIN calculation method is to be used.
IBM-PINO	This keyword specifies that the IBM 3624 PIN Offset calculation method is to be used.
GBP-PIN	This keyword specifies that the IBM German Bank Pool Institution PIN calculation method is to be used.
VISA-PVV	This keyword specifies that the VISA PVV calculation method is to be used.
INBK-PIN	This keyword specifies that the Interbank calculation method is to be used.

The second element in the rule array must specify one of the keywords that indicate a PIN extraction method, as shown in Figure 8-13 on page 8-37.

Notes:

1. In the table, the PIN block format keyword is the keyword that you specify in the *input_PIN_profile* parameter.
2. If the PIN block format allows you to choose the PIN extraction method, and if you specify a rule array count value of 1, the keyword that is listed first in the following table is the default keyword.

Figure 8-13. Encrypted_PIN_Verify Rule_Array Keywords (Second Element)

PIN Block Format Keyword	PIN Extraction Method Keyword	Meaning
3624	PADDIGIT, HEXDIGIT, PINLEN04 to PINLEN16, PADEXIST	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 3624 PIN block format.
ISO-0	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-0 PIN block format.
ISO-1	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-1 PIN block format.
ISO-2	PINBLOCK	This keyword specifies the default PIN extraction method for an ISO-2 PIN block format.

PIN_check_length

The *PIN_check_length* parameter points to an integer in the range from 4 to 16 for the number of digits of PIN information that the verb should verify.

The verb uses the value in the variable if you specify the **IBM-PIN** or **IBM-PINO** keyword for the calculation method. The specified number of

digits is selected from the low order (right side) of the PIN. Ensure that this parameter always points to an integer variable in application data storage.

Note: The PIN check length must be less than or equal to the PIN length.

data_array
The *data_array* parameter points to 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.

When using the **IBM-PIN**, **IBM-PINO** or **GBP-PIN** keyword, identify the following elements in the data array.

Element	Description
decimalization_table	This element contains the decimalization table of 16 characters (0 to 9) that are used to convert the hexadecimal digits (X'0' to X'F') of the encrypted validation data to decimal digits (X'0' to X'9').
validation_data	This element contains one to 16 characters of account data. The data must be left-justified and padded on the right with space characters.
offset data	When using the **IBM-PINO** keyword, this element contains the offset data which must be left-justified and padded with space characters. The PIN length specifies the number of digits that are processed for the IBM-PINO PIN calculation method. When using the **IBM-PIN** or **GBP-PIN** keyword, this element is ignored, but must be declared.

When using the **VISA-PVV** keyword, identify the following elements in the data array. For more information about these elements, and transaction security data for the VISA PVV calculation method, see "VISA PIN Validation Value (PVV) Calculation Method" on page E-6.

Element	Description
transaction_security_parameter	This element contains 16 characters that include the following: • Eleven (rightmost) digits of PAN data • One digit of key index from 1 to 6 • Four space characters.
PVV (O-PIN)	This element contains 4 numeric characters, which are the referenced PVV value. This value is followed by 12 space characters.
reserved_3	The information in this element will be ignored, but the element must be declared.

When using the **INBK-PIN** keyword, identify the following elements in the data array. For more information about these elements and transaction security data for the Interbank calculation method, see "Interbank PIN Calculation Method" on page E-7.

Element	Description
transaction_security_parameter	This element contains 16 numeric characters that include the following: • Eleven (rightmost) digits of PAN data • A constant of 6 • A 1-digit key index selector from 1 to 6 • Three numeric characters of validation data.
reserved_2	The information in this element will be ignored, but the element must be declared.
reserved_3	The information in this element will be ignored, but the element must be declared.

Required Commands

The Encrypted_PIN_Verify verb requires the following commands to be enabled in the hardware, based on the keyword specified for the PIN calculation methods.

PIN Calculation Method	Command Offset	Command
IBM-PIN IBM-PINO	X'00AB'	Verify Encrypted 3624 PIN
GBP-PIN	X'00AC'	Verify Encrypted German Bank Pool PIN
VISA-PVV	X'00AD'	Verify Encrypted VISA PVV
INBK-PIN	X'00AE'	Verify Encrypted Interbank PIN
NL-PIN-1	X'0232'	Verify Encrypted NL-PIN-1

SET_Block_Compose (CSNDSBC)

Platform/ Product	**OS/2**	**AIX**	**NT**	Verb Subset
IBM-4758	**X**	**X**	**X**	SET

The SET_Block_Compose verb creates a SET-protocol RSA-OAEP block and DES encrypts the data block in support of the SET protocols. Optionally the verb will compute the SHA-1 hash of the supplied data block and include this in the OAEP block.

Restrictions

The data block length variable is restricted to 32 mega-bytes.

The *DES_key_block_length* parameter must point to an integer valued to zero. The *DES_key_block* parameter should be a null address pointer, or point to an unused 64-byte application variable.

The *chaining_key_vector* parameter must be a null address pointer, or point to an unused 18-byte application variable. This parameter is included to support a possible future extension to enable segmented data encryption.

Note: The API for this verb has been modified from that originally published in August, 1997.

Format

CSNDSBC			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	*exit_data_length* bytes
rule_array_count	Input	Integer	
rule_array	Input	String	*rule_array_count* * 8 byte
block_contents_identifier	Input	String	1 byte
XData_string_length	Input	Integer	
XData_string	Input	String	XData_string_length bytes
data_to_encrypt_length	In/Output	Integer	
data_to_encrypt	Input	String	data_to_encrypt_length bytes
data_to_hash_length	Input	Integer	
data_to_hash	Input	String	data_to_hash_length bytes
initialization__vector	Input	String	8 bytes
RSA_public_key_identifier_length	Input	Integer	
RSA_public_key_identifier	Input	String	RSA_public_key_identifier_length bytes
DES_key_block_length	In/Output	Integer	
DES_key_block	In/Output	String	DES_key_block_length bytes
RSA-OAEP_block_length	In/Output	Integer	
RSA-OAEP_block	In/Output	String	RSA-OAEP_block_length bytes
chaining_vector	In/Output	String	18 bytes
DES_encrypted_block	Output	String	data_block_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see “Parameters Common to All Verbs” on page 1-10.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable. The value of the *rule_array_count* must be one for this verb.

rule_array

The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Block Type (Required)	
SET1.00	Specifies structure of the RSA-OAEP encrypted block is defined by the SET protocol.

Block_contents_identifier

The *block_contents_identifier* parameter is a pointer to a one-byte string variable containing a binary value that will be copied into the Block Contents (BC) field of the SET DB data block. The BC field indicates what data is carried in the Actual Data Block, ADB, and the format of any extra data (*XData_string*).

XData_string_length

The *XData_string_length* parameter is a pointer to an integer variable containing the length (in bytes) of the *XData_string*. The maximum length is 94 bytes.

XData_string

The *XData_string* parameter is a pointer to the string containing extra-encrypted data within the OAEP-processed and RSA-encrypted block. If *Xdata_string_length* is zero, this parameter is ignored, but it must still be specified.

data_to_encrypt_length

The *data_to_encrypt_length* parameter is a pointer to an integer variable containing the length (in bytes) of the data block that is to be encrypted. The maximum length is the same limit as on the Encipher service. On output, and if the field is of sufficient length, the variable is updated with the actual length of the DES-encrypted data block.

data_to_encrypt

The *data_to_encrypt* parameter is a pointer to a string variable containing the data to be DES-encrypted with a single-use 64-bit DES key (generated by this service). The data will first be padded by this service according to the PKCS #5 padding rule before encryption.

data_to_hash_length

The *data_to_hash_length* parameter is a pointer to an integer variable containing the length (in bytes) of the data block that is to be hashed.

The hash is an optional part of the OAEP block. If the *data_to_hash_length* is zero, no hash will be included in the OAEP block. If the length is not zero, a SHA-1 hash of the *data_to_hash* will be included in the OAEP block.

data_to_hash
The *data_to_hash* parameter is a pointer to a string variable containing the data that is to be hashed and included in the OAEP block.

No hash is computed or inserted into the OAEP block if the data_to_hash_length is zero.

initialization_vector
The *initialization_vector* parameter is a pointer to an eight-byte string variable containing the initialization_vector the verb uses with the input data.

RSA_public_key_identifier_length
The *RSA_public_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the variable that contains the key token or the key label of the PKA96 RSA public key used to encipher the OAEP block. The maximum size that should be specified is 2500 bytes.

RSA_public_key_identifier
The *RSA_public_key_identifier* parameter is a pointer to a string variable containing the PKA96 RSA key token with the RSA public key used to perform the RSA encryption of the OAEP block.

DES_key_block_length
The *DES_key_block_length* parameter is a pointer to an integer variable containing the length (in bytes) of the variable identified by the *DES_key_block* parameter. The variable must be set to zero.

DES_key_block
The *DES_key_block* parameter must be a null pointer, or a pointer to an unused 64-byte application variable.

RSA-OAEP_block_length
The *RSA-OAEP_block_length* parameter is a pointer to an integer variable containing the length (in bytes) of the RSA-OAEP block variable used to hold the RSA-OAEP block. The length must be at least 128 bytes. On output, and if the field is of sufficient length, the variable is updated with the actual length of the RSA-OAEP block.

RSA-OAEP_block
The *RSA-OAEP_block* parameter is a pointer to a string variable to contain the RSA-OAEP block.

chaining_vector
The *chaining_vector* parameter is a pointer to an 18-byte string variable that the security server uses as a work area to carry segmented data between calls. The parameter must contain a null pointer or a pointer to an unused 18-byte application variable.

DES_enciphered_data_block
The *DES_enciphered_data_block* parameter is a pointer to a string variable to receive the DES-encrypted data block (clear text was identified with the *data_to_encrypt* variable). The starting address must not fall inside the *data_to_encrypt* area.

Required Commands

The SET_Block_Compose verb requires the x'010B' command to be enabled in the hardware.

SET_Block_Decompose (CSNDSBD)

Platform/ Product	**OS/2**	**AIX**	**NT**	Verb Subset
IBM-4758	**X**	**X**	**X**	SET

The SET_Block_Decompose verb decomposes the RSA-OAEP block and DES decrypts the data block in support of the SET protocols.

Restrictions

The maximum data block that can be supplied for DES decryption is the limit on the Decipher service.

The *DES_key_block_length* parameter must point to an integer valued to zero. The *DES_key_block* parameter should be a null address pointer, or point to an unused 64-byte application variable.

| The *chaining_Key_vector* parameter must be a null address pointer, or point to
| an unused 18-byte application variable. This parameter is included to support a
| possible future extension to enable segmented data encryption.

| **Note:** The API for this verb has been modified from that originally published in
| August, 1997.

Format

CSNDSBD			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	Input	Integer	
exit_data	In/Output	String	*exit_data_length* bytes
rule_array_count	Input	Integer	
rule_array	Input	String	8 bytes
RSA-OAEP_block_length	Input	Integer	
RSA-OAEP_block	Input	String	RSA-OAEP_block_length bytes
DES_encrypted_data_block_length	In/Output	Integer	
DES_encrypted_data_block	Input	String	DES_encrypted_data_block_length bytes
| *initialization__vector*	Input	String	8 bytes
RSA_private_key_identifier_length	Input	Integer	
RSA_private_key_identifier	Input	String	RSA_private_key_identifier_length bytes
DES_key_block_length	In/Output	Integer	
DES_key_block	In/Output	String	DES_key_block_length bytes
block_contents_identifier	Output	String	1 byte
XData_string_length	In/Output	Integer	
XData_string	Output	String	XData_string_length bytes
| *chaining__vector*	In/Output	String	18 bytes
data_block	Output	String	DES_encrypted_data_block_length bytes
| *data_to_hash_length*	In/Output	Integer	
| *data_to_hash*	Output	String	hash_block_length bytes

Parameters

For the definitions of the *return_code*, *reason_code*, *exit_data_length*, and *exit_data* parameters, see "Parameters Common to All Verbs" on page 1-10.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer containing the number of elements in the rule_array variable. The value of the *rule_array_count* must be one for this verb.

rule_array
The *rule_array* parameter is a pointer to an array of keywords. The keywords are eight bytes in length, and must be uppercase, left-justified, and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Block Type (Required)	
SET1.00	Specifies structure of the RSA-OAEP encrypted block is defined by SET protocol.

RSA-OAEP_block_length
The *RSA-OAEP_block_length* parameter is a pointer to an integer that is the length in bytes of the RSA-OAEP block field. This length must be 128 bytes.

RSA-OAEP_block
The *RSA-OAEP_block* parameter is a pointer to the string that contains RSA-OAEP block. When the OAEP is returned, it is left justified within the RSA-OAEP block field.

DES_encrypted_data_block_length
The *DES_encrypted_data_block_length* parameter is a pointer to an integer variable containing the length (in bytes) of the *DES_encrypted_data_block*. On output, the variable is updated with the actual length of the decrypted data with padding removed.

DES_encrypted_data_block
The *DES_encrypted_data_block* parameter is a pointer to a string variable containing the DES-encrypted data block.

| **initialization_vector**
| The *initialization_vector* parameter is a pointer to an eight-byte string
| variable containing the initialization_vector the verb uses with the input data.

RSA_private_key_identifier_length
The *RSA_private_key_identifier_length* parameter is a pointer to an integer variable containing the length (in bytes) of the variable that contains the key token or the key label of the PKA96 RSA private key used to decipher the OAEP block. The maximum size that should be specified is 2500 bytes.

RSA_private_key_identifier
The *RSA_private_key_identifier* parameter is a pointer to a string variable containing the PKA96 RSA key token with the RSA private key used to perform the RSA decryption of the OAEP block.

DES_key_block_length
The *DES_key_block_length* parameter is a pointer to an integer variable containing the length (in bytes) of the field DES key block. The length must be 64 bytes.

DES_key_block
The *DES_key_block* parameter is a pointer to a string variable to contain the generated internal token of a DES DATA key. Your application must not change the data in this string.

block_contents_identifier
The *block_contents_identifier* parameter is a pointer to a one-byte string variable to contain the the Block Contents (BC) field of the SET DB data block. The BC field indicates what data is carried in the Actual Data Block, ADB, and the format of any extra data (XData string).

XData_string_length
The *XData_string_length* parameter is a pointer to an integer variable containing the length (in bytes) of the *XData_string* field. The minimum length is 94 bytes. On output, and if the field is of sufficient length, the variable is updated with the actual length of the *XData_string* returned.

XData_string
The *XData_string* parameter is a pointer to the string variable containing the extra-encrypted data within the OAEP-processed and RSA-decrypted block.

chaining_vector
The *chaining_vector* parameter is a pointer to an 18-byte string variable that the security server uses as a work area to carry segmented data between calls. The parameter must contain a null pointer or a pointer to an unused 18-byte application variable.

data_block
The *data_block* parameter is a pointer to a string variable to contain the decrypted DES encrypted data block. The starting address must not fall inside the DES encrypted data block area. Padding characters are removed.

data_to_hash_length
The *hash_block_length* parameter is a pointer to an integer variable that is set to the length of the SHA-1 hash returned in the *hash_block* parameter.

On input, this parameter must be set to the size of the buffer pointed to by parameter *hash_block*. An error will be returned if the buffer is not large enough to hold the 20 byte SHA-1 hash.

On output, this field is updated to reflect the length of the hash data returned in *hash_block*, either 0 or 20 bytes.

data_to_hash
The *hash_block* parameter is a pointer to a string variable which will receive the SHA-1 hash extracted from the OAEP block.

Required Commands

The SET_Block_Decompose verb requires the x'010C' command to be enabled in the hardware.

Appendix A. Return Codes and Reason Codes

This appendix describes the return codes and the reason codes that a verb uses to report the results of processing.

Each return code is associated with a reason code that supplies details about the result of verb processing. A successful result can include return code 0 and reason code 0 or another combination of a return code and a reason code. Generally, you should be able to base your application program design on the return codes; the reason codes amplify the meaning supplied by the return codes.

A verb supplies a return code and a reason code in the *return_code* parameter and in the *reason_code* parameter.

Return Codes

A return code provides a summary of the results of verb processing. A return code can have the values shown in Figure A-1.

Figure A-1. Return Code Values

Hex Value	Decimal Value	Meaning
00	00	This return code indicates a normal completion of verb processing. To provide additional information, a few nonzero reason codes are associated with this return code.
04	04	This return code is a warning that indicates that the verb completed processing; however, an unusual event occurred. The event is most likely related to a problem created by the user, or it is a normal occurrence based on the data supplied to the verb.
08	08	This return code indicates that the verb stopped processing. Either an error occurred in the application program or a possible recoverable error occurred in a Transaction Security System product.
0C	12	This return code indicates that the verb stopped processing. Either a Transaction Security System product is not available or a processing error occurred in a Transaction Security System product. The reason is most likely related to a problem in the setup of the hardware or in the configuration of the software.
10	16	This return code indicates that the verb stopped processing. A processing error occurred in a Transaction Security System product. If these errors persist, a repair of the Transaction Security System hardware or a correction to the Transaction Security System software may be required.

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.

Figure A-2 on page A-2 shows the reason codes, listed in numeric sequence and grouped by their corresponding return code. The return codes appear in decimal form, and the reason codes appear in decimal and hexadecimal (hex) form.

© Copyright IBM Corp. 1997-98

Return Code 0

Figure A-2. Reason Codes for Return Code 0

Return Code Dec	Reason Code Dec (Hex)	Meaning
0	000 (000)	The verb completed processing successfully.
0	002 (002)	One or more bytes of a key do not have odd parity.
0	008 (008)	No value is present to be processed.
0	151 (097)	The key token supplies the MAC length or MACLEN4 is the default for key tokens that contain MAC or MACVER keys.
0	1000 (3E8)	The key value in an internal key token was dynamically re-enciphered.
0	701 (2BD)	A new master key value was found to have duplicate thirds.
0	702 (2BE)	A provided master key part did not have odd parity.
0	10000 (2710)	The verb dynamically updated and returned one or more keys that the application program provided.
0	10001 (2711)	A key encrypted under the old master key was used.

Return Code 4

Figure A-3 (Page 1 of 2). Reason Codes for Return Code 4		
Return Code Dec	**Reason Code Dec (Hex)**	**Meaning**
4	001 (001)	The verification test failed.
4	013 (00D)	The key token has an initialization vector, and the *initialization_vector* parameter value is nonzero. The verb uses the value in the key token.
4	016 (010)	The rule array and the rule array count are too small to contain the complete result.
4	017 (011)	The requested ID is not present in any profile in the specified cryptographic hardware component.
4	018 (012)	The time that was specified by the time-out value expired.
4	019 (013)	The financial PIN in a PIN block is not verified, or the password in a Cryptographic Adapter or the PIN in a Personal Security Card is not verified.
4	020 (014)	If you provided text with an odd length for the Character/Nibble_Translate verb, the right nibble of the last byte is padded with X'00'.
4	021 (015)	The key is marked inactive in flag byte 1 of the key token.
4	052 (034)	A request for END-EX is issued while the server is already in non-exclusive control mode.
4	053 (035)	A request for BEGIN-EX is issued while the server is already in exclusive control mode.
4	123 (07B)	A key-encrypting key count value is zero, and the key notarization or offset process is requested.
4	158 (09E)	The Key_Token_Change or Key_Record_Delete verb did not process any records.
4	166 (0A6)	The control vector is not valid because of parity bits, anti-variant bits, or inconsistent KEK bits, or because bits 59 to 62 are not zero.
4	179 (0B3)	The control-vector keywords that are in the rule array are ignored.
4	182 (0B6)	The actual size of the allocated Personal Security card block is not a multiple of 8 bytes.
4	260 (104)	The plaintext is not a multiple of eight bytes. The security server padded the plaintext to a multiple of 8 bytes for the SNA-SLE ciphering method.
4	282 (11A)	The coprocessor intrusion latch is set.
4	283 (11B)	The coprocessor battery is low.
4	284 (11C)	The requested command completed, but the device is in the initialization state.
4	285 (11D)	The Personal Security card detected an EEPROM checksum error while reading a data block. Data was returned, but some part of the data is incorrect.
4	286 (11E)	The signature verification overlay file was not found.
4	287 (11F)	The PIN block format is not consistent.
4	296 (128)	Signature enrollment completed, but the signature was of marginal length.
4	316 (13C)	The signature was not verified successfully.
4	348 (15C)	A probable operator error occurred. Signature verification or re-enrollment was attempted; however, no signature reference information is stored on the Personal Security Card.
4	349 (15D)	A probable operator error occurred. Signature enrollment was attempted; however, signature reference information already exists on the Personal Security card.

Return Code Dec	Reason Code Dec (Hex)	Meaning
4	350 (15E)	A probable operator error occurred; the verb stopped processing because no data was received from the signature verification pen.
4	356 (164)	A probable operator error occurred; the verb stopped processing because the signature verification pen touched the paper before the beep sounded.
4	358 (166)	A probable operator error occurred; the enrollment signatures were too short or the signatures were too inconsistent.
4	421 (1A5)	The PCF-KEY-PREFIX parameter card was not found. The default value of $$CUSP$$ will be used.
4	429 (1AD)	The digital signature is not verified. The verb completed its processing normally.

Figure A-3 (Page 2 of 2). Reason Codes for Return Code 4

Return Code 8

Figure A-4 (Page 1 of 8). Reason Codes for Return Code 8

Return Code Dec	Reason Code Dec (Hex)	Meaning
8	010 (00A)	The value that the *p_origin* parameter specifies is not valid.
8	011 (00B)	The value that the *d_origin* parameter specifies is not valid.
8	012 (00C)	The token-validation value in an external key token is not valid.
8	022 (016)	The ID number in the request field is not valid.
8	023 (017)	An access to the data area was outside the data-area boundary.
8	024 (018)	The master key verification pattern is not valid .
8	025 (019)	The value that the *text_length* parameter specifies is not valid.
8	026 (01A)	The value of the PIN is not valid.
8	027 (01B)	The card in the security interface unit is not a supported type of card.
8	028 (01C)	The object name is not valid.
8	029 (01D)	The token-validation value in an internal key token is not valid.
8	030 (01E)	No record with a matching key label is in key storage.
8	031 (01F)	The control vector did not specify a DATA key.
8	032 (020)	A key label format is not valid.
8	033 (021)	A rule array or other parameter specifies a keyword that is not valid.
8	034 (022)	A rule array keyword combination is not valid.
8	035 (023)	A rule array count is not valid.
8	036 (024)	The action command must be specified in the rule array.
8	037 (025)	The object type must be specified in the rule array.
8	038 (026)	No record in key storage exists for a key label in a cross-domain key record.
8	039 (027)	A control vector violation occurred.
8	040 (028)	The service code does not contain numerical character data.
8	041 (029)	The keyword supplied with the *key_form* parameter is not valid.
8	042 (02A)	The expiration date is not valid.
8	043 (02B)	The keyword supplied with the *key_length* or the *key_token_length* parameter is not valid.
8	044 (02C)	A record with a matching key label already exists in key storage.
8	045 (02D)	The input character string cannot be found in the code table.
8	046 (02E)	The card-validation value (CVV) is not valid.
8	047 (02F)	A source key token is unusable because it contains data that is not valid or undefined.
8	048 (030)	One or more keys has a master key verification pattern that is not valid.
8	049 (031)	A key-token-version-number found in a key token is not supported.
8	050 (032)	The key-serial-number specified in the rule array is not valid.
8	051 (033)	The value that the *text_length* parameter specifies is not a multiple of eight bytes.
8	054 (036)	The value that the *pad_character* parameter specifies is not valid.
8	055 (037)	The initialization vector in the key token is enciphered.
8	056 (038)	The master key verification pattern in the OCV is not valid.
8	058 (03A)	The parity of the operating key is not valid.
8	059 (03B)	Control information (for example, the processing method or the pad character) in the key token conflicts with that in the rule array.

Return Code Dec	Reason Code Dec (Hex)	Meaning
8	060 (03C)	A cryptographic request with the **FIRST** or **MIDDLE** keywords and a text length less than 8 bytes is not valid.
8	061 (03D)	The keyword supplied with the *key_type* parameter is not valid.
8	062 (03E)	The source key was not found.
8	063 (03F)	A key token had an invalid token header (for example, no t an internal token).
8	064 (040)	The RSA key is not permitted to perform the requested operation. Likely causes are key distribution usage is not enabled for the key.
8	065 (041)	The key token failed consistency checking.
8	066 (042)	The recovered PKCS encryption block failed validation checking.
8	067 (043)	RSA encryption failed.
8	068 (044)	RSA decryption failed.
8	070 (046)	The block name that the *block_ID* parameter specifies is not valid.
8	071 (047)	The block name was not found on the card.
8	072 (048)	The value that the *size* parameter specifies is not valid (too large, negative, or zero).
8	078 (04E)	The block name that the *block_ID* parameter specifies already exists on the card.
8	079 (04F)	The key token does not have a key-register number, the key-register number specifies an unavailable key register, or the same key-encrypted key was specified for both export keys.
8	080 (050)	The keyword supplied with the *control* parameter is not valid.
8	081 (051)	The modulus length (key size) exceeds the allowable maximum.
8	084 (054)	The time-out value is not valid.
8	085 (055)	The date or the time value is not valid.
8	086 (056)	The cryptographic period specification is not valid.
8	087 (057)	The key-reference number is not valid.
8	090 (05A)	Access is denied for this verb; the authorization level is too low, or the authorization level is not identical.
8	091 (05B)	The time sent in your logon request was more than five minutes different from the clock in the secure module.
8	092 (05C)	Your user profile has expired.
8	093 (05D)	Your user profile has not yet reached its activation date.
8	094 (05E)	Your authentication data (for example, passphrase) has expired.
8	095 (05F)	Access to the data is not authorized.
8	096 (05F)	An error occurred reading the secure clock.
8	100 (064)	The PIN length is not valid.
8	101 (065)	The PIN check length is not valid. It must be in the range from 4 to the PIN length inclusive.
8	102 (066)	The value of the decimalization table is not valid.
8	103 (067)	The value of the validation data is not valid.
8	104 (068)	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.
8	105 (069)	The cryptographic hardware component reported that the user ID or role ID is not valid.

Figure A-4 (Page 2 of 8). Reason Codes for Return Code 8

Return Code Dec	Reason Code Dec (Hex)	Meaning
8	106 (06A)	The PIN block format keyword is not valid.
8	107 (06B)	The format control keyword is not valid.
8	108 (06C)	The value of the PAD data is not valid.
8	109 (06D)	The extraction method keyword is not valid.
8	110 (06E)	The value of the PAN data is not numeric character data.
8	111 (06F)	The sequence number is not valid.
8	112 (070)	The PIN offset is not valid.
8	114 (072)	The PVV value is not valid.
8	116 (074)	The clear PIN value is not valid.
8	120 (078)	An origin or destination identifier is not valid.
8	121 (079)	The value of the *inbound_key* or *source_key* parameter is not valid.
8	122 (07A)	The value of the *inbound_KEK_count* or *outbound_count* parameter is not valid.
8	124 (07C)	An ANSI key-encrypting key is not notarized.
8	125 (07D)	The control vector for an ANSI key-encrypting key does not allow notarization, and the notarization process is requested.
8	152 (098)	The security interface unit and the Personal Security card do not provide the requested cipering method.
8	153 (099)	The text length exceeds the system limits, or you attempted data chaining with the Security Interface Unit and the Personal Security card.
8	154 (09A)	The key token that the *key_identifier* parameter specifies is not an internal key token or a key label.
8	155 (09B)	The value that the *generated_key_identifier* parameter specifies is not valid, or it is not consistent with the value that the *key_form* parameter specifies.
8	156 (09C)	A keyword is not valid with the specified parameters.
8	157 (09D)	The key-token type is not specified in the rule array.
8	159 (09F)	The keyword supplied with the *option* parameter is not valid.
8	160 (0A0)	The key type and the key length are not consistent.
8	161 (0A1)	The value that the *data_set_name_length* parameter specifies is not valid.
8	162 (0A2)	The offset value is not valid.
8	163 (0A3)	The value that the *data_set_name* parameter specifies is not valid.
8	164 (0A4)	The starting address of the output area falls inside the input area.
8	165 (0A5)	The carry_over_character_count that is specified in the chaining vector is not valid.
8	168 (0A8)	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.
8	169 (0A9)	An MDC_Generate text length error occurred.
8	170 (0AA)	The minimum authorization level value is not valid. The valid range is from 0 to 255.
8	171 (0AB)	The *control_array_count* value is not valid.
8	172 (0AC)	The device_type field of the key token is not valid.
8	173 (0AD)	The key tokens specify different cryptographic hardware components.
8	175 (0AF)	The key token cannot be parsed because no control vector is present.
8	176 (0B0)	The *binary_time_stamp* value is not valid.

Figure A-4 (Page 3 of 8). Reason Codes for Return Code 8

Figure A-4 (Page 4 of 8). Reason Codes for Return Code 8		
Return Code Dec	**Reason Code Dec (Hex)**	**Meaning**
8	177 (0B1)	The *time_stamp* value is not valid.
8	178 (0B2)	The device type must be specified in the rule array.
8	180 (0B4)	A null key token was presented for parsing.
8	181 (0B5)	The key token is not valid. The first byte is not valid, or an incorrect token type was presented.
8	183 (0B7)	The key type is not consistent with the key type of the control vector.
8	184 (0B8)	An input pointer is null (workstation security API only).
8	185 (0B9)	The data-set file does not exist or a disk I/O error occurred.
8	186 (0BA)	The key-type field in the control vector is not valid.
8	187 (0BB)	The requested MAC length (MACLEN4, MACLEN6, MACLEN8) is not consistent with the control vector (key-a, key-b).
8	189 (0BD)	The key cannot be stored in the key register.
8	190 (0BE)	This function cannot operate on a key stored in a key register.
8	191 (0BF)	The requested MAC length (MACLEN6, MACLEN8) is not consistent with the control vector (MAC-LN-4).
8	192 (0C0)	A key-storage record contains a record validation value that is not valid.
8	193 (0C1)	The specified cryptographic hardware component is the Personal Security card; therefore, you must use a key-register number.
8	198 (0C6)	The user can be identified only through signature verification. The signature verification pen is not installed.
8	203 (0CB)	The name_list_array_count value is too small or not valid. The value must be equal to or greater than the number of block names. The maximum value is 255.
8	204 (0CC)	A memory allocation failed (workstation security API only).
8	205 (0CD)	The X9.23 ciphering method is not consistent with the use of the CONTINUE keyword.
8	304 (130)	The secure session between the components cannot be established.
8	323 (143)	The ciphering method that the Decipher verb used does not match the ciphering method that the Encipher verb used.
8	335 (14F)	Either the specified cryptographic hardware component or the environment does not implement this function.
8	340 (154)	One of the input control vectors has odd parity.
8	343 (157)	Either the data block or the buffer for the block is too small.
8	345 (159)	Insufficient storage space exists for data in the data block area.
8	346 (15A)	The requested command is not valid in the current state of the cryptographic hardware component.
8	358 (166)	The PPV enroll or re-enroll function was attempted, but the signatures were too inconsistent.
8	360 (168)	A PPV function was attempted, but the signature that the signature verification pen gathered was too short.
8	362 (16A)	An enroll or a re-enroll was attempted, but not enough space exists on the Personal Security card to hold the signature reference.
8	364 (16C)	The download code table was full when a Load MDC command was attempted.
8	365 (16D)	The download code name already existed in the download code table when a Load MDC command was attempted.

Figure A-4 (Page 5 of 8). Reason Codes for Return Code 8		
Return Code Dec	**Reason Code Dec (Hex)**	**Meaning**
8	366 (16E)	The download code name did not exist when a Load Code command was attempted.
8	367 (16F)	The program was not loaded when the EXEC program option of the Load Code command was attempted.
8	368 (170)	The requested command is not valid when the device is in the initialization state.
8	370 (172)	The requested option is not valid under the current circumstances (for example, when you issue a Read Block command with the option for reading a secured block, but the requested block is defined as non-secured).
8	371 (173)	You are not authorized to use this key. This might be due to an incorrect security token.
8	372 (174)	The cryptographic hardware component reported an unknown command. This might be caused by the Command Unavailable bit being turned on for this command in the Command Configuration Table.
8	373 (175)	The security token is not correct. (A security token is a password to a key register.)
8	374 (176)	Less data was supplied than expected or less data exists than was requested.
8	377 (179)	A key storage error occurred.
8	379 (17B)	This verb requires a secure session to be established.
8	382 (17E)	A time limit violation occurred.
8	383 (17F)	The user re-inserted the card or a card-eject failure occurred. A manual eject is required.
8	385 (181)	The cryptographic hardware component reported that the data passed as part of a command is not valid for that command.
8	387 (183)	
8	388 (184)	A control vector with an extension was received; however, no control-vector extension table was loaded.
8	389 (185)	The first byte of a control-vector extension was not X'00'.
8	390 (186)	A control vector extension is not valid for this key type.
8	391 (187)	The index byte of the extension (for example, the second byte) was X'00', or the index byte of the extension was greater than the number of entries in the currently loaded control-vector extension table.
8	392 (188)	One or more bits were turned on in the control-vector extension for which the corresponding bit was turned off in the selected control-vector extension table entry.
8	393 (189)	The command was not processed because the profile cannot be used.
8	394 (18A)	The command was not processed because the expiration date was exceeded.
8	395 (18B)	The command was not processed because processing on a holiday was attempted.
8	397 (18D)	The command was not processed because the active profile requires the user to be pre-verified.
8	398 (18E)	The command was not processed because the maximum PIN/password failure limit is exceeded.
8	401 (191)	The data key conversion user exit, CSUDMGR9, returned a return code of 4. The data key conversion is rejected.
8	402 (192)	The data key conversion user exit, CSUDMGR9, returned a return code of 8. The data key conversion is terminated.

Return Code Dec	Reason Code Dec (Hex)	Meaning
8	403 (193)	The data key conversion user exit, CSUDMGR9, returned an invalid reason code. The process is terminated.
8	406 (196)	A PIN formatting error occurred.
8	407 (197)	A PIN block consistency check error occurred.
8	412 (19C)	The signature has more than 25 segments.
8	420 (1A4)	One or more key records are temporarily locked by an in-process key-storage synchronization operation. Please try again (MVS host security API only).
8	421 (1A5)	The request cannot be processed because the key-storage synchronization server is dumping key storage or changing the master key (MVS host security API only).
8	601 (259)	The object name that is being registered already exists in the table.
8	602 (25A)	The object that is being loaded is not registered.
8	603 (25B)	The object that is being managed is not known. It probably is not registered.
8	604 (25C)	The user-defined function facility does not recognize the requested user-defined function.
8	605 (25D)	The number of output bytes is greater than the number that is permitted.
8	606 (25E)	A stack operation of a user-defined function addressed an entry that is beyond the limits of the stack.
8	608 (260)	The first specified Save Area for this DIVISA instruction in a user-defined function contains a zero.
8	609 (261)	The target of a JUMP instruction is outside the user-defined function Set Code area.
8	610 (262)	The target of a UCALL instruction is outside the user-defined function Set Code area.
8	611 (263)	The user-defined function attempted to use a control vector that has non-even parity bytes.
8	612 (264)	The user-defined function attempted to use a key that has non-odd parity bytes.
8	613 (265)	The user-defined function's access to the I/O buffer is outside the I/O buffer boundary.
8	614 (266)	The user-defined function attempted a POP instruction, but the stack was empty. The top-of-stack pointer indicated the initial stack address.
8	615 (267)	The user-defined function attempted a PUSH instruction, but the stack was full. The top-of-stack pointer indicated the last stack address.
8	616 (268)	The system attempted to register an object, but the internal object table was full.
8	617 (269)	The system attempted to load an external object, but external objects cannot be loaded into the coprocessor.
8	618 (26A)	The system attempted to load a user-defined program, but the MCS storage did not contain enough space to hold the program.
8	619 (26B)	The calculated MDC did not match the MDC that is registered for the object.
8	620 (26C)	The requested object is not loaded into the coprocessor.
8	621 (26D)	The level of the UDF_MACS.INC file that this user-defined function used is not compatible with the level of microcode.
8	622 (26E)	The user-defined function nesting level is greater than 16.

Figure A-4 (Page 6 of 8). Reason Codes for Return Code 8

Figure A-4 (Page 7 of 8). Reason Codes for Return Code 8		
Return Code Dec	**Reason Code Dec (Hex)**	**Meaning**
8	623 (26F)	The user-defined function UCALL nesting level is greater than 16.
8	624 (270)	The user-defined program attempted to call the user-defined function, but the user-defined function's name or extension was not valid.
8	625 (271)	The total object size is too large.
8	626 (272)	The code-only of the external object cannot be deleted.
8	627 (273)	The object is already loaded.
8	628 (274)	The format of the user-defined program is not valid.
8	630 (276)	A user-defined program attempted to access memory outside the memory that is allocated to the user-defined program.
8	703 (2BF)	A new master key value was found to be one of the weak DES keys.
8	704 (2C0)	The new master key would have the same master key verification pattern as current the current master key.
8	705 (2C1)	The same key-encrypting key was specified for both exporter keys.
8	706 (2C2)	Pad count in deciphered data is not valid.
8	707 (2C3)	The Master Key registers are not in the state required for the requested function.
8	713 (2C9)	The algorithm or function is not available on current hardware (DES on a CDMF-only system).
8	714 (2CA)	A reserved parameter was not a null pointer or an expected value.
8	718 (2CE)	The hash of the data block in the decrypted RSA-OAEP block does not match the hash of the decrypted data block.
8	719 (2CF)	The block format (BT) field in the decrypted RSA-OAEP block does not have the correct value.
8	720 (2D0)	The initial byte (I) in the decrypted RSA-OAEP block does not have a valid value.
8	721 (2D1)	The V field in the decrypted RSA-OAEP does not have the correct value.
8	752 (2F0)	The key-storage file path is not usable.
8	753 (2F1)	Opening the key-storage file failed.
8	754 (2F2)	An internal call to the key_test command failed.
8	756 (2F4)	Creation of the key-storage file failed.
8	760 (2F8)	An RSA-key modulus length in bits or in bytes is not valid.
8	761 (2F9)	An RSA-key exponent length is not valid.
8	762 (2FA)	A length in the key value structure is not valid.
8	763 (2FB)	The section identification number within a key token is invalid.
8	770 (302)	The PKA key token has an invalid field.
8	771 (303)	The user is not logged on.
8	772 (304)	The requested role was not found.
8	773 (305)	The requested profile was not found.
8	774 (306)	The profile already exists.
8	775 (307)	The supplied data is not replaceable.
8	776 (308)	The requested Id is already logged on.
8	777 (309)	The authentication data is invalid.
8	778 (30A)	The checksum for the role is in error.
8	779 (30B)	The checksum for the profile is in error.
8	780 (30C)	There is an error in the profile data.
8	781 (30D)	There is an error in the role data.
8	782 (30E)	The Function-Control-Vector header is invalid.

Figure A-4 (Page 8 of 8). Reason Codes for Return Code 8		
Return Code Dec	**Reason Code Dec (Hex)**	**Meaning**
8	783 (30F)	The command is not permitted by the Function-Control-Vector value.
8	784 (310)	The operation you requested cannot be performed because the user profile is in use.
8	785 (311)	The operation you requested cannot be performed because the role is presently in use.
8	1025 (401)	Registered Public Key or Retained Private Key Name already exists.
8	1026 (402)	Key name (Registered Public Key or Retained Private Key) does not exist.
8	1027 (403)	Environment Identification Data is already set.
8	1028 (404)	Master Key Share Data is already set.
8	1029 (405)	There is an error in the Environment Identification Data.
8	1030 (406)	There is an error in using the Master Key Share Data.
8	1031 (407)	There is an error in using Registered Public Key or Retained Private Key data.
8	1032 (408)	There is an error in using Registered Public Key Hash data.
8	1033 (409)	The Public Key Hash was not registered.
8	1034 (40A)	The Public Key was not registered.
8	1035 (40B)	The Public Key Certificate Signature was not verified.
8	1037 (40D)	There is a Master Key Shares distribution error.
8	1038 (40E)	The Public Key Hash is not marked for cloning.
8	1039 (40F)	The Registered Public Key Hash does not match the Registered Hash.
8	1040 (410)	The Master Key Share Enciphering Key failed encipher.
8	1041 (411)	The Master Key Share Enciphering Key failed decipher.
8	1042 (412)	The Master Key Share Digital Signature Generate failed.
8	1043 (413)	The Master Key Share Digital Signature Verify failed.
8	1044 (414)	There is an error in reading VPD data from the adapter.
8	1045 (415)	Encrypting the Cloning Information failed.
8	1046 (416)	Decrypting the Cloning Information failed.
8	1047 (417)	There is an error loading New Master Key from Master Key Shares.
8	1048 (418)	The Clone Information has one or more invalid sections.
8	1049 (419)	The Master Key Share Index is not valid.
8	1100 (44C)	General hardware device driver execution error.
8	1101 (44D)	Hardware device driver invalid parameter.
8	1102 (44E)	Hardware device driver invalid buffer length.
8	1103 (44F)	Hardware device driver too many opens. Cannot open device now.
8	1104 (450)	Hardware device driver access denied. Cannot access device.
8	1105 (451)	Hardware device driver device is busy and cannot perform request now.
8	1106 (452)	Hardware device driver buffer too small. Received data truncated.
8	1107 (453)	Hardware device driver request interrupted. Request aborted.
8	1108 (454)	Hardware device driver security tamper. Hardware intrusion detected.

Return Code 12

Figure A-5. Reason Codes for Return Code 12

Return Code Dec	Reason Code Dec (Hex)	Meaning
12	093 (05D)	The security server is not available or not loaded.
12	097 (061)	File space in key storage is insufficient to complete the operation.
12	194 (0C2)	No internal working storage is available in the Network Security Processor.
12	195 (0C3)	The Network Security Processor group is not valid (MVS host security API only).
12	196 (0C4)	The device driver, the security server, or the directory server is not installed, or is not active, or in AIX, file permissions are not valid for your application.
12	197 (0C5)	A key-storage file I/O error occurred, or a file was not found (workstation security API only).
12	199 (0C7)	A Network Security Processor is not available (MVS host security API only).
12	201 (0C9)	The Network Security Processor subsystem is not active (MVS host security API only).
12	202 (0CA)	The Network Security Processor subsystem was not loaded (MVS host security API only).
12	206 (0CE)	The key-storage file is not valid, or the master-key verification failed.
12	207 (0CF)	The verification method flags in the profile are not valid.
12	324 (144)	The device driver attempted to allocate memory, but no memory is available.
12	338 (152)	This cryptographic hardware component is not installed.
12	339 (153)	A system error occured in interprocess communication routine.
12	428 (1AC)	The BWK parameter file (DDNAME=BWKPARM) did not open properly.
12	607 (25F)	A microcode service that the user-defined function microcode called returned an unexpected error.
12	629 (275)	The user-defined program overlay file has not loaded yet.
12	764 (2FC)	The master key(s) are not loaded and therefore a key could not be recovered or enciphered.
12	768 (300)	One or more paths for key storage directory operations is improperly specified.

Return Code 16

Figure A-6 (Page 1 of 2). Reason Codes for Return Code 16

Return Code Dec	Reason Code Dec (Hex)	Meaning
16	099 (063)	An unrecoverable error occurred in the security server; contact your IBM service representative.
16	099 (063)	A software error occurred (OS/400 security API only).
16	150 (096)	An error occurred in the Network Security Processor MVS support program.
16	167 (0A7)	An error occurred in the security server, possibly due to inconsistent device-driver and security-server logic.
16	200 (0C8)	The cross-memory server or request manager abended (MVS host security API only).
16	298 (12A)	The MDC of the signature verification overlay file did not verify, or the format of the signature verification overlay file is not valid.
16	326 (146)	An error occurred when reading the signature verification overlay file.
16	327 (147)	An error occurred when opening the signature verification overlay file.
16	336 (150)	An error occurred in a cryptographic hardware component.
16	337 (151)	A device software error occurred.
16	347 (15B)	A communications error occurred.
16	351 (15F)	An unknown signature verification error occurred.
16	352 (160)	A signature data acquisition error occurred.
16	353 (161)	An unknown error occurred during a card-read function.
16	354 (162)	An unknown error occurred during a card-write function.
16	355 (163)	An unknown error occurred during a create-block function.
16	357 (165)	A signature verification function was attempted, but the signature reference information that the signature verification pen sent was not valid.
16	359 (167)	The signature verification function completed, but a failure occurred when notifying the security interface unit or the Personal Security card.
16	361 (169)	A signature verification function was attempted, but the security interface unit pen buffer had an overrun error.
16	363 (16B)	The signature verification option is not valid.
16	375 (177)	The Personal Security Card processor indicated that an error occurred while writing to the EEPROM.
16	376 (178)	Data that was read from the Personal Security card's EEPROM did not match the data that was written there.
16	399 (18F)	The cryptographic adapter intrusion latch reset failed.
16	413 (19D)	A signature verification communication error occurred.
16	414 (19E)	A signature verification file-length error occurred.
16	415 (19F)	A signature verification tone-generation error occurred.
16	416 (1A0)	A signature verification enroll-authorization communication error occurred.
16	444 (1BC)	The verb-unique-data had an invalid length.
16	556 (22C)	The request parameter block failed consistency checking.
16	708 (2C4)	Inconsistent data was returned from the cryptographic engine.
16	709 (2C5)	Cryptographic engine internal error, could not access the master key data.
16	710 (2C6)	An unrecoverable error occurred while attempting to update master key data items.
16	712 (2C8)	An unexpected error occured in the master key manager.

16 712 (2C8) An unexpected error occured in the master key manager.

Figure A-6 (Page 2 of 2). Reason Codes for Return Code 16		
Return Code Dec	**Reason Code Dec (Hex)**	**Meaning**
16	769 (301)	The host system code or the CCA application in the &retc.769b.

Return Code 24

Figure A-7. Reason Codes for Return Code 24		
Return Code Dec	**Reason Code Dec (Hex)**	**Meaning**
24	057 (039)	The verb processing is rejected because the server is in exclusive control mode with another application program.
24	057 (039)	The verb processing is rejected because the server is in exclusive control mode with another application program.

Appendix B. Data Structures

This appendix describes the following data structures:

- Key tokens
- Chaining vector records
- Key storage records
- Key record list data set
- Access control data structures
- | Master key shares
- | Distributed function control vector.

Key Tokens

This section describes the DES and RSA *key tokens* used with the product. A “key token” is a data structure that contains information about a key and usually contains a key or keys.

in general, keys available to an application program, or keys held in key storage, are 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* is used to hold a key and its related information for use at a specific node.

An *external key token* 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 RSA private keys in an external key token are multiply-enciphered by a *transport* key. In a CCA-node, a transport key is a double-length DES Key-Encrypting-Key.

The remainder of this section describes the structures used with the Fortress product family:

- Token master key verification pattern
- Token-validation value
- Record-validation value
- Null key token
- DES key tokens
  - Internal DES key token
  - External DES key token
  - DES key token flag bytes
- RSA key tokens
- Chaining Vector Records
- Key Storage Records
- Key Record List Data Set

Master Key Verification Pattern

A Master Key Verification Pattern (MKVP) within an internal key token permits the cryptographic engine to detect if the key within the token is enciphered by an available master key. These steps produce the master key verification pattern:

- Prefix the 24-byte master key with a header byte of X'01'
- Calculate a SHA-1 hash on the 25-byte string

© Copyright IBM Corp. 1997-98

- Return the high-order two bytes of the 20-byte SHA-1 hash as the master key verification pattern.

A CCA node will not permit the introduction of a new master key value that has the same two-byte verification pattern as either the current-master-key verification pattern or as the old-master-key verification pattern.

Token-Validation Value and Record-Validation Value

The Token-Validation Value (TVV) is a checksum that helps ensure that an application program-provided key token is valid. A Token-Validation Value is the sum (two's complement ADD), ignoring carries and overflow, on the key token by operating on four bytes at a time, starting with bytes zero to three and ending with bytes 56 to 59. The four-byte strings are treated as big-endian binary numbers with the high-order byte stored in the lower address. DES key token bytes 60 to 63 contain the Token-Validation Value.

When an application program supplies a key token, the CCA node checks the Token-Validation Value. When a CCA verb generates a DES key token, it generates a Token-Validation Value in the key token.

The record-validation value (RVV) used in DES key storage records uses the same algorithm as the Token-Validation Value. The RVV is the sum of the bytes in positions 0 to 123 except for bytes 60 to 63.

Null Key Token

Figure B-1 shows the null key token format. With some CCA verbs, a null key token can be used instead of an internal or an external key token. A verb generally accepts a null key token as a signal to use a key token with default values in lieu of the null key token.

A null key token is indicated by the value X'00' at offset zero in a key token, a key token variable, or a key identifier variable.

PKA key storage uses an 8-byte structure, shown below, to represent a null key token. The PKA_Key_Record_Read verb will return this structure if a key record with a null key token is read. Also, if you examine PKA key storage, you should expect key records without a key token containing specific key values to be represented by a "null key token." In the case of key storage records, the record length (offset 2 and 3) can be greater than 8.

Figure B-1. PKA Null Key Token Format

Offset	Length	Meaning
00	01	X'00' This indicates that this is a null key token
01	X'00'	Version zero
02	02	X'0008' Indicates a PKA null key token.
04	04	Reserved

The key_import verb accepts input with offset zero valued to X'00'. In this special case, the verb treats information starting at offset 16 as an enciphered, single length key. In a very limited sense, this special case can be considered a "null key token."

Internal DES Key Token

Figure B-2. Internal Key Token Format

Offset	Length	Meaning
00	1	X'01' (a flag that indicates an internal key token)
01	1	Reserved, binary zero
02	2	Master key verification pattern
04	1	The version number (X'03')
05	1	Reserved, binary zero
06	1	Flag byte 1; for more information, see Figure B-4 on page B-4
07	1	Reserved, binary zero
08-15	8	Reserved, binary zero
16-23	8	The single-length encrypted key or the left half of a double-length encrypted key.
24-31	8	Null, or the right half of a double-length operational key
32-39	8	The control-vector base
40-47	8	Null, or the control vector base for the second eight-byte portion of a 16-byte key
48-59	12	Reserved, binary zero
60-63	4	The token-validation value

External DES Key Token

Figure B-3. External Key Token Format

Offset	Length	Meaning
00	1	X'02' (a flag that indicates an external key token)
01	3	Reserved, binary zero
04	1	The version number (X'00')
05	1	Reserved, binary zero
06	1	Flag byte 1; for more information, see Figure B-4
07	1	Flag byte 2; for more information, see Figure B-5 Reserved, generally X'00', except X'02' will be tolerated.
08-15	8	Reserved, binary zero
16-23	8	The single-length encrypted key or the left half of a double-length encrypted key.
24-31	8	Null, or the right half of a double-length encrypted key
32-39	8	The control-vector base
40-47	8	Null, or the control vector base for the second 8-byte portion of a 16-byte key
48-59	12	Reserved, binary zero
60-63	4	The token-validation value

DES Key Token Flag Byte 1

Figure B-4. Key Token Flag Byte 1

Bits (MSB...LSB)[1]	Meaning
1xxx xxxx	The encrypted key value, and as used in an implementation, the Master Key Version Number or verification pattern are present
0xxx xxxx	An encrypted key is not present
x0xx xxxx	The control-vector value is not present
x1xx xxxx	The control-vector value is present
	All other bit combinations are reserved; undefined bits should be zero.

DES Key Token Flag Byte 2

Figure B-5. Key Token Flag Byte 2

Bits (MSB...LSB)	Meaning
	For Key-Encrypting Keys
0000 0010	This Key-Encrypting key will import and export external key tokens using the Transaction Security System key token format.

[1] MSB is the most significant bit; LSB is the least significant bit.

RSA Key Token Formats

An RSA key token contains various items, some of which are optional, and some of which can be present in different forms. The token is composed of concatenated *sections* that must occur in the prescribed order.

As with other CCA key tokens, both internal and external forms are defined.

- An RSA internal key token contains a private key that is protected by encrypting the information using the CCA-node master key. The internal key token will also contain private key blinding information, the modulus and the public-key exponent. A master key verification pattern is also included to enable determination that the proper master key is available to process the protected private key. The format and content of an internal key token is local to a specific node and product implementation, and does not represent an interchange format.
- An RSA external key token contains the modulus and the public-key exponent. Also, the external key token optionally contains the private key. If present, the private key may be in the clear or may be protected by encryption using a double-length DES transport key. An external key token is an inter-product interchange data structure.

The private key can be represented in one of two forms:

- By a modulus and the private-key exponent
- By a set of numbers used in the *Chinese-remainder-theorem*.

Protection of the private key is provided by encrypting a *confounder* (a random number) and the private key information. The private key in an external key token is protected by a double-length transport key and the EDE2 algorithm, see "CCA RSA Private Key Encryption and Decryption Process" on page C-10. The private key and the blinding values in an internal key token are protected by the triple-length master key and the EDE3 algorithm, see "CCA RSA Private Key Encryption and Decryption Process" on page C-10.

An RSA key token is the concatenation of this ordered set of sections:

- A token header:
  - An internal header (first-byte X'1F')
  - An external header (first-byte X'1E')
- An optional private-key section in one of these formats:
  - 1024-bit modular-exponentiation format, fixed length (section identifier X'02')
  - 2048-bit Chineese-remainder format, variable length (section identifier X'05')
- A public-key section (section identifier X'04')
- An optional key-name section (section identifier X'10')
- | An optional certificate(s) section (section identifier X'40')
- A private-key blinding section on an internal key token (section identifier is X'FF').

| The key tokens can be built with the PKA_Key_Token_Build verb.

RSA Key Token Integrity: If the token contains private key information, then the integrity of the information within the token can be verified by computing the SHA-1 hash values that are found in the private-key sections (portions of the key token). The SHA-1 hash value at offset four within the private-key section requires access to the cleartext values of the private-key components. The cryptographic engine will verify this hash quantity whenever it retrieves the secret key for productive use.

A second SHA-1 hash value is located at offset 30 within the private key section. This hash value is computed on the remainder of the key token following the private-key section. The value of this SHA-1 hash is included in the computation of the hash at offset four. As with the offset-four hash value, the hash at offset 30 is validated whenever a private key is recovered from the token for productive use.

In addition to the hash checks, various token format and content checks are performed to validate the key values.

The optional private-key name section can be used by access monitor systems (e.g. RACF) to ensure that the application program is entitled to employ the particular private key.

RSA Key Token Sections

These key-token-section data structures are described in the following tables:

- Figure B-6 on page B-7, RSA Token Header
- Figure B-7 on page B-7, RSA Private Key, 1024-Bit Modular-Exponentiation Format
- Figure B-8 on page B-8, RSA Private Key, 2048-Bit Chineese-Remainder Format
- Figure B-9 on page B-9, RSA Public Key
- Figure B-10 on page B-9, RSA Private-key Name
- Figure B-11 on page B-10, RSA Public-key Certificate(s)
- Figure B-18 on page B-13, RSA Private-key Blinding Information

Notes:

1. All length fields are in binary.
2. All binary fields (exponents, lengths, etc.) are stored with the high-order byte first (left, low-address, S/390 format); thus the significant bits are to the right and preceded with zero-bits to the width of a field.
3. In variable length binary fields that have an associated field-length value, leading bytes that would contain X'00' can be dropped and the field shortened to contain the significant bits.

Figure B-6. RSA Token Header

Offset (Bytes)	Length (Bytes)	Description
000	001	Token identifier X'1E' External token; the optional private key is either in cleartext or enciphered by a transport key-encrypting key. X'1F' Internal token; the private key is enciphered by the master key.
001	001	Version, X'00'
002	002	Length of the key token structure
004	004	Reserved, binary zero

Figure B-7. RSA Private Key, 1024-Bit Modular-Exponentiation Format

Offset (Bytes)	Length (Bytes)	Description
000	001	X'02', Section identifier, RSA private key, modular-exponent format (RSA-PRIV)
001	001	X'00', Version
002	002	Length of the RSA private-key section X'016C' (364 decimal)
004	020	SHA-1 hash value of the private-key subsection cleartext, offset 28 to the section end
024	002	Reserved, binary zero
026	002	Master key verification pattern in an internal key token, else X'0000'
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, etc. sections; if there is no name section or other optional section, then 20 bytes of X'00'.
050	001	Key usage flag X'00' Signature usage only X'80' Signature and symmetric key management usage permitted
051	009	Reserved, binary zero
060	024	Reserved, binary zero
052	Start of the optionally-encrypted secure subsection	
084	024	Random number, confounder
108	128	Private-key exponent, d. $d=e^{-1}mod((p-1)(q-1))$, and $1<d<n$ where e is the public exponent.
	End of the optionally encrypted subsection; all of the fields starting with the confounder field and ending with the variable length pad field are enciphered for key confidentiality when the key format and security flags (offset 28) indicate that the private key is enciphered.	
236	128	Modulus, n. $n=pq$ where p and q are prime and $2^{512}<n<2^{1024}$

Figure B-8 (Page 1 of 2). Private Key, 2048-Bit Chineese-Remainder Format		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'05', Section identifier, RSA private key, CRT (RSA-OPT) format
001	001	X'00', Version
002	002	Length of the RSA private-key section, 76 +ppp +qqq +rrr +sss +ttt +uuu +xxx +nnn
004	020	SHA-1 hash value of the private-key subsection cleartext, offset 28 to the end of the modulus.
024	002	Length in bytes of the optionally-encrypted secure subsection, or X'0000' if the subsection is not encrypted
026	002	Master key verification pattern in an internal key token, else X'0000'
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, binary zero
030	020	SHA-1 hash of the optional key-name, etc. sections; if there is no name section or other optional section, then 20 bytes of X'00'.
050	001	Key usage flag X'00' Signature usage only X'80' Signature and symmetric-key-management usage permitted
051	001	Reserved, binary zero
052	Start of the optionally-encrypted secure subsection	
052	008	Random number, confounder
060	002	Length of the prime number, p, in bytes: ppp
062	002	Length of the prime number, q, in bytes: qqq
064	002	Length of the d_p, in bytes: rrr
066	002	Length of the d_q, in bytes: sss
068	002	Length of the A_p, in bytes: ttt
070	002	Length of the A_q, in bytes: uuu
072	002	Length of the modulus, n., in bytes: nnn
074	002	Length of the padding field, in bytes: xxx
076	ppp	Prime number, p
076 +ppp	qqq	Prime number, q
076 +ppp +qqq	rrr	d_p = d mod(p-1)
076 +ppp +qqq +rrr	sss	d_q = d mod(q-1)

Figure B-8 (Page 2 of 2). Private Key, 2048-Bit Chineese-Remainder Format		
Offset (Bytes)	**Length (Bytes)**	**Description**
076 +ppp +qqq +rrr +sss	ttt	$A_p = q^{p-1} \bmod(n)$
076 +ppp +qqq +rrr +sss +ttt	uuu	$A_q = (n+1-A_p)$
076 +ppp +qqq +rrr +sss +ttt +uuu	xxx	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
		End of the optionally-encrypted subsection; all of the fields starting with the confounder field and ending with the variable length pad field are enciphered for key confidentiality when the key format-and-security flags (offset 28) indicate that the private key is enciphered.
076 +ppp +qqq +rrr +sss +ttt +uuu +xxx	nnn	Modulus, n. n=pq where p and q are prime and $2^{512}<n<2^{2048}$

Figure B-9. RSA Public Key		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'04', Section identifier, RSA public key
001	001	X'00', Version
002	002	Section length, 12+xxx+yyy
004	002	Reserved, binary zero
006	002	RSA public-key exponent field length in bytes, “xxx”
008	002	Public-key modulus length in bits.
010	002	RSA public-key modulus field length in bytes, “yyy” **Note:** If the token contains an RSA private-key section, this field length, yyy, should be zero. The RSA private-key section will contain the modulus.
012	xxx	Public-key exponent, e (this field length will generally be 1, 3, or 64 to 256 bytes). e must be odd and $1<e<n$. (e is frequently valued to 3 or $2^{16}+1$ (=65 537), otherwise e is of the same order of magnitude as the modulus) **Note:** You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation will not generate an “RSA” key with a public exponent valued to two (a “Rabin” key).
012 +xxx	yyy	Modulus, n. n=pq where p and q are prime and $2^{512}<n<2^{2048}$. This field will be absent when the modulus is contained in the private-key-section. If present, the field length will be 64 to 256 bytes

Figure B-10. RSA Private-key Name		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'10', Section identifier, private-key name
001	001	X'00', Version
002	002	Section length, X'0044' (68 decimal)
004	064	Private-key name, left-justified, padded with space characters (X'20'). The private-key name can be used by an access control system to validate the calling application's entitlement to employ the key

RSA Public-key Certificate Section: An optional *public key certificate(s)* section can be included in an RSA key token. The section consists of:

- The section header (identifier X'40')
- A public key subsection
- An optional certificate information subsection with any or all of these elements:
  - User data
  - EID
  - Serial number
- A self-signature subsection.

The section (as with the rest of the key token) is composed of a series of "tag-length-variable" (TLV) items to form a self-defining data structure. One or more TLV items can be included in the variable portion of a higher level TLV item.

The section header is described followed by descriptions of the TLV items that can be included in the section.

Figure B-11. RSA Public-key Certificate(s) Section Header		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'40', Section identifier, certificate
001	001	X'00', Version
002	002	Section length; includes: • Section header • Public key subsection (except for a signature usage in Access_Control_Maintenance and Cryptographic_Facility_Query verbs) • Information subsection (optional) • Signature subsection(s).

Figure B-12. RSA Public-key Certificate(s) Public Key Subsection		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'41', Public Key Subsection identifier
001	001	X'00', Version
002	002	Subsection length, 12+xxx+yyy
004	002	RSA public-key exponent field length in bytes,
006	002	RSA public-key exponent field length in bytes, “xxx”
008	002	Public-key modulus length in bits
010	002	RSA public-key modulus field length in bytes, “yyy”
012	xxx	Public-key exponent, e (this field length will generally be 1, 3, or 64 to 256 bytes). e must be odd and 1<e<n. (e is frequently valued to 3 or 2^{16}+1 (=65 537), otherwise e is of the same order of magnitude as the modulus) **Note:** You can import an RSA public key having an exponent valued to two (2). Such a public key can correctly validate an ISO 9796-1 digital signature. However, the current product implementation will not generate an “RSA” key with a public exponent valued to two (a “Rabin” key).
012+xxx	yyy	Modulus, n. n=pq where p and q are prime and 2^{512}<n<2^{2048}. This field will be absent when the modulus is contained in the private-key-section. If present, the field length will be 64 to 256 bytes

Figure B-13. RSA Public-key Certificate(s) Optional Information Subsection Header		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'42', Information Subsection Header
001	001	X'00', Version
002	002	Subsection length, 4+iii
004	iii	The information field that will contain any of the includable TLV entities: • User data (Id = 50) • EID (Id = 51) • Serial number (Id = 52)

Figure B-14. RSA Public-key Certificate(s) User Data TLV		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'50', User Data TLV Header
001	001	X'00', Version
002	002	TLV length, 4+uuu
004	uuu	User provided data. 0 ≤ uuu ≤ 64

Figure B-15. RSA Public-key Certificate(s) Environment Identifier (EID) TLV		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'51', Private Key Environment Identifier TLV Header
001	001	X'00', Version
002	002	X'0014', TLV length, 20
004	016	EID string of the CCA node that generated the public (and private) key. (This TLV must be provided in a skeleton key token with usage of the PKA_Key_Generate verb. The verb will fill in the EID string prior to certifying the public key.)

Figure B-16. RSA Public-key Certificate(s) Serial Number TLV		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'52', Serial Number TLV Header
001	001	X'00', Version
002	002	X'000C', TLV length, 12
004	008	Serial number of the Coprocessor that generated the public (and private) key. (This TLV must be provided in a skeleton key token with usage of the PKA_Key_Generate verb. The verb will fill in the serial number prior to certifying the public key.)

Figure B-17. RSA Public-key Certificate(s) Signature Subsection		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'45', Signature Subsection Header
001	001	X'00', Version
002	002	Subsection length, 70+sss
004	001	Hashing algorithm identifier; X'01' signifies use of the SHA-1 hashing algorithm.
005	001	Signature formatting identifier; X'01' signifies use of the ISO-9796 process.
006	064	Signature-key identifier; the key label of the key used to generate the signature.
070	sss	The signature field. The signature is calculated on data that begins with the Signature Section Identifier (X'40') through the byte immediately preceding this signature field.
Note: Note that more than one Signature Subsection can be included in a Signature Section; this accommodates the possibility of a self-signature as well as a device-key signature.		

RSA Private-key Blinding Information:

Figure B-18. RSA Private-key Blinding Information		
Offset (Bytes)	**Length (Bytes)**	**Description**
000	001	X'FF', Section identifier, private-key blinding information
001	001	X'00', Version
002	002	Section length, 34 + rrr + iii
004	020	SHA-1 hash value of the internal information subsection cleartext, offset 28 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
024	002	Length in bytes of the encrypted secure subsection
026	002	Reserved, binary zero
028	Start of the encrypted secure subsection	
028	002	Length of the random number r, in bytes: rrr
030	002	Length of the random number inverse r^{-1}, in bytes: iii
032	002	Length of the padding field, in bytes xxx
034	rrr	Random number r (used in blinding)
034 +rrr	iii	Random number r^{-1} (used in blinding)
034 +rrr +iii	xxx	0x00 padding of length xxx bytes such that the length from the start of the encrypted subsection to the end of the padding field is a multiple of eight bytes.
	End of the encrypted subsection.	

Chaining Vector Records

The *chaining_vector* parameter specifies an address that points to the place in main storage that contains an 18-byte work area that is required with the Cipher, MAC_Generate and MAC_Verify, verbs. The application program should not change the chaining vector information. The verb uses the chaining vector to carry information between procedure calls.

Figure B-19. Cipher, MAC_Generate, and MAC_Verify Chaining Vector Format		
Offset	**Length**	**Meaning**
00-07	8	The cryptographic Output Chaining Vector (OCV) of the service. When used with the MAC_Generate and MAC_Verify verbs, the OCV is enciphered as a cryptographic variable
08	1	The count of the bytes that are carried over and not processed (from 0 to 7)
09-15	7	The bytes that are carried over and left-justified
16	2	The token master-key verification pattern

Key Storage Records

Key storage exists as an online, Direct Access Storage Device (DASD)-resident data set for the storage of key records. Key records contain a key label, space for a key token, and control information. The first two records in key storage contain key-storage control information that includes the key verification information for the master key that is used to multiply-encipher the keys that are held in key storage.

Figure B-20 shows the format of the first record in the file header of the key storage file. This record contains the default master-key verification pattern, and part of the file description.

Figure B-21 on page B-15 shows the format of the second record in the file header of the key storage file. This record contains the rest of the file description for key storage.

Figure B-22 on page B-15 shows the format of the records that contain key tokens.

Figure B-20. Key Storage File Header, Record 1

Offset	Length	Meaning
00	04	The total length of this key record.
04	04	The record validation value.
08	64	The key label without separators. $$FORTRESS$REL01$MASTERKEYVERIFY$PATTERN .
72	15	The date and time of when this record was created. The date string consists of an 8 digit date and a 6 digit time (ccyymmddhhmmssz) where: • cc - century • yy - year • mm - month • dd - day • hh - Hour in 24 hour format (00-24). • mm - Minutes. • ss - Seconds. • z - String terminator (0x00)
87	15	The date and time of when this record was last updated. This field has the same format as the created date.
102	26	Reserved
128	01	An indicator that this is either an internal DES or PKA key token.
129	01	Reserved
130	02	Token length which is a value of 64.
132	04	Reserved
136	16	The master key verification pattern of the current master key in the cryptographic facility when this file was initialized.
152	24	The first 24 bytes of the file description (the remaining 40 bytes are stored in the second record).
176	12	Reserved
188	04	The token validation value. Bytes 128 through 191 are considered to be the 64 byte token.

Figure B-21. Key Storage File Header, Record 2		
Offset	**Length**	**Meaning**
00	04	The total length of this key record.
04	04	The record validation value.
08	64	The key label without separators. For the DES key storage file the key label is $$FORTRESSDESREL01KEYSTORAGE$FILE$HEADER . For the PKA key storage file the key label is $$FORTRESSPKAREL01KEYSTORAGE$FILE$HEADER .
72	15	The date and time of when this record was created. This field has the same format as the created date in Figure B-20.
87	15	The date and time of when this record was last updated. This field has the same format as the created date in Figure B-20.
102	26	Reserved
128	01	An indicator that this is either an internal DES or PKA key token.
129	01	Reserved
130	02	Token length which is a value of 64.
132	04	Reserved
136	40	The last 40 bytes of the file description (the first 24 bytes were stored in the first record).
176	12	Reserved
188	04	The token validation value. Bytes 128 through 191 are considered to be the 64 byte token.

Figure B-22. Key Record Format in Key Storage		
Offset	**Length**	**Meaning**
00	04	The total length of this key record.
04	04	The record validation value.
08	64	The key label without separators.
72	15	The date and time of when this record was created. This field has the same format as the created date in Figure B-20 on page B-14.
87	15	The date and time of when this record was last updated. This field has the same format as the created date in Figure B-20 on page B-14.
102	26	Reserved
128	??	A DES or PKA key token.

Key Record List Data Set

There are two Key_Record_List verbs, one for the DES key store and one for the PKA key store. Each creates an internal data set that contains information about specified key records in key storage. Both verbs return the list in a data set, KYRLT*nnn*.LST, where *nnn* is the numeric portion of the name and *nnn* starts at 001 and increments to 999 and then wraps back to 001. For the DES key store, the data set is stored in the subdirectory specified by the optional environmental variable, CSUDESLD. If CSUDESLD is not set, x:\KEYDIR is used where x is the current disk. For the PKA key store, the data set is stored in the subdirectory specified by the optional environmental variable, CSUPKALD. If CSUPKALD is not set, x:\PKADIR is used where x is the current disk. For information about the Key_Record_List verbs, see "Key_Record_List" 7-7.

The data set has a header record, followed by zero to *n* detail records, where *n* is the number of key records with matching key labels.

Figure B-23 (Page 1 of 2). Key Record List Data Set Format

Offset	Length	Meaning
Header Record (Part 1)		
0	24	This field contains the installation-configured listing header (the default value for the DES key store is DES KEY RECORD LIST and for the PKA key store is PKA KEY RECORD LIST).
24	2	This field contains spaces for separation.
26	19	This field contains the date and the time when the list was generated. The format is *ccyy-mm-dd hh:tt:ss*, where: *cc* Is the century *yy* Is the year *mm* Is the month *dd* Is the day *hh* Is the hour *tt* Is the minute *ss* Is the second. A space character separates the day and the hour.
45	5	This field contains spaces for separation.
50	6	This field contains the number of detail records.
56	2	This field contains spaces for separation.
58	4	This field contains the length of each detail record, in character form, and left-justified. (The length is 154.)
62	4	This field contains the offset to the first detail record, in character form, and left-justified. (The offset is 154.)
66	9	This field is reserved filled wilth space characters.
75	2	This field contains carriage return/line feed (CR/LF).
Header Record (Part 2)		
77	64	This field contains the key-label pattern that you used to request the list.
141	11	This field is reserved filled wilth space characters.
152	2	This field contains a carriage return or line feeds (CR/LF).

Figure B-23 (Page 2 of 2). Key Record List Data Set Format		
Offset	**Length**	**Meaning**
Detail Record (Part 1)		
0	1	This field contains an asterisk (*) if the key-storage record did not have a correct record validation value; this record should be considered to be a potential error.
1	2	This field contains spaces for separation.
3	64	This field contains the key label.
67	8	This field contains the key type. If a null key token exists in the record or if the key token does not contain the key value, this field is set to NO-KEY. For the DES key storage, if the key token does not contain a control vector, this field is set to NO-CV. If the control vector cannot be decoded to a recognized key type, this field is set to ERROR, and an asterisk (*) is set into the record at offset 0. For PKA key storage, the possible key types are: RSA-PRIV, RSA-PUBL, or RSA-OPT.
75	2	This field contains a carriage return or line feeds (CR/LF).
Detail Record (Part 2)		
77/0	4	For an internal token, this field will contain the Master key verification pattern in the token, else it is filled with space characters.
81/4	1	This field contains spaces for separation
82/5	8	Reserved, filled with space characters.
90/13	2	This field contains spaces for separation.
92/15	19	This field contains the date and time when the record was created. The format is *ccyy-mm-dd hh:tt:ss*, where: *cc* Is the century *yy* Is the year *mm* Is the month *dd* Is the day *hh* Is the hour *tt* Is the minute *ss* Is the second. A space character separates the day and the hour.
111/34	2	This field contains spaces for separation.
113/36	19	This field contains the last time and date when the record was updated. The format is *ccyy-mm-dd hh:tt:ss*, where: *cc* Is the century *yy* Is the year *mm* Is the month *dd* Is the day *hh* Is the hour *tt* Is the minute *ss* Is the second. A space character separates the day and the hour.
132/55	1	This field contains a space character for separation.
133/56	8	This field contains type of token, INTERNAL, EXTERNAL or NO-KEY (null token). Anything else, this field is set of ERROR and an asterisk (*) is set into the record offset 0 field.
141/64	11	Reserved, filled with space characters.
152/75	2	This field contains a carriage return (CR) or line feeds (LF).

Access Control Data Structures

The following sections define the data structures that are used in the access control system.

Unless otherwise noted, all two-byte and four-byte integers are in *big-endian* format; the high order byte of the value is in the lowest numbered address in memory.

Role Structure

This section describes the data structures used with roles.

Basic Structure of a Role

The following figure describes how the *Role* data is structured. This is the format used when role data is transferred to or from the coprocessor, using verbs CSUAACI or CSUAACM.

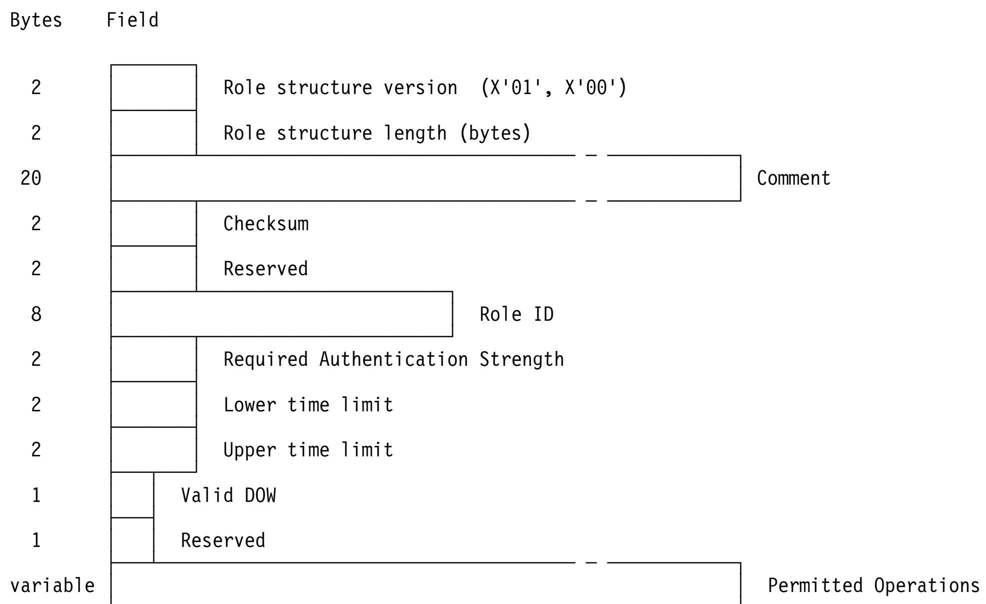

Figure B-24. Role layout

The *checksum* is defined as the exclusive-OR (XOR) of each byte in the role structure. The high-order byte of the checksum field is set to zero (X'00'), and the exclusive-OR result is put in the low-order byte.

Note: The checksum value is not used in the current role structure. It may be verified by the Cryptographic Coprocessor with a future version of the role structure.

The *Permitted Operations* are defined by the *Access Control Point list*, described in “The Access Control Point List” on page B-19 below.

The lower time limit and upper time limit fields are two-byte structures with each byte containing a binary value. The first byte contains the hour (0-23) and the second byte contains the minute (0-59). For example, 8:45 AM is represented by X'08' in the first byte, and X'2D' in the second.

If the lower time limit and upper time limit are identical, the role is valid for use at any time of the day.

The valid days-of-the-week are represented in a single byte with each bit representing a single day. Set the appropriate bit to one to validate a specific day. The first, or Most Significant Bit (MSB) represents Sunday, the second bit represents Monday, and so on. The last or Least Significant Bit (LSB) is reserved and must be set to zero.

Aggregate Role Structure

A set of one or more role definitions are sent in a single data structure. This structure consists of a *header*, followed by one or more role structures as defined in "Basic Structure of a Role" on page B-18.

The header defines the number of roles which follow in the rest of the structure. Its layout is shown in Figure B-25, with three concatenated role structures shown for illustration.

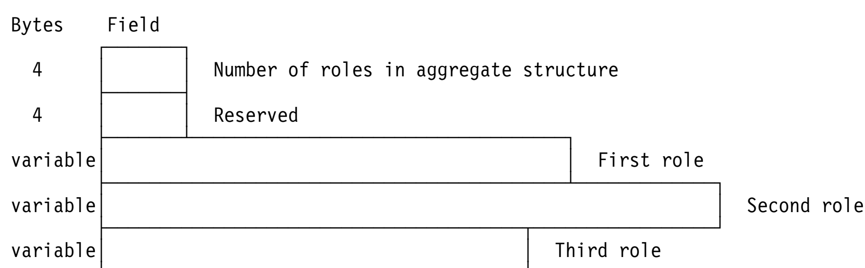

Figure B-25. Aggregate role structure with header

The Access Control Point List

The user's permissions are attached to each Role in the form of an *Access Control Point* list. This list is a map of bits, with one bit for each primitive function that can be independently controlled. If a bit is *True* (1), the user has the authority to use the corresponding function, if all other access conditions are also satisfied. If the bit is *False* (0), the user is not permitted to make use of the function that bit represents.

The access control point identifiers are two byte integers. This provides a total space of 64K possible bits. Only a small fraction of these are used, so storing the entire 64K bit (8K byte) table in each role would be an unnecessary waste of memory space. Instead, the table is stored as a sparse matrix, where only the necessary bits are included.

To accomplish this, each bitmap is stored as a series of one or more bitmap *segments*, where each can hold a variable number of bits. Each segment must start with a bit that is the high order bit in a byte, and each must end with a bit that is the low order bit in a byte. This restriction results in segments that have no partial bytes at the beginning or end. Any bits that do not represent defined access control points must be set to zero, indicating that the corresponding function is not permitted.

The bitmap portion of each segment is preceded by a header, providing information about the segment. The header contains the following fields.

Starting bit number The index of the first bit contained in the segment. The index of the first access control point in the table is zero (X'0000').

Ending bit number The index of the last bit contained in the segment.

Number of bytes in segment The number of bytes of bitmap data contained in this segment.

The entire access control point structure is comprised of a header, followed by one or more access control point segments. The header indicates how many segments are contained in the entire structure.

The layout of this structure is illustrated in Figure B-26.

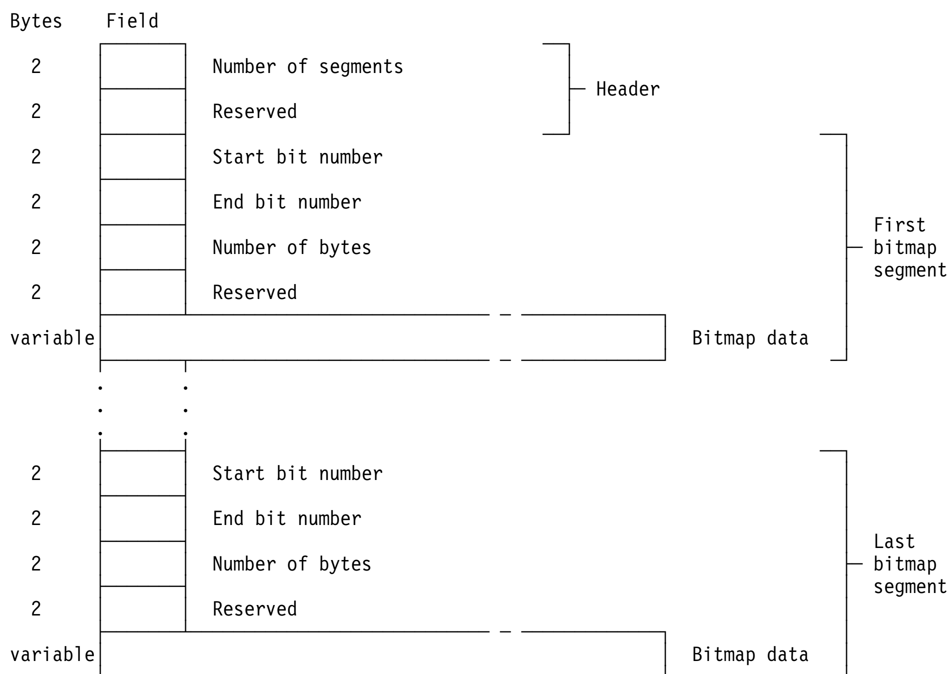

Figure B-26. Access control point structure

Contents of the Default Role

The default role will have the following characteristics.

- The role ID will be **DEFAULT**.
- The required authentication strength level will be zero.
- The role will be valid at all times and on all days of the week.
- The only functions that will be permitted are those related to access control initialization. This will guarantee that the owner will initialize the coprocessor before any useful cryptographic work can be done. This requirement prevents security "accidents" in which unrestricted default authority might accidentally be left intact when the system is put into service.

  The access control points that are enabled in the default role are shown in Figure B-27.

Figure B-27. Functions permitted in Default Role	
Code	**Function Name**
X'0107'	PKA96 One Way Hash
X'0110'	Set Clock
X'0111'	Reinitialize Device
X'0112'	Initialize access control system roles and profiles
X'0113'	Change the expiration date in a user profile
X'0114'	Change the authentication data (e.g. passphrase) in a user profile
X'0115'	Reset the logon failure count in a user profile
X'0116'	Read public access control information
X'0117'	Delete a user profile
X'0118'	Delete a role

Profile Structure

This section describes the data structures related to user profiles.

Basic Structure of a Profile

The following figures describe how the *Profile* data is structured. This is the format used when profile data is transferred to or from the coprocessor, using verbs Access_Control_Initialization or Access_Control_Maintenance.

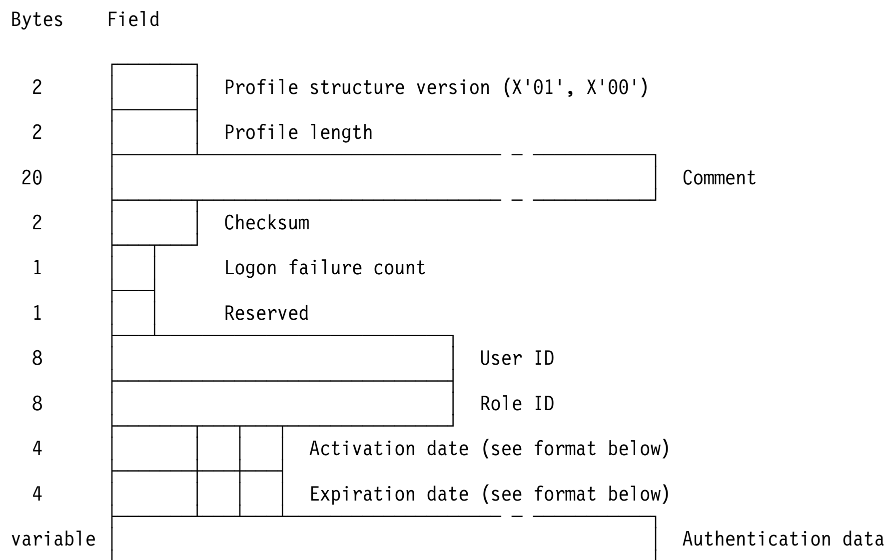

Figure B-28. Profile layout

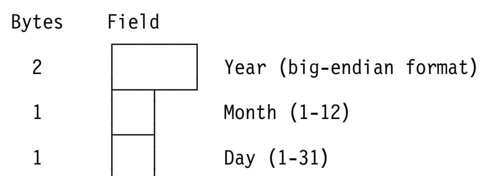

Figure B-29. Layout of profile Activation and Expiration dates

When a new profile is loaded, the host application does not provide the *Logon failure count* value. This field is automatically set to zero when the profile is stored in the coprocessor. The failure count field should have a value of zero in the initialization data you send with Access_Control_Initialization.

The *checksum* is defined as the exclusive-OR (XOR) of each byte in the profile structure. The high-order byte of the checksum field is set to zero (X'00'), and the exclusive-OR result is put in the low-order byte.

Note: The checksum value is not used in the current profile structure. It may be verified by the Cryptographic Coprocessor with a future version of the profile structure.

Aggregate Profile Structure

For initialization, a set of one or more profile definitions are sent to the coprocessor together, in a single data structure. This structure consists of a *header*, followed by one or more profile structures as defined in "Profile Structure" on page B-21.

The header defines the number of profiles which follow in the rest of the structure. Its layout is shown in Figure B-30, with three concatenated profile structures shown for illustration.

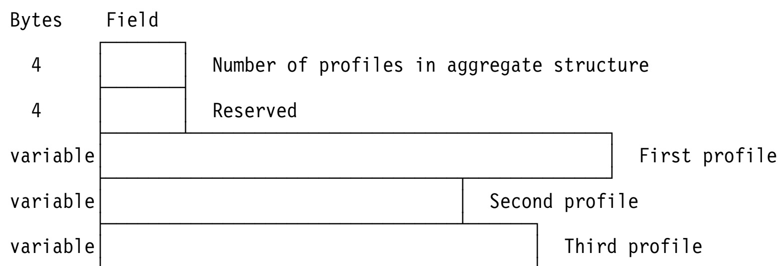

Figure B-30. Aggregate profile structure with header

The Authentication Data Structure

This section describes the authentication data, which is part of each user profile. Authentication data is the information the coprocessor uses to verify your identity when you log on.

There are two versions of the authentication data structure, corresponding to profiles versions 1.0 and 1.1. The only difference is in the meaning of the length field, as described below.

General Structure of Authentication Data: The Authentication Data field is a series of one or more Authentication Data structures, each containing the data and parameters for a single authentication method. The field begins with a header, which contains two data elements.

Length A two-byte integer value defining how many bytes of authentication information are in the strucure. For profile structure version 1.0, the Length includes all bytes after the Length field itself. For profile structure version 1.1, the Length includes all bytes after the *header*, where the header includes both the Length field and the Field Type Identifier field.

Field Type Identifier A two-byte integer value which identifies the type of data following the header. The identifier must be set to the integer value X'0001', which indicates that the data is of type "*Authentication Data*."

The header is followed by individual sets of authentication data, each containing the data for one authentication mechanism. This layout is shown pictorially in Figure B-31.

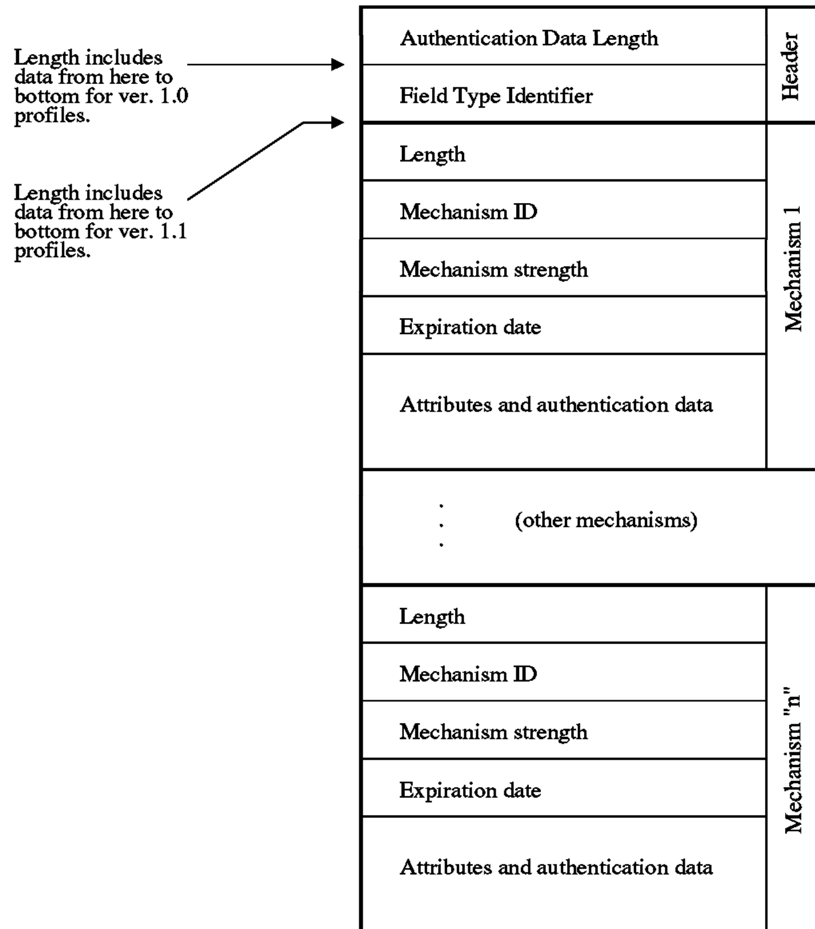

Figure B-31. Layout of the Authentication Data field

The content of the individual Authentication Data structures is shown in Figure B-32 below.

Figure B-32 (Page 1 of 2). Authentication Data for each authentication mechanism

<table>
<tr><th>Field name</th><th>Length (bytes)</th><th>Description</th></tr>
<tr><td>Length</td><td>2</td><td>The size of this set of authentication mechanism data, in bytes. See page B-22 for a description of this value, which differs for profile structure versions 1.0 and 1.1.</td></tr>
<tr><td>Mechanism ID</td><td>2</td><td>An identifier which describes the authentication mechanism associated with this set of data. For example, there might be identifiers for passphrase, PIN, fingerprint, public-key based identification, and others. This is an integer value.

For passphrase authentication, the mechanism ID is the integer value X'0001'.</td></tr>
<tr><td>Mechanism strength</td><td>2</td><td>An integer value which defines the strength of this identification mechanism, relative to all others. Higher values reflect greater strength. A value of zero is reserved for users who have not been authenticated in any way.</td></tr>
<tr><td>Expiration date</td><td>4</td><td>The last date on which this authentication data may be used to identify the user. The field contains the month, day, and year of expiration. All four digits of the year are stored, so that no problems occur at the turn of the century.

The expiration date is a four-byte structure, as shown in the C type definition below.

typedef struct {
   unsigned char exp_year[2];
   unsigned char exp_month;
   unsigned char exp_day;
} expiration_date_t;

The two-byte exp_year is in big-endian format. The high-order byte is at the lower numbered address.</td></tr>
<tr><td>Mechanism attributes</td><td>4</td><td>This field contains flags and attributes needed to fully describe the operation and use of of the authentication mechanism. One flag is defined for all methods:

Renewable A Boolean value which indicates whether the user is permitted to renew the authentication data. If this value is True (1), the user can renew the data by authenticating, and then providing new authentication data. For example, to replace a passphrase, the user would first log on using his or her passphrase. Then, the passphrase would be changed by providing the new passphrase authentication data using the Access_Control_Initialization verb with the CHG-AD rule-array keyword. The format of the passphrase authentication data is described immediately below under ‘mechanism data’.

The Renewable bit is the most-significant bit (MSB) in the four-byte attributes field. The other 31 bits are unused, and must be set to zero.</td></tr>
</table>

Figure B-32 (Page 2 of 2). Authentication Data for each authentication mechanism

Field name	Length (bytes)	Description
Mechanism data	variable	This field contains the data needed to perform the authentication. The size, content, and complexity of this data will vary according to the authentication mechanism. For example, the content could be as simple as a password that is compared to one entered by the user, or it could be as complex as a set of sophisticated biometric reference data, or a public key certificate.

Authentication Data for Passphrase Authentication: For passphrase authentication, the mechanism data field contains the 20-byte SHA-1 hash of the user's passphrase. The hash is computed in the host, where it is used to construct the profile that is downloaded to the Leeds card.

Examples of the data structures

Passphrase authentication data

Figure B-33 shows the contents of a sample authentication mechanism data structure for a passphrase.

```
00 20 00 01 01 80 07 ce 06 01 80 00 00 00 fb f5     . ..............
c4 84 75 5f ba 59 6b ca 4a 9d ca 08 fb 52 9e e2     ..u_.Yk.J....R..
45 41                                               EA
```

Figure B-33. Passphrase authentication data structure

This data breaks down into the following fields.

00 20 The length of the authentication mechanism data, excluding the length field itself. (32 bytes)

Note: The example is for a version 1.0 profile structure. For version 1.1, the length would be X'1E', or decimal 30.

00 01 The mechanism identifier, for *Passphrase Authentication Data*.

01 80 The mechanism strength. Hex 0180, or decimal 384.

07 CE The year of the passphrase expiration date. Hex 07CE, or decimal 1998.

06 01 The month and year of the passphrase expiration date. This represents June 1.

80 00 00 00 The mechanism attributes. The *Renewable* bit is set.

FB F5 C4 84 75 5F BA 59 6B CA 4A 9D CA 08 FB 52 9E E2 45 41 The authentication data. This 20-byte value is the SHA-1 hash of the user's passphrase. In this case, the passphrase is "*This is my passphrase.*."

User profile

Figure B-34 shows the contents of an entire user profile, containing the passphrase data shown above.

```
01 00 00 5a 2d 20 53 61 6d 70 6c 65 20 50 72 6f     ...Z- Sample Pro
66 69 6c 65 20 31 20 2d ab cd 00 00 4a 5f 53 6d     file 1 -....J_Sm
69 74 68 20 41 44 4d 49 4e 31 20 20 07 cd 06 01     ith ADMIN1  ....
07 cd 0c 1f 00 22 00 01 00 20 00 01 01 80 07 ce     ....."... ......
06 01 80 00 00 00 fb f5 c4 84 75 5f ba 59 6b ca     ..........u_.Yk.
4a 9d ca 08 fb 52 9e e2 45 41                       J....R..EA
```

Figure B-34. User profile data structure

This user profile contains the following fields.

01 00 The profile structure version number. For a version 1.1 profile structure, this would have the value **01 01**.

00 5A The length of the profile, including the length field itself. Hex 5A is equal to decimal 90.

"- Sample Profile 1 -" The 20 character comment for this user profile.

AB CD The checksum for the user profile.

Note: The checksum value is not used. In future versions of the profile structure, the checksum may be verified in the Cryptographic Coprocessor.

00 The logon failure count.

00 Reserved field, which must be zero.

"J_Smith " The user ID for this profile.

"ADMIN1 " The role that will define the authority associated with this profile.

07 CD The year of the profile's activation date. Hex 07CD is equal to decimal 1997.

06 01 The month and day of the profile's activation date. This represents June 1.

07 CD The year of the profile's expiration date. Hex 07CD is equal to decimal 1997.

0C 1F the month and day of the profile's expiration date. Hex 0C is equal to decimal 12, and hex 1F is equal to decimal 31, so the profile expires on December 31.

00 22 The total length of all the authentication data for this profile, not including the length of this field itself.

00 01 The field type identifier, indicating that the following data is *Authentication Data*.

Passphrase data The remainder of the field is the passphrase data structure, as described above.

Aggregate profile structure

Figure B-35 shows the aggregate profile structure, containing one user profile. This is the structure that is passed to the CSUAACI verb in order to load one or more user profiles.

```
00 00 00 01 00 00 00 00 01 00 00 5a 2d 20 53 61     ...........Z- Sa
6d 70 6c 65 20 50 72 6f 66 69 6c 65 20 31 20 2d     mple Profile 1 -
ab cd 00 00 4a 5f 53 6d 69 74 68 20 41 44 4d 49     ....J_Smith ADMI
4e 31 20 20 07 cd 06 01 07 cd 0c 1f 00 22 00 01     N1  ........."..
00 20 00 01 01 80 07 ce 06 01 80 00 00 00 fb f5     . .............
c4 84 75 5f ba 59 6b ca 4a 9d ca 08 fb 52 9e e2     ..u_.Yk.J....R..
45 41                                               EA
```

Figure B-35. Aggregate profile structure

This structure contains the following data fields.

00 00 00 01 The number of profiles that are in the aggregate structure. This example contains only one user profile, but any number can be included in the same aggregate structure.

00 00 00 00 A reserved field, which must contain zeroes.

User profile The remainder of this structure contains the single user profile that was described earlier in this section.

Access control point list

Figure B-36 shows the contents of a sample Access Control Point List.

```
00 02 00 00 00 00 01 17 00 23 00 00 f0 ff ff ff     .........#......
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff     ................
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 02     ................
00 02 17 00 03 00 00 8f 99 fe                       ..........
```

Figure B-36. Access Control Point List

The Access Control Point list contains the following data fields.

00 02 The number of segments of data in the access control point list. In this list, there are two discontiguous segments of access control points. One starts at access control point 0, and the other starts at access control point X'200'.

00 00 A reserved field, which must be filled with zeroes.

00 00 The number of the first access control point in this segment.

01 17 The number of the last access control point in this segment. The segment starts at access control point 0, and ends with access control point X'117', which is decimal 279.

00 23 The number of bytes of data in the access control points for this segment. There are X'23' bytes, which is 35 decimal.

00 00 A reserved field, which must be filled with zeroes.

F0 FF FF FF ... FF FF (35 bytes) This is the first set of access control points, with one bit corresponding to each point. Thus, the first byte contains bits 0-7, the next byte contains 8-15, and so on.

02 00 The number of the first access control point in the second segment.

02 17 The number of the last access control point in this segment. The segment starts at access control point X'200' (decimal 512), and ends with access control point X'217' (decimal 535).

00 03 The number of bytes of data in the access control points for this segment. There are 3 bytes, for the access control points from 512 through 535.

00 00 A reserved field, which must be filled with zeroes.

8F 99 FE This is the second set of access control points, with one bit corresponding to each point. Thus, the first byte contains bits 512-519, the second byte contains 520-527, and the third byte contains 528-535.

Role data structure

Figure B-37 shows the contents of a role data structure.

```
01 00 00 62 2a 4e 65 77 20 64 65 66 61 75 6c 74     ....*New default
20 72 6f 6c 65 20 31 2a ab cd 00 00 44 45 46 41      role 1*....DEFA
55 4c 54 20 23 45 01 0f 17 1e 7c 00 00 02 00 00     ULT #E....|.....
00 00 01 17 00 23 00 00 f0 ff ff ff ff ff ff ff     .....#..........
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff     ................
ff ff ff ff ff ff ff ff ff ff ff 02 00 02 17 8f     ................
99 fe                                               ..
```

Figure B-37. Role data structure

This structure contains the following data fields.

00 01 The role structure version number.

00 62 The length of the role structure, including the length field itself.

"*New default role 1*" The 20 character comment describing this role.

AB CD The checksum for the role.

Note: The checksum value is not used. In future versions of the role structure, the checksum may be verified in the Cryptographic Coprocessor.

00 00 A reserved field, which must be filled with zeroes.

"DEFAULT " The Role ID for this role. The role in this example will replace the DEFAULT role.

23 45 The Required Authentication Strength field

01 0F The lower time limit. X'01' is the hour, and X'0F' is the minute (decimal 15), so the lower time limit is 1:15 AM, GMT.

17 1E The upper time limit. X'17' is the hour (decimal 23), and X'1E' is the minute (30), so the upper time limit is 23:30 GMT.

7C This byte maps the valid days of the week for the role. The first bit represents Sunday, the second represents Monday, and so on. Hex 7C is binary 01111100, and enables the weekdays Monday through Friday.

00 This byte is a reserved field, and must be zero.

Access control point list The remainder of the role structure contains the Access Control Point list described above.

Aggregate role data structure

Figure B-38 shows the an aggregate role data structure, like you would load using the CSUAACI verb.

```
00 00 00 01 00 00 00 00 01 00 00 62 2a 4e 65 77     ............*New
20 64 65 66 61 75 6c 74 20 72 6f 6c 65 20 31 2a      default role 1*
ab cd 00 00 44 45 46 41 55 4c 54 20 23 45 01 0f     ....DEFAULT #E..
17 1e 7c 00 00 02 00 00 00 00 01 17 00 23 00 00     ..|..........#..
f0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff     ................
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff     ................
ff ff ff 02 00 02 17 8f 99 fe                       ..........
```

Figure B-38. Aggregate role data structure

This structure contains the following data fields.

00 00 00 01 The number of roles that are in the aggregate structure. This example contains only one role, but any number can be included in the same aggregate structure.

00 00 00 00 A reserved field, which must contain zeroes.

Role data structure The remainder of the aggregate structure contains the role structure, which was described above.

Master Key Shares Data Formats

Master key shares, and potentially other information to be "cloned" from one coprocessor to another coprocessor are packed into a data structure as described in Figure B-39.

Figure B-39. Cloning Information Token Data Structure

Offset (Bytes)	Length (Bytes)	Description
000	001	X'1D', token identifier
001	001	X'00', Version
002	002	Length of the cloning information token
004	004	Reserved, binary zero
008	004	Cloning-share index number, i; 1≤i≤15
012	016	Origin-node Environment Identifier, EID
028	008	Origin-coprocessor serial number
036	xxx	Cloning information TLV's: • Master key share • Signature And one to seven bytes of padding to ensure that length 'xxx' is a multiple of eight bytes.
Note: The information from offset 036 through 035+xxx is triple encrypted with a triple-length DES key using the EDE3 encryption process, see "Triple-DES Ciphering Algorithms" on page D-11.		

Figure B-40. Master Key Share TLV

Offset (Bytes)	Length (Bytes)	Description
000	001	X'01', master key share identifier
001	001	X'00', Version
002	002	X'001D', length of the TLV
004	001	Index value, i, binary
005	024	Master-key share

Figure B-41. Cloning Information Signature TLV

Offset (Bytes)	Length (Bytes)	Description
000	001	X'45', Signature Subsection Header
001	001	X'00', Version
002	002	Subsection length, 70+sss
004	001	Hashing algorithm identifier; X'01' signifies use of the SHA-1 hashing algorithm.
005	001	Signature formatting identifier; X'01' signifies use of the ISO-9796 process.
006	064	Signature-key identifier; the key label of the key used to generate the signature.
070	sss	The signature field. The signature is calculated on data that begins with the Cloning Information Token Data Structure identifier (X'1D') through the byte immediately preceding this signature field.

Function Control Vector

The export (distribution) of cryptographic implementations by USA companies is controlled under USA Government export regulations. An IBM 4758 becomes a practical cryptographic engine when it accepts and validates digitally signed software. IBM has chosen to export the IBM 4758 as a non-cryptographic product, and to control and report the export of the cryptography-enabling software.

The CCA software that can be loaded into the Coprocessor limits the functionality of the Coprocessor based on the values in a *function control vector* (FCV). At the present time, two capabilities are controlled:

- Use of 56-bit keys with the DES algorithm for general data encryption
- The length of an RSA key used to encipher DES keys.

Notes:

1. Government policies and the FCV do not limit the key-length of keys used in digital signature operations.
2. The SET services can employ 56-bit DES for data encryption, and 1024-bit RSA key-lengths when distributing DES keys.

IBM distributes the FCV in a digitally signed data structure. Figure B-42 displays the format of the data structure that contains the function control vector as distributed by IBM.

Figure B-42. FCV Distribution Structure

Offset Decimal (Hex)	Length Decimal	Meaning
000 (000)	390	Package header and validating-key certificate
390 (186)	080	Descriptive text coded in ASCII
470 (1D6)	204	Function control vector (FCV) This is the information that can be supplied to the Coprocessor using the Cryptographic_Facility_Control verb.
674 (2A2)	128	Digital signature on the complete structure (excepting this signature itself).

Appendix C. CCA Control Vector Definitions and Key Encryption

This appendix describes the following:

- DES control vector values[1]
- Specifying a control vector base value
- CCA key encryption and decryption processes.

In the Common Cryptographic Architecture (CCA), a control vector is a non-secret quantity that expresses permissible usages for an associated key. When a CCA DES key is encrypted, the key-encrypting key is exclusive-ORed with the control vector to form the actual key used in the DES key-encrypting process. This technique allows the generator or introducer of a key to specify how the key is to be distributed and used. Attacks can be mounted against a cryptographic system when it is possible to use a key for other than its intended purpose. The CCA control vector key-typing scheme and the command authorization and control vector checking performed by a CCA node together provide an important defense against misuse of keys and related attacks.

DES Control Vector Values

The CCA key token includes the control vector and the key that the control vector describes. The control vector is as long as the key, either 64 or 128-bits in length. The control vector is "coupled" to the key because it modifies the key-encrypting key value used to encrypt the key found in the key token. See "CCA DES Key Encryption and Decryption Process" on page C-8.

Although the CCA architecture permits several advanced techniques, the product implementations described in this book use the same control vector value for the second half of a double length key as for the first half...except for the reversal of two bits. Therefore this discussion of control vector values focuses on a 64-bit vector with the understanding that, for a double-length key, the control vector value associated with each key half is essentially the same.

Most of the first 16 bits of a control vector define the key as belonging to one of several general (*generic*) classes of keys as shown in the following list:

Key-Encrypting Keys:

IMPORTER Used to decrypt a key brought to this local node

EXPORTER Used to encrypt a key taken from this local node

IKEYXLAT Used to decrypt an input key in the Key_Translate service

OKEYXLAT Used to encrypt an output key in the Key_Translate service.

[1] In this appendix, *control vector* means DES control vector base unless noted otherwise. This document does not include information about encoding a control vector extension.

© Copyright IBM Corp. 1997-98

Data keys:

DATA	Used to encrypt or decrypt data, or to generate or verify a MAC
MAC	Used to generate or verify a MAC
MACVER	Used to verify a MAC code (cannot be used in MAC-generation).

PIN-processing keys:

PINGEN	Used to generate and verify PIN values
PINVER	Used to verify PIN values (can not be used in PIN-generation)
OPINENC	Used to encrypt a PIN-block
IPINENC	Used to decrypt a PIN-block.

Key-generating keys:

Generate[2]	Used to generate or derive other keys

There is a default control vector associated with each of the generic key types just listed; see Figure C-1 on page C-3. The bits in positions 16-22 and 33-37 generally have different meanings for every generic key class. Many of the remaining bits in a control vector have a common meaning. Most of the DES key-management services permit you to use the default control vector value by naming the generic key class in the service's key-type variable; *this does not apply to all generic key-type classes.*[3]

You can use the default control vector for a generic key type, or you can create a more restrictive control vector. The default control vector for a generic key type provides basic key-separation functions. The cryptographic subsystem creates a default control vector for a generic key type when you use the Key_Generate verb and specify a null key token and a generic key-type in the *key_type* parameter. When you import or export a key, you can also specify a key type to obtain a default control vector instead of supplying a control vector in a key token. If you specify a key type with the Key_import verb, ensure that the default control vector is the same as the control vector that was used to encrypt the key.

The additional control vector bits that you can turn on (beyond those already on in the generic control vector value) permit you to further restrict the use of a key. This gives you the ability to implement the general security policy of permitting only those capabilities actually required in a system. The additional bits are designed to block specific attacks although these attacks are almost always very obscure.

You can obtain the value for a control vector in one of several ways:

- Use a generic control vector and obtain the value from Figure C-1 on page C-3.
- See "Specifying a Control Vector Base Value" on page C-5. The material presents an ordered set of questions to enable you to create the value for a control vector.

---

[2] When generating, importing, or exporting the Generate key-type, this key-type must be requested through the specification of a proper control vector in a key token and the use of the **TOKEN** keyword.

[3] The Key_Token_Build verb has not been extended to support some key-type classes, for example: PINGEN, PINVER, IPINENC, OPINENC, etc.

- For some of the key types, you can use the Key_Token_Build verb and keywords to construct a control vector and incorporate this control vector into a key token.

Figure C-1. Control Vector Default Values for Generic Key Types

Key Type	**Control Vector Hexadecimal Value for Single-length Key or Left Half of Double-Length Key**	**Control Vector Hexadecimal Value for Right Half of Double-Length Key**
DATA (Internal) DATA (External)	00 00 7D 00 03 00 00 00 00 00 00 00 00 00 00 00	
EXPORTER	00 41 7D 00 03 41 00 00	00 41 7D 00 03 21 00 00
IKEYXLAT	00 42 42 00 03 41 00 00	00 42 42 00 03 21 00 00
IMPORTER	00 42 7D 00 03 41 00 00	00 42 7D 00 03 21 00 00
IPINENC	00 21 5F 00 03 41 00 00	00 21 5F 00 03 21 00 00
MAC	00 05 4D 00 03 00 00 00	
MACVER	00 05 44 00 03 00 00 00	
OKEYXLAT	00 41 42 00 03 41 00 00	00 41 42 00 03 21 00 00
OPINENC	00 24 77 00 03 41 00 00	00 24 77 00 03 21 00 00
PINGEN	00 22 7E 00 03 41 00 00	00 22 7E 00 03 21 00 00
PINVER	00 22 42 00 03 41 00 00	00 22 42 00 03 21 00 00
Generate[2]	00 53 50 00 03 41 00 00	00 53 50 00 03 21 00 00

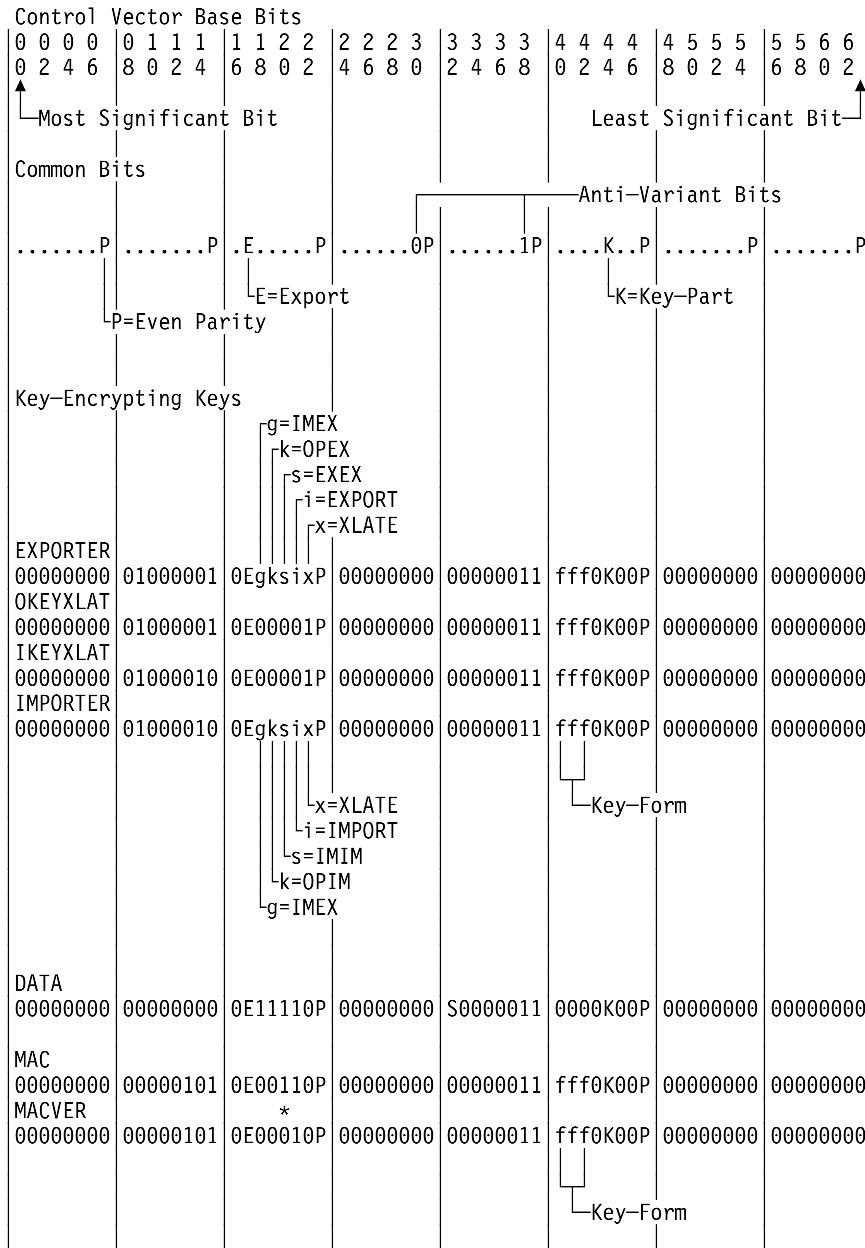

Figure C-2 (Part 1 of 2). Control Vector Base Bit Map

```
Control Vector Base Bits
0 0 0 0 |0 1 1 1 |1 1 2 2 |2 2 2 3 |3 3 3 3 |4 4 4 4 |4 5 5 5 |5 5 6 6
0 2 4 6 |8 0 2 4 |6 8 0 2 |4 6 8 0 |2 4 6 8 |0 2 4 6 |8 0 2 4 |6 8 0 2

PIN Keys

0000 NO-SPEC                Prohibit offsets:
0001 IBM-PIN/IBM-PINO       NOOFFSET
0101 NL-PIN-1

        PINGEN

aaaa000P|00100010|0E.....P|0000000P|S0000o1P|fff0K00P|00000000|0000000P
         CPINGEN
         CPINGENA
         EPINVER

        PINVER
aaaa000P|00100010|0E00001P|0000000P|S0000o1P|fff0K00P|00000000|0000000P

                           EPINVER
                           CPINGENA
IPINENC
0000000P|00100001|0E0..trP|00000000|S000001P|fff0K00P|0000000P|0000000P

OPINENC
0000000P|00100100|0E..0trP|00000000|S000001P|fff0K00P|0000000P|0000000P
                           REFORMAT
                           TRANSLAT
                           EPINGEN
                           CPINENC

Key-Generating Key (for generating diversified keys in CSNBGDK)
00000000|01010011|0E01000P|00000000|00000011|0100K00P|00000000|00000000
```

Figure C-2 (Part 2 of 2). Control Vector Base Bit Map

Key Form Bits, 'fff'

The key form bits, 40-42 ...and for a double-length key, bits 104-106... are designated 'fff' in the preceding diagram. These bits can have these values:

000 Single length key
010 Double length key, left half
001 Double length key, right half

And these values in some CCA implementations although not created in the IBM 4758 implementation:

110 Double length key, left half, halves guarenteed unique
101 Double length key, right half, halves guarenteed 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 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:

Bits 8 to 11	Main Key Type
0000	Data operation keys
0010	PIN keys
0100	Key-encrypting keys
0101	Key-generating keys

- The key subtype bits (bits 12 to 14). Set bits 12 to 14 to one of the following values:

Bits 12 to 14	Key Subtype
Data Operation Keys	
010	MAC key (MAC or MACVER)
000	Compatibility key (DATA)
Key-Encrypting Keys	
000	Transport-sending keys (EXPORTER and OKEYXLAT)
001	Transport-receiving keys (IMPORTER and IKEYXLAT)
PIN Keys	
001	PIN-generating key (PINGEN, PINVER)
000	Inbound PIN-block decrypting key (IPINENC)
010	Outbound PIN-block encrypting key (OPINENC)
Key-Generating Keys	
001	Key-generating keys

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 that 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 C-2 on page C-4). 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 that 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 C-8.
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 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.
0101	NL-PIN-1	A key with this control vector can be used only with the NL-PIN-1, Netherlands 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 to 22 to indicate whether the key can be used with the following verbs; for the bit numbers, see Figure C-2 on page C-4:

Verb Allowed	Bit Name	Bit
Clear_PIN_Generate	CPINGEN	
Clear_PIN_Generate_Alternate	CPINGENA	21 for PINGEN 20 for IPINENC
Encrypted_Pin_Verify	EPINVER	19
Clear_PIN_Encrypt	CPINENC	18

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.
   - Set the REFORMAT bit (r, bit 22) to 1 to permit the key to be used in the PIN_Translate verb.
   - Set PIN-block format bits (bbbbbb, bits 49 to 54) to one of the values in the following table. For more information about these bits, see “Processing Financial PINs” on page 8-1.

Bits 49 to 54	Control_Vector_ Generate Keyword	PIN Block Format
000000		Any format
000001	3624	IBM 3624
000100	ISO-0	ISO 0 (equivalent to ANSI X9.8, VISA format 1, and ECI 1 formats)

7. For key-generating keys, set the following bits:
   - Set bit 19 to 1 if the key will be used in the Diversified_Key_Generate (CSNBDKG) verb to generate a diversified key.
8. 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.
   - 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 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 exclusive-ORed 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.

CCA Key Encryption and Decryption Process

This section describes the CCA key encryption processes:

- CCA DES key encryption
- CCA RSA private key encryption.

CCA DES Key Encryption and Decryption Process

With the CCA, multiply-enciphering or deciphering a key is a two-step process. The implementation first exclusive-ORs the subject key's control vector with the master key or with a key-encrypting key to form keys K1 through K6. The resulting keys (Kn) are used in the multiple-encipherment of a clear key, or the multiple-decipherment of an encrypted key; see Figure C-3 on page C-9 for the formation of K1 through K6 and their use with DES DEA encoding and decoding.

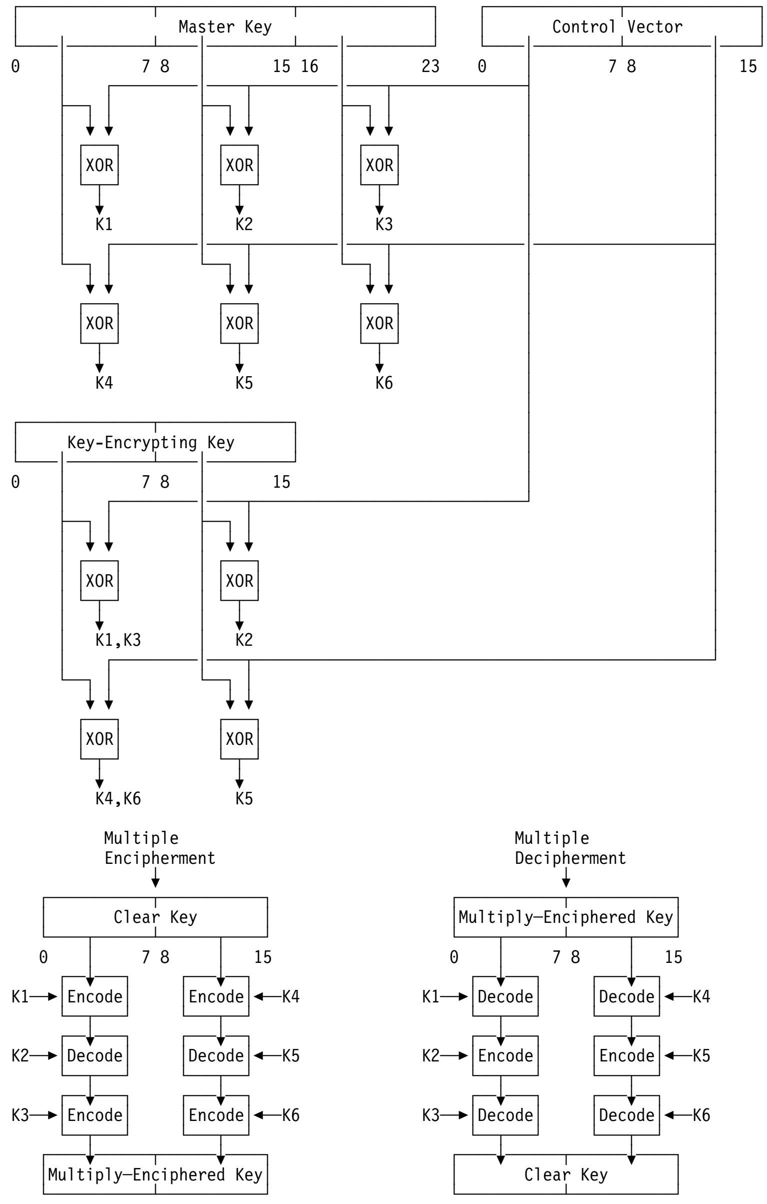

Figure C-3. Multiply-Enciphering and Multiply-Deciphering CCA Keys

Notes:

1. The encode and decode processes are the DES Electronic Code Book (ECB) processes for ciphering 64 data bits using a single-length key, K_n.
2. A CCA cryptographic implementation processes a single-length key in the same way as it processes the left half of a double-length key.
3. If the left and right halves of a double-length key-encrypting key have the same value, using the key in multiple-encipherment or multiple-decipherment of a key is equal to single-encipherment or single-decipherment of a key.
4. The control vector for a double-length key consists of two halves. The second half is the same as the first half except for bits 41 and 42, which are reversed in value.

CCA RSA Private Key Encryption and Decryption Process

Private keys in PKA96 implementations use the *EDE2* process to encipher the secret portion of an RSA private key in an external key token encrypted by a transport key-encrypting key. See Figure D-8 on page D-11. A private key in an internal key token encrypted by the master key is encrypted using the EDE3 process. See Figure D-10 on page D-13. The secret key is deciphered using the DED2 and DED3 processes. See Figure D-9 on page D-12 and Figure D-11 on page D-14.

The EDE2 algorithm uses a 112-bit key to encrypt any number of 64-bit blocks of information. The DED2 algorithm is used to decrypt this information. The key-encrypting key is a transport key-encrypting key for an external key.

The EDE3 algorithm uses a 168-bit key to encrypt any number of 64-bit blocks of information. The DED3 algorithm is used to decrypt this information. The key-encrypting key is the master key for an internal key.

Changing Control Vectors

Use the pre-exclusive-OR technique to change a key's control vector when exporting or importing the key from or to a cryptographic node. By exclusive-ORing information with the KEK used to import or export the key, you can effectively change the control vector associated with the key.

The pre-exclusive-OR technique requires exclusive-ORing additional information into the value of the importer or exporter KEK by one of the following methods:

- Exchange the KEK in the form of a plaintext value or in the form of key parts. For example, if you use the Key_Part_Import verb to enter the KEK key parts, you can enter another part that is set to the value of the *pre-exclusive-OR quantity*.
- Use the Key_Generate verb to generate an IMPORTER/EXPORTER pair of KEKs, with the key-part control vector bit set on. Then use the Key_Part_Import verb to enter an additional key part that is set to the value of the *pre-exclusive-OR quantity*.

To understand how you can change a key's control vector when importing or exporting keys, you must first understand the importing and exporting process. For example, when exporting key **K**, the cryptogram $e^*Km \oplus CV_k(K)$ is changed to the cryptogram $e^*KEK \oplus CV_{k1}(K)$.

Notes:

1. The first cryptogram is read as “the multiple-encipherment of key K by the key formed from the exclusive-OR of the master key and the control vector, CV_k, of key K.”
2. The second cryptogram is read as “the multiple-encipherment of key K by the key formed from the exclusive-OR of the KEK and the control vector, CV_{k1}, of key K.” KEK represents the value of the exporter key.
3. A control vector of value binary zero is equivalent to not having a control vector.

The CCA specifies that in all but one case, CV_k is the same as CV_{k1}. The exception is that a DATA key whose CV_k contains the value of a default CV for that key type, has a CV_{k1} equal to binary zero. (Key importing and exporting performed by the Personal Security Card does not obey this exception; for the card, CV_k is always equal to CV_{k1}.)

To change the control vector on key K, the KEK must be set to the value:

$$KEK \oplus CV_{k1} \oplus CV_{k2}$$

where:

- KEK is the value of the shared exporter key.
- $\oplus$ represents exclusive-OR.
- CV_{k1} is the control vector value used with the operational key K at the local node.
- CV_{k2} is the desired control vector value for the exported key K.

This process works because the value CV_{k1} is specified in the key token for the exported key. The Key_Export verb provides this control-vector value to the hardware, which exclusive-ORs it with the exporter KEK. However, you have set the exporter KEK to the value $KEK \oplus CV_{k1}$.... When CV_{k1} is exclusive-ORed with CV_{k1}, the effect is that CV_{k1} is removed. Because you also set the KEK to include the desired control vector, CV_{k2}, the exported key will have a changed control vector.

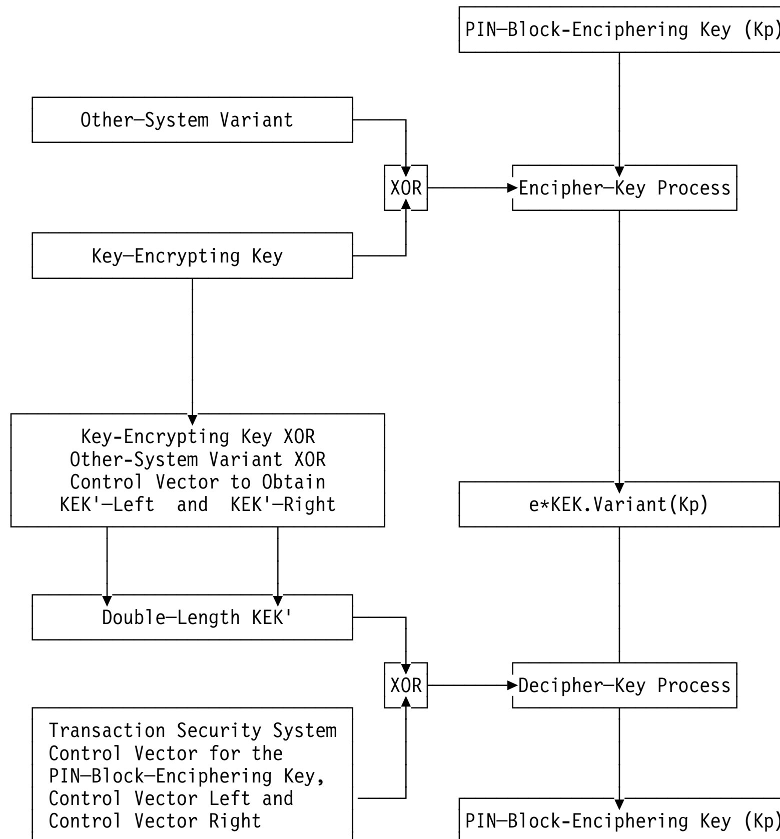

Figure C-4. Exchanging a Key with a Non-Control-Vector System

Appendix D. Algorithms and Processes

This appendix provides processing details for the following aspects of the CCA design:

- Cryptographic key-verification techniques
- Ciphering methods
- MAC calculation methods
- Multiple encipherment of DES keys with a control vector
- Triple-DES algorithms, EDE2 and EDE3
- Access control algorithms
- Encipherment of DES keys under RSA in “PKA92” format
- Encipherment of a DES key-encrypting key under RSA in “NL-EPP-5” format
- Master key splitting algorithm
- RSA key-pair generation.

Cryptographic Key Verification Techniques

The CCA implementations described in this book employ mechanisms for assuring the integrity and/or value of the key. These subjects are discussed:

- Master key verification algorithm
- DES key and key-part verification algorithm
- Encrypt zeros algorithm.

Master Key Verification Algorithm

The Fortress product family implementations employ a “triple-length” master key (3 DES keys) that is internally represented in 24 bytes. 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.

A SHA-1 hash is calculated on the quantity X'01' prepended to the 24-byte register contents. Then the high-order 8 bytes (0...7) of the 20-byte SHA-1 hash are returned in the *random_number* variable from a Key_Test verb call. The next eight-bytes of the SHA-1 hash (8...15) are returned in the *verification_pattern* variable.

The master key verification pattern used in an internal DES key record is calculated in a similar manner with the high-order two bytes of the SHA-1 used as the verification pattern (MKVP).

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.

© Copyright IBM Corp. 1997-98

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 computationally infeasible to compute the input given an output.

For information about how you can use an application program to invoke this verification method, see page 5-35.

The CCA DES key verification algorithm does the following:

1. Sets *KKR′* = *KKR* exclusive-OR *RN*
2. Sets *K1* = X'4545454545454545'
3. Sets *X1* = DES encoding of *KKL* using key *K1*
4. Sets *K2* = *X1* exclusive-OR *KKL*
5. Sets *X2* = DES encoding of *KKR′* using key *K2*
6. Sets *VP* = *X2* exclusive-OR *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.

A double-length key is split in halves. The left half (high-order half) DEA 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.

The leftmost 32 bits of the result are compared to the trial input value or returned from the verb.

Ciphering Methods

The Data Encryption Standard (DES) algorithm defines operations on eight-byte data strings. Although the fundamental concepts of ciphering (enciphering and deciphering) and data verification are simple, different methods exist to process data strings that are not a multiple of eight bytes in length. The standards and IBM products that define these methods are as follows:

- ANSI X3.106 (CBC)
- ANSI X9.23.

Note: These methods also differ in how they define the initial chaining value (ICV).

This section describes how the verbs implement these methods.

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 eight-byte data strings. Figure D-1 and Figure D-2 on page D-4 show Cipher Block Chaining 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 eight-byte blocks. The ciphered result from processing an eight-byte block is exclusive-ORd with the next block of eight input bytes. The last eight-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 exclusive-ORd 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 or the key token is exclusive-ORd with the first eight bytes of data. If you specify the **CONTINUE** keyword, the OCV identified by the *chaining_vector* parameter is exclusive-ORd with the first eight bytes of data.

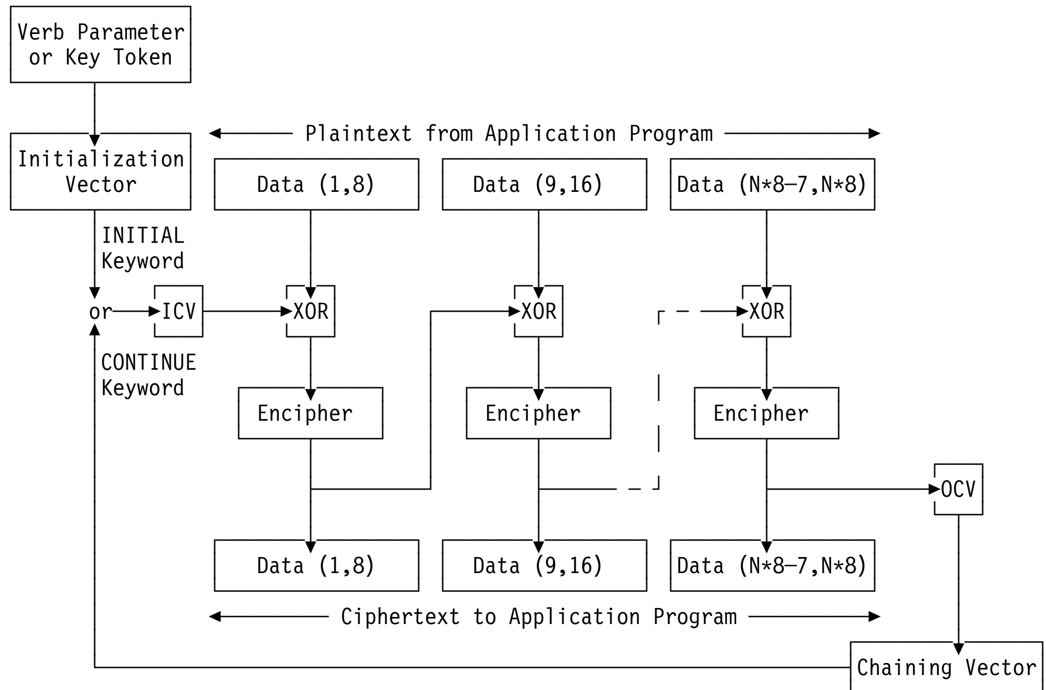

Figure D-1. Enciphering Using the CBC Method

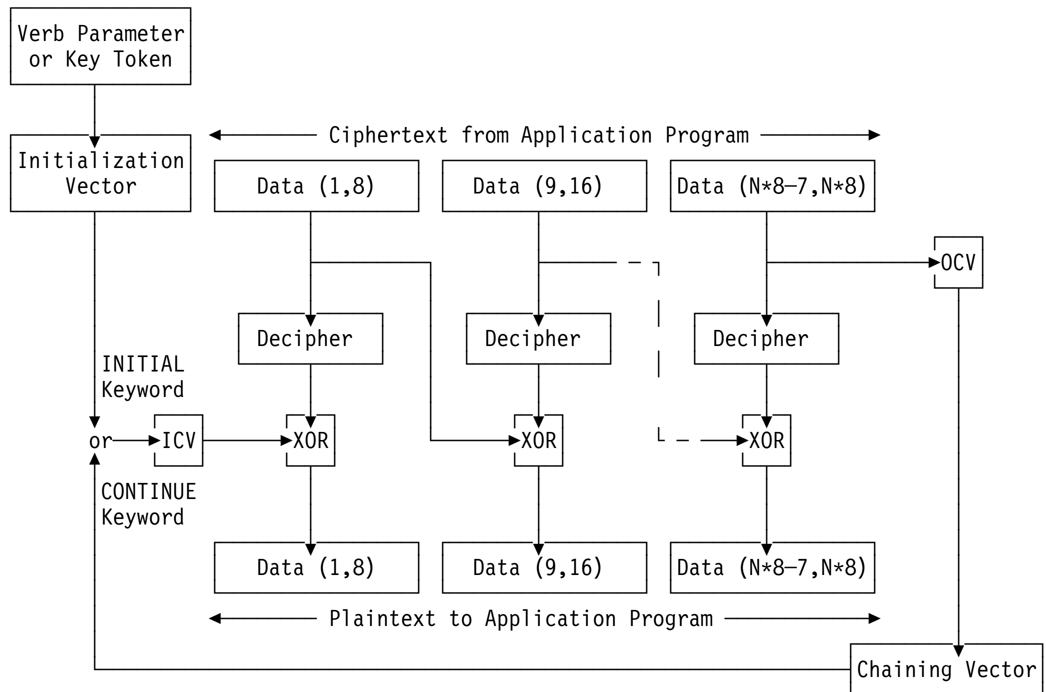

Figure D-2. Deciphering Using the CBC Method

ANSI X9.23

An enhancement to the basic Cipher Block Chaining mode of X3.106 is defined so that the system can process data lengths that are not exact multiples of eight bytes.

The ANSI X9.23 method *always* adds from one byte to eight bytes to the plaintext before encipherment. With these methods, the last added byte is the count of the added bytes and is within the range of X'01' to X'08'. The other added padding bytes are set to X'00'.

For other than the CBC method, when the security server deciphers the ciphertext, the security server uses the last byte of the deciphered data as the number of bytes to be removed (the pad bytes and the count byte). The resulting plaintext is the same length as the original plaintext.

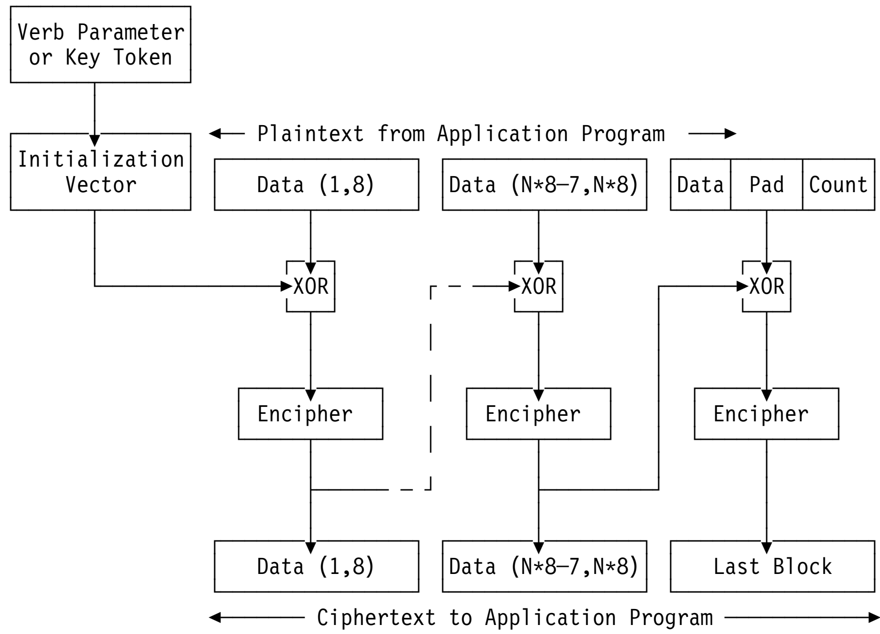

Figure D-3. Enciphering Using the ANSI X9.23 Method

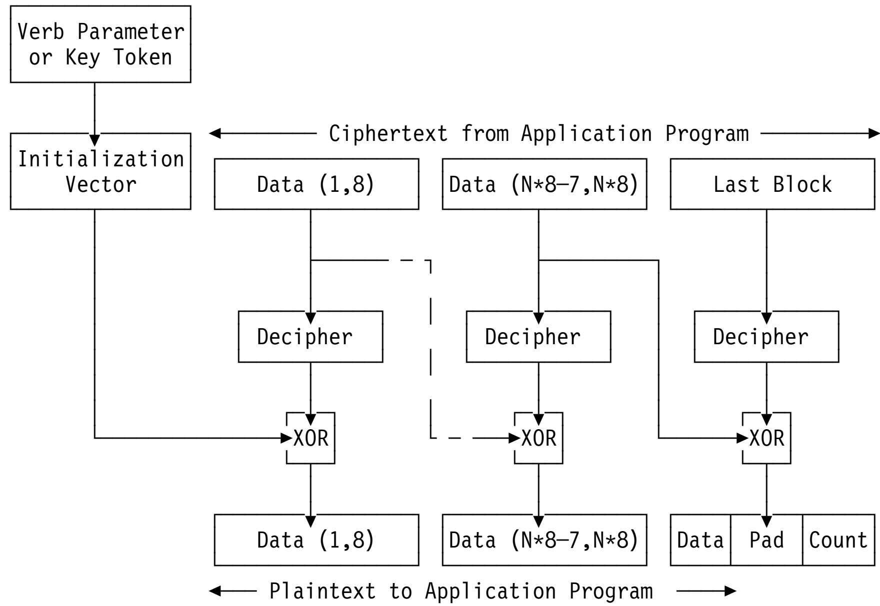

Figure D-4. Deciphering Using the ANSI X9.23 Method

MAC Calculation Method

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 D-5 shows the MAC calculation for binary data. KEY is a 64-bit key, and T_1 through T_n are 64-bit data blocks of text. In the standard, the Initial Chaining Value is binary zeros. If T_n is less than 64 bits long, binary zeros are appended (padded) to the right of T_n. The leftmost 32 bits of (O_n) are taken as the MAC.

The Financial Institution (Retail) Message Authentication Standard, ANSI X9.19 Optional Procedure 1, specifies additional processing of the 64-bit MAC value as calculated above. The "**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 returned according to other specifications supplied to the verb call.

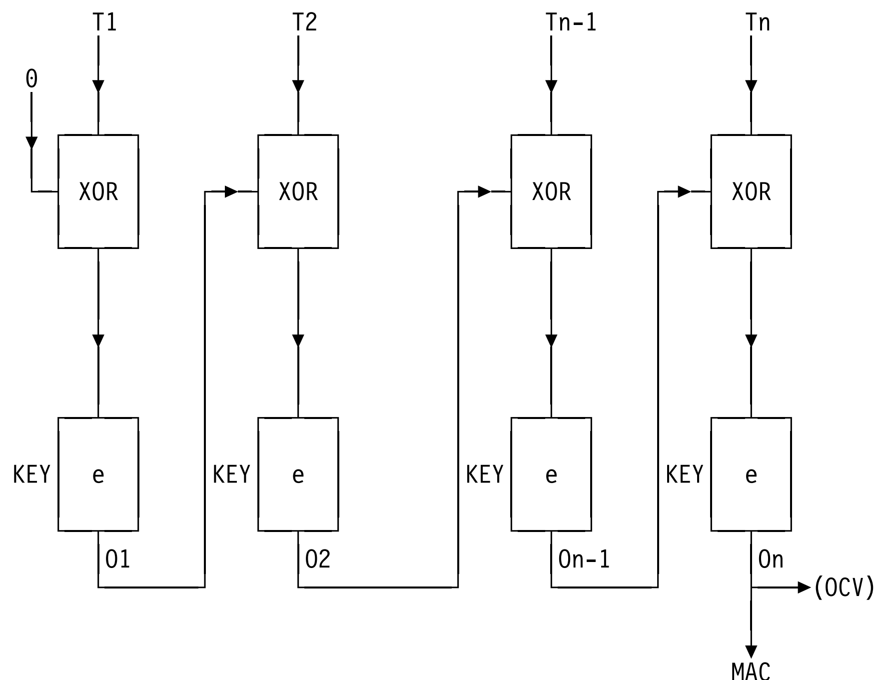

Figure D-5. MAC Calculation Method

Notes:

1. A footnote in the ANSI X9.9 standard suggests the future use of a 48-bit or 64-bit MAC. For these cases, the left-most 48 bits or the entire final output (O_n) is taken as the MAC.
2. The ANSI X9.9 standard defines five options. The MAC_Generate and MAC_Verify verbs implement option 1, binary data. The X9.9_Data_Editing verb is supplied as a subroutine to perform data fitting as required for options 2 and 4.

PKA92 Key Format and Encryption Process

The PKA_Symmetric_Key_Export, PKA_Symmetric_Key_Generate, and the PKA_Symmetric_Key_Import verbs optionally support a **PKA92** method of encrypting a DES or CDMF 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 only support 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 Figure D-6 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.

Figure D-6. PKA96 Clear DES Key Record

Offset (Bytes)	Length (Bytes)	Description
Zero-bit padding to form a structure as long as the length of the public key modulus. The implementation constrains the public key modulus to a multiple of 64 bits in the range of 512 to 1024 bits. Note that governmental export or import regulations can impose limits on the modulus length. The maximum length is validated by a check against a value in the Function Control Vector.		
000	005	Header and flags: X'01 0000 0000'
005	016	Environment Identifier (EID), encoded in ASCII
021	008	Control vector base for the DES key
029	008	Repeat of the CV data at offset 021
037	008	The single-length DES key or the left half of a double-length DES key
045	008	The right half of a double-length DES key or a random number. This value is locally designated “K.”
053	008	Random number, “IV”
061	001	Ending byte, X'00'

Masking Sub-process: Create a mask by CBC encrypting a multiple of 8 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. Exclusive-OR 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'.

Exclusive-OR 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 that the high-order two bits of the PKR record are valued to B'01'. and that the low-order four bits ot the PKR record are valued to B'0110'.

Unmasking Sub-process: Set IV to the value of the 8 bytes at offset 53 of the PKR record. Note that there is a variable quantity of padding prior to offset 0. See Figure D-6 on page D-8.

Set K to the exclusive-OR of IV and the value of the 8 bytes at offset 45 of the PKR record.

Create a mask that is equal in length to the PKR record by CBC encrypting a multiple of 8 bytes of binary zeros using K as the key and IV as the initialization vector. Exclusive-OR the mask with PKR and call the result the key record.

Extraction Sub-process: Confirm that:

- The five bytes at offset 0 in the key record are valued to X'01 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, that the 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', 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', the key is double length.

Encrypting a Key_Encrypting Key in the NL-EPP-5 Format

The PKA_Symmetric_Key_Generate verb supports a **NL-EPP-5** method of encrypting a DES key-encrypting key with an RSA public key. The verb returns an encrypted key block by RSA encrypting a key record formed in the following manner:

1. Format the key and other data per Figure D-7
2. Insert random padding data into the record
3. Insert the count of pad bytes plus one.

Figure D-7. NL-EPP-5 Key Record Format

Offset (Bytes)	Length (Bytes)	Description
000	02	Header and Null Cancelation bytes, X'0B00'
002	08	Single length key-encrypting key
002	16	Double length key-encrypting key
010 or 018		Random padding data
063	01	Padding count byte, X'36' for a single length key-encrypting key, or X'2E' for a double length key-encrypting key.

Triple-DES Ciphering Algorithms

For the IBM 4758-001, Triple-DES is used to encrypt keys. DES keys, when triple encrypted under a double length DES key, are ciphered using an e-d-e scheme without feedback. RSA private keys are also enciphered with triple DES techniques, and the techniques may also be used for additional purposes. Two techniques are employed depending on the length of the enciphering DES key, double or triple length: EDE2/DED2 and EDE3/DED3.

The EDE2 algorithm uses a 112-bit key to encrypt any number of 64-bit blocks of information. The DED2 algorithm is used to decrypt this information. The Key-Encrypting Key is a transport Key-Encrypting Key for an external key.

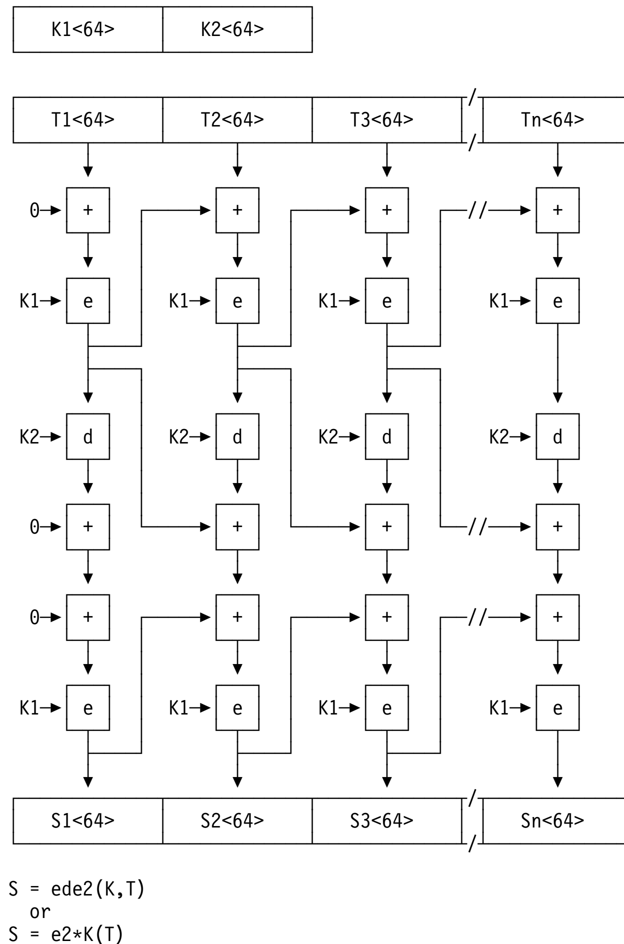

Figure D-8. EDE2 Algorithm

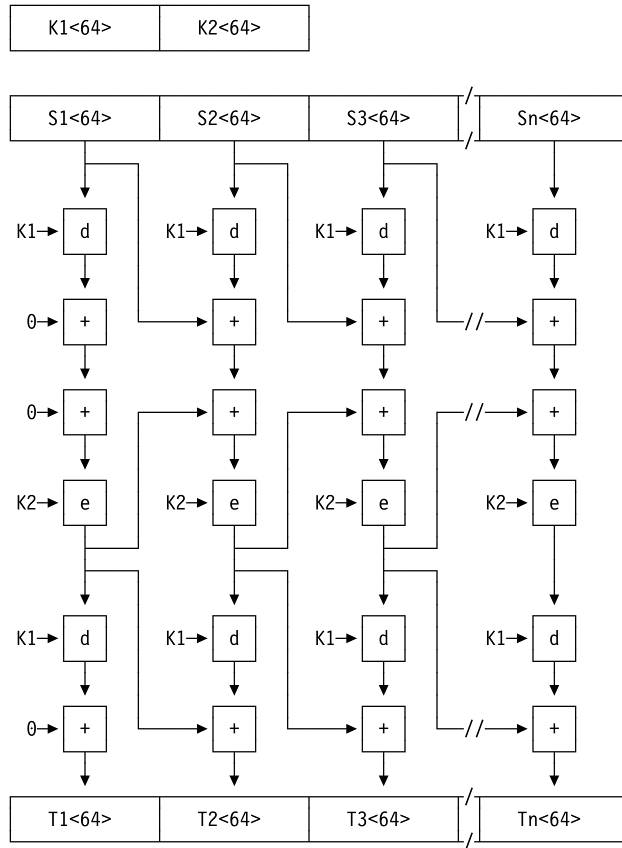

Figure D-9. DED2 Algorithm

The EDE3 algorithm uses a 168-bit key to encrypt any number of 64-bit blocks of information. The DED3 algorithm is used to decrypt this information. The Key-Encrypting Key is the master key for an internal key.

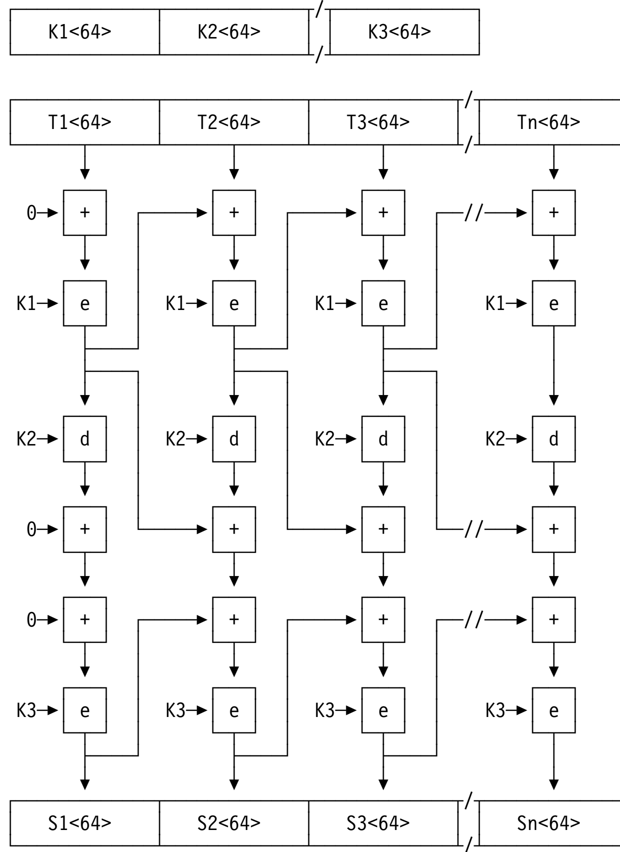

Figure D-10. EDE3 Algorithm

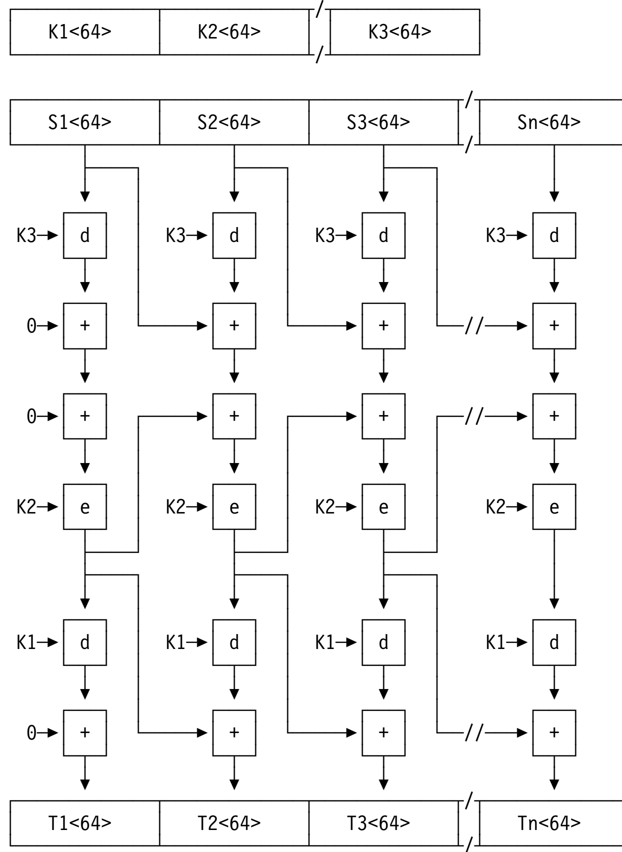

Figure D-11. DED3 Algorithm

RSA Key-Pair Generation

This section describes RSA Key Generation in the IBM 4755. The conditions on the RSA key elements for the 4755 are as follows.

1. p and q must be randomly chosen prime numbers
2. p and q must be somewhat different in length
3. (p-1) must contain a large prime factor, denoted p1, and likewise (q-1) must contain a large prime factor, q1
4. (p1-1) must contain a large prime factor, p2, and (q1-1) must contain a large prime factor, q2
5. (p+1) must not factor entirely small factors; ditto for (q+1)
6. the greatest common divisor of (p-1) and (q-1) must be small
7. the ratio p/q must not be close to the ratio of two small integers
8. let phi = (p-1) * (q-1); then phi and e must be relatively prime
9. let phi' = phi/GCD(p-1,q-1); then d must satisfy the equation e*d mod phi' = 1

The length requirements are:

1. len(n) = desired length of the modulus in bits
2. len(p) = 0.5 * len(n)
3. 16
4. len(q) = len(n)
5. len(p)
6. len(p1) = len(q1) = 0.8 * len(p)
7. len(p2) = len(q2) = 0.8 * len(p1)

Access Control Algorithms

The following sections describe algorithms and protocols used by the access control system.

Passphrase Verification Protocol

This section describes the process used to log a user on to the Cryptographic Coprocessor.

Design Criteria

The passphrase verification protocol is designed to meet the following criteria.

1. The use of cryptographic algorithms is permitted in the client logon software, but there must be no storage of any long-term cryptographic keys. This is because secure key storage is generally not available in the client workstation.
2. Replay attacks must not be feasible. This means that the logon request message must be protected so that it cannot be captured by an adversary, and later replayed to gain access to the genuine user's privileges.
3. An attacker should not be able to guess the cleartext content of the logon request message.
4. No special hardware should be required on the client workstation.
5. The logon process must result in the establishment of a session key known only to the Cryptographic Coprocessor and the client. This key will be used on subsequent transactions to prove the identity of the sender, and to secure transmitted data.
6. The session key will be generated in the coprocessor. Its hardware-based random number generator is of higher quality than software-based random number sources generally available.

Description of the Protocol

The protocol is comprised of the following steps.

1. The user provides his User ID (UID) and passphrase.
2. The passphrase is hashed in the client workstation, using SHA-1. The resulting hash is used to construct a logon key, denoted K_L.

   K_L is a triple-length DES key. The three components of the triple-length key are denoted $K1_L$, $K2_L$, and $K3_L$. $K1_L$ is comprised of the first eight bytes of the hash, $K2_L$ is comprised of the second eight bytes, and $K3_L$ is comprised of the last four bytes, concatenated with four bytes of X'00'. Figure D-12 shows an example to clarify this.

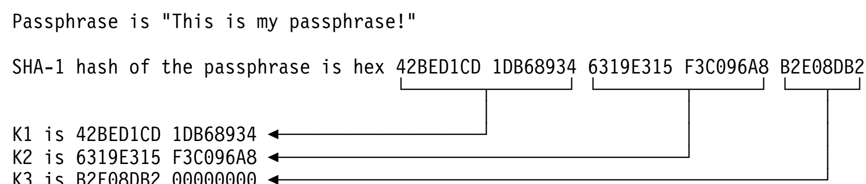

Figure D-12. Example of logon key computation

3. The client workstation generates a random number, RN (64 bits).

   Note: Note: The random number RN is not used inside the Cryptographic Coprocessor. It is only included in the protocol to guarantee that the cleartext of the logon request is different every time.

4. The client workstation sends a logon request to the Cryptographic Coprocessor, including the following information:

   { UID, eK_L(RN, UID, Timestamp) }

   Encryption uses DES EDE3[1] mode, performed in software in the client workstation. The timestamp includes both the time and the date, in GMT. It is used to prevent replay of the logon request. The timestamp is formed from the concatenation of binary encoded values of the year, month, day , hour, minute, and second. Each value is held in one byte except for the year which is held in a two-byte value.

5. The Cryptographic Coprocessor retrieves the user profile, which it has in secure internal memory. It uses the received userid value UID to locate the right profile. If the user's profile is not found, the logon request is rejected.

6. The coprocessor reads the hash of the user's passphrase from the profile, thus obtaining K_L.

7. The coprocessor uses K_L to decrypt the user's logon data, thus recovering the UID, Timestamp, and RN. It compares the recovered UID with the cleartext UID it received, and aborts if the two are not equal. Inequality is an indication that the passphrase was incorrect, or that someone tried to splice another user's captured logon data into their own request.

8. The coprocessor verifies that the recovered Timestamp is within 5 minutes of the current time, according to the Coprocessor's secure clock. If the Timestamp falls outside this window, it indicates a probable replay attack, and the logon request is rejected.

9. If everything in the preceding steps was acceptable, the user is logged on to the coprocessor. It generates a DES session key K_S, and returns this key to the client in the form $eK_L(K_S)$. The session key is a triple-length DES key.

10. In a secure internal table, the coprocessor stores the userid UID, the value of K_S, and the user's Role, which is extracted from the profile. This table is used on later requests to verify that the user is logged on, and to find the role defining the user's privileges. The table entry is destroyed when the user logs off.

11. The client workstation software (SAPI) saves K_S for use in subsequent transactions.

[1] For a description of the EDE3 encryption process, see Figure D-10 on page D-13.

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, Km, which is a triple-DES key and thus 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 Km among n trustees in a way that no set of t or less of trustees will have ANY information about Km, while t+1 trustees (or more) will be able to reconstruct Km.

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=Km.

   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:

1. After generating the set of authentic values (above sharing pahse) proceed as follows:
2. Take t+1 such values and interpolate the polynomial f(x) of degree t passing through these values using Lagrange interpolation. This will define a polynomial f(x) such that: f(i)=mk_i, and further more f(0) = MK. As we are only interested in Km, we present 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=SUM_j(b_{k_j} PROD_h (x_{k_h} / (x_{k_h}- x_{k_j}))) mod P
3. Proceed to install Km = a_0 = f(0) mod P.

Appendix E. Financial PIN Calculation Methods and PIN Blocks

This appendix describes the following:

- PIN calculation methods
- PIN block formats.

The PIN calculation methods are independent from PIN block formats. A PIN can be calculated by any method and used in any PIN format. For example, a PIN can be calculated by the IBM 3624 PIN calculation method and used either in the IBM 3624 PIN block format *or* in another PIN block format.

PIN Calculation Methods

The financial PIN verbs support the following PIN calculation methods:

- IBM 3624 PIN (IBM-PIN)
- IBM 3624 PIN Offset (IBM-PINO)
- Netherlands PIN-1 (NL-PIN-1).
- IBM German Bank Pool Institution PIN
- VISA PIN Validation Value (PVV)
- Interbank PIN

In the description of the financial PIN verbs, these terms are employed:

A-PIN The quantity derived from a function of the account number, PIN-generating key (PINGEN or PINVER), and other inputs such as a *decimalization table*.

C-PIN The quantity that a customer *should use* to identify himself; in general, this can be a customer-selected or institution-assigned quantity.

O-PIN A quantity, sometimes called an *offset*, that relates the A-PIN to the C-PIN as permitted by certain methods.

T-PIN The *trial* PIN presented for verification.

© Copyright IBM Corp. 1997-98

IBM 3624 PIN Calculation Method

The IBM 3624 PIN calculation method calculates a PIN that is from 4 to 16 digits in length.

The IBM 3624 PIN calculation method consists of the following steps to create the A-PIN:

1. Encrypt the hexadecimal validation data with a key that has a control vector that specifies the PINGEN (or PINVER) key type to produce a 64-bit quantity.
2. Convert the character format decimalization table to an equivalent array of sixteen 4-bit hexadecimal digits, and use the decimalization table to convert the hexadecimal digits (X'0' to X'F') of the encrypted validation data to decimal digits (X'0' to X'9'). Call this result newpin.

   Let newpin(i), decimalization_table(i), and encrypted_validation_data(i) each represent the (i)th hexadecimal digit in each quantity.

   The digits of newpin are obtained by the following procedure:

   ```
   For i = 1 to 16 do:
    j := encrypted_validation_data(i)
    newpin(i) := decimalization_table(j)
   end do
   ```
3. Select the *n* leftmost decimal digits of newpin, where *n* is the PIN length. The result is an *n*-digit calculated A-PIN. The PIN must be from 4 to 16 digits in length.

Example:

```
Encrypted validation data  = E5C1BD67B66AE7C6
Decimalization table index = 0123456789ABCDEF
Decimalization table       = 8351296477461538
Newpin                     = 3913656466643416
PIN length                 = 6
Calculated A-PIN           = 391365 (leftmost 6 digits of newpin)
```

IBM 3624 PIN Offset Calculation Method

The IBM 3624 PIN Offset calculation method is the same as the IBM 3624 PIN calculation method except that a step is added after the A-PIN is calculated to calculate or use an offset, O-PIN:

- To calculate an O-PIN, the additional step subtracts (digit-wise, modulo 10, with no carry) the calculated A-PIN from the customer-selected C-PIN.

  The result is an O-PIN (offset) of *n* decimal digits, where *n* is the PIN length and must be in the range from 4 to 16. The *PIN_check_length* parameter specifies *n* as the low-order (rightmost) digits of the *n*-digit PIN offset. The O-PIN (offset) is not encrypted.

- To use an offset to verify a trial PIN, the additional step adds (digit-wise, modulo 10, with no carry) the offset to the calculated A-PIN. The result is compared to the customer-entered trial PIN (T-PIN).

Notes:

1. The digit-wise subtraction is defined only for digits in the range from `X'0'` to `X'9'`. Any other value is not valid and causes processing to fail.
2. The length of the offset depends on the length of the PIN and must be less than or equal to the length of the PIN. The financial institution that issues the magnetic-stripe card determines the length of the PIN offset, which you specify with the *PIN_check_length* parameter.
3. When the length of the PIN offset is less than the length of the calculated PIN, the subtraction or addition begins with the low order PIN digit.

Netherlands PIN-1 Calculation Method

The Netherlands PIN-1 (NL-PIN-1) calculation method calculates a PIN that is 4 digits in length.

The method consists of the following steps to create the A-PIN:

1. Encrypt the hexadecimal validation data with a key that has a control vector that specifies the PINGEN (or PINVER) key type to produce a 64-bit quantity.
2. Convert the character format decimalization table to an equivalent array of sixteen 4-bit hexadecimal digits, and use the decimalization table to convert the third through sixth hexadecimal digits (X'0' to X'F') of the encrypted validation data to decimal digits (X'0' to X'9'). Call this result newpin.

   Note: The application must specify a decimalization table of 0, 1, ...9, 0, ...5.

   Let A-PIN(i), decimalization_table(i), and encrypted_validation_data(i) each represent the (i)th hexadecimal digit in each quantity.

   The digits of A-PIN are obtained by the following procedure:

   ```
   For i = 3 to 6 do:
    j := encrypted_validation_data(i)
    A-PIN(i-2) := decimalization_table(j)
   end do
   ```
3. The O-PIN offset, also a 4 digit quantity, when added digit-wise modulo 10 to the A-PIN results in the C-PIN, customer-used-PIN value.

Example:

```
Encrypted validation data  = 8325A637B66EA7A8
Decimalization table index = 0123456789ABCEDF
Decimalization table       = 0123456789012345
A-PIN                      = 2506
O-PIN                      = 9957
C-PIN, Customer PIN        = 1453
```

IBM German Bank Pool Institution PIN Calculation Method

The IBM German Bank Pool Institution PIN calculation method calculates an institution PIN that is 4 digits in length.

The German Bank Pool Institution PIN calculation method consists of the following steps:

1. Encrypt the hexadecimal validation data with an &instk. that has a control vector that specifies the PINGEN (or PINVER) key type to get a 64-bit quantity.
2. Convert the character format decimalization table to an equivalent array of sixteen 4-bit hexadecimal digits, and use the decimalization table to convert the first 6 hexadecimal digits (X'0' to X'F') of the encrypted validation data to decimal digits (X'0' to X'9'). Call this result newpin.

   The digits of newpin are obtained by the following procedure:

   ```
   For i = 1 to 6 do:
    j := encrypted_validation_data(i)
    newpin(i) := decimalization_table(j)
   end do
   ```
3. Select the 4 rightmost digits of newpin. The result is a 4-digit intermediate PIN.
4. If the first digit of the intermediate PIN is 0, assign 1 to the first digit of the institution PIN, and assign the remaining 3 digits of the intermediate PIN to the institution PIN.

   If the first digit of the intermediate PIN is not 0, assign the value of the intermediate PIN to the institution PIN.

   The PIN is not encrypted.

Example:

```
Encrypted validation data  = E5A4FD67B66AE7C6
Decimalization table index = 0123456789ABCDEF
Decimalization table       = 0123456789012345
Newpin                     = 450453
Intermediate PIN           = 0453 (4 rightmost digits of newpin)
Institution PIN            = 1453 (first digit is changed to 1
                             because the intermediate PIN had a
                             first digit of 0)
```

VISA PIN Validation Value (PVV) Calculation Method

The VISA PVV calculation method calculates a VISA PVV that is 4 digits in length.

The VISA PIN Validation Value (PVV) calculation method consists of the following steps:

1. Let X denote the transaction_security_parameter element. This parameter is the result of concatenating the 12-numeric-digit generating data with the 4-numeric-digit customer-entered PIN.
2. Encrypt X with the double-length key that has a control vector that specifies the PINGEN (or PINVER) key type 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.
4. Concatenate and use the resulting digits for the PVV. The PVV is not encrypted.

Interbank PIN Calculation Method

The Interbank PIN calculation method consists of the following steps:

1. Let X denote the transaction_security_parameter element converted to an array of sixteen 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. The PIN is not encrypted.

PIN Block Formats

The PIN verbs support one or more of the following PIN block formats:

- 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

3624 PIN Block Format

The 3624 PIN block format supports a PIN from 1 to 16 digits in length. A PIN that is longer than 16 digits is truncated on the right.

The following is the 3624 PIN block format:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X

Figure E-1. 3624 PIN Block Format

where:

P Is a PIN digit, which is a 4-bit value from X'0' to X'9'. The values of the PIN digits are independent.

P/X Is a PIN digit or a pad value. A PIN digit has a 4-bit value from X'0' to X'9'. A pad value has a 4-bit value from X'0' to X'F' and must be different from any PIN digit. The number of pad values for this format is in the range from 0 to 15, and all the pad values must have the same value.

Example:

```
PIN = 0123456, Pad = X'E'.
PIN block = X'0123456EEEEEEEEE'.
```

ISO-0 PIN Block Format

An ISO-0 PIN block format is equivalent to the ANSI X9.8, VISA-1, and ECI-1 PIN block formats. The ISO-0 PIN block format supports a PIN from 4 to 12 digits 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-0 PIN block:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	L	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F

Intermediate PIN Block = IPB

0	0	0	0	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN

PAN Block

0	L	P	P	P XOR PAN	P XOR PAN	P/F XOR PAN	P/F XOR PAN	P/F XOR PAN	P/F XOR PAN	P/F XOR PAN	P/F XOR PAN	P/F XOR PAN	P/F XOR PAN	F XOR PAN	F XOR PAN

PIN Block = IPB XOR PAN Block

Figure E-2. ISO-0 PIN Block Format

where:

0 Is the value X'0'.

L Is the length of the PIN, which is a 4-bit value from X'4' to X'C'.

P Is a PIN digit, which is a 4-bit value from X'0' to X'9'. The values of the PIN digits are independent.

P/F Is a PIN digit or pad value. A PIN digit has a 4-bit value from X'0' to X'9'. A pad value has a 4-bit value of X'F'. The number of pad values in the intermediate PIN block (IPB) is from 2 to 10.

F Is the value 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-0, ANSI X9.8, VISA-1, or ECI-1

Each PAN digit has a value from X'0' to X'9'.

The PIN block is the result of exclusive-ORing the 64-bit IPB with the 64-bit PAN block.

Example:

```
L= 6, PIN = 123456, Personal Account Number = 111222333444555
06123456FFFFFFFF : IPB
0000222333444555 : PAN block for ISO-0 (ANSI X9.8, VISA-1, ECI-1) format
06121675CCBBBAAA : PIN block for ISO-0 (ANSI X9.8, VISA-1, ECI-1) format.
```

ISO-1 PIN Block Format

The ISO-1 PIN block format is equivalent to an ECI-4 PIN block format. The ISO-1 PIN block format supports a PIN from 4 to 12 digits in length. A PIN that is longer than 12 digits is truncated on the right.

The following is the ISO-1 PIN block format:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	L	P	P	P	P	P/R	P/R	P/R	P/R	P/R	P/R	P/R	P/R	R	R

Figure E-3. ISO-1 PIN Block Format

where:

1 Is the value X'1'.

L Is the length of the PIN, which is a 4-bit value from X'4' to X'C'.

P Is the PIN digit, which is a 4-bit value from X'0' to X'9'. The values of the PIN digits are independent.

R Is a random digit, which is a value from X'0' to X'F'. Typically, this should be used for predetermined transaction unique data such as a sequence number.

P/R Is a PIN digit or a random digit, depending on the value of PIN length L. The number of random digits is in the range from 2 to 10, and the random digits can be different.

Example:

```
L=6, PIN = 123456, L = X'6'.
PIN block = X'161234566ABCFDE1', where X'6', X'A', X'B', X'C', X'F',
X'D', X'E', and X'1' are the random fillers.
```

ISO-2 PIN Block Format

The ISO-2 PIN block format supports a PIN from 4 to 12 digits in length. A PIN that is longer than 12 digits is truncated on the right.

The following is the ISO-2 PIN block format:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	L	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F

Figure E-4. ISO-2 PIN Block Format

where:

1 Is the value X'1'.

L Is the length of the PIN, which is a 4-bit value from X'4' to X'C'.

P Is the PIN digit, which is a 4-bit value from X'0' to X'9'. The values of the PIN digits are independent.

F Is a fill digit valued to X'F'.

P/F Is a PIN digit or a fill digit.

Example:

```
L=6, PIN = 123456, L = X'6'.
PIN block = X'26123456FFFFFFFF'.
```

Appendix F. Verb List

This appendix lists the verbs supported by the CCA Support Program feature for the IBM 4758 PCI Cryptographic Coprocessor.

Figure F-1 lists each verb by the verb's pseudonym and entry-point name and shows the operating environment under which the verb is supported. A check (√) in the operating environment column means that the verb is available for use in that operating environment[1].

Figure F-1 (Page 1 of 2). Security API Verbs in Supported Environments

Pseudonym	Entry-Point Name	OS/2	AIX	NT	Page/ Book
DES Key Processing and Key Storage Verbs					
Clear_Key_Import	CSNBCKI	√	√	√	5-16
Data_Key_Export	CSNBDKX	√	√	√	5-17
Data_Key_Import	CSNBDKM	√	√	√	5-18
Diversified_Key_Generate	CSNBDKG	√	√	√	5-20
Key_Export	CSNBKEX	√	√	√	5-23
Key_Generate	CSNBKGN	√	√	√	5-25
Key_Import	CSNBKIM	√	√	√	5-31
Key_Part_Import	CSNBKPI	√	√	√	5-33
DES_Key_Record_Create	CSNBKRC	√	√	√	7-4
DES_Key_Record_Delete	CSNBKRD	√	√	√	7-5
DES_Key_Record_List	CSNBKRL	√	√	√	7-7
DES_Key_Record_Read	CSNBKRR	√	√	√	7-9
Key_Record_Write	CSNBKRW	√	√	√	7-10
Key_Test	CSNBKYT	√	√	√	5-35
Key_Token_Build	CSNBKTB	√	√	√	5-38
Key_Token_Change	CSNBKTC	√	√	√	5-41
Key_Translate	CSNBKTR	√	√	√	5-43
Random_Number_Generate	CSNBRNG	√	√	√	5-45
PKA_Symmetric_Key_Export	CSNDSYX	√	√	√	5-47
PKA_Symmetric_Key_Generate	CSNDSYG	√	√	√	5-49
PKA_Symmetric_Key_Import	CSNDSYI	√	√	√	5-52

[1] Figure F-1 lists the verbs that are used with DES and PKA96 processing; for information about the verbs that are used with PKA92 public-key processing, see the *TSS Programming Reference: Volume II, Public-Key Cryptography* SC31-2888.

© Copyright IBM Corp. 1997-98

Figure F-1 (Page 2 of 2). Security API Verbs in Supported Environments					
Pseudonym	**Entry-Point Name**	**OS/2**	**AIX**	**NT**	**Page/ Book**
Data Confidentiality and Data Integrity Verbs					
Decipher	CSNBDEC	√	√	√	6-4
Digital_Signature_Generate	CSNDDSG	√	√	√	4-4
Digital_Signature_Verify	CSNDDSV	√	√	√	4-7
Encipher	CSNBENC	√	√	√	6-7
MAC_Generate	CSNBMGN	√	√	√	6-10
MAC_Verify	CSNBMVR	√	√	√	6-13
One_Way_Hash	CSNBQWH	√	√	√	4-10
Hardware Access-Control Verbs					
Access_Control_Initialization	CSUAACI	√	√	√	2-13
Access_Control_Maintenance	CSUAACM	√	√	√	2-16
Cryptographic_Facility_Control	CSUSCFC	√	√	√	2-22
Cryptographic_Facility_Query	CSUSCFQ	√	√	√	2-26
Key_Storage_Initialization	CSNBKSI	√	√	√	2-36
Logon_Control	CSUSLCT	√	√	√	2-38
Master_Key_Distribution	CSUAMKD	√	√	√	2-42
Master_Key_Process	CSNBMKP	√	√	√	2-46
RSA Key Administration and Key Storage Verbs					
PKA_Key_Generate	CSNDPKG	√	√	√	3-6
PKA_Key_Import	CSNDPKI	√	√	√	3-10
PKA_Key_Token_Build	CSNDPKB	√	√	√	3-12
PKA_Key_Token_Change	CSNDKTC	√	√	√	3-18
PKA_Key_Record_Create	CSNDKRC	√	√	√	7-11
PKA_Key_Record_Delete	CSNDKRD	√	√	√	7-13
PKA_Key_Record_List	CSNDKRL	√	√	√	7-15
PKA_Key_Record_Read	CSNDKRR	√	√	√	7-17
PKA_Key_Record_Write	CSNDKRW	√	√	√	7-19
PKA_Public_Key_Extract	CSNDPKX	√	√	√	3-20
PKA_Public_Key_Hash_Register	CSNDPKH	√	√	√	3-22
PKA_Public_Key_Register	CSNDPKR	√	√	√	3-24
Retained_Key_Delete	CSNDRKD	√	√	√	7-21
Retained_Key_List	CSNDRKL	√	√	√	7-22
Financial Services Support Verbs					
Clear_PIN_Encrypt	CSNBCPE	√	√	√	8-12
Clear_PIN_Generate	CSNBPGN	√	√	√	8-15
Clear_PIN_Generate_Alternate	CSNBCPA	√	√	√	8-18
Encrypted_PIN_Generate	CSNBEPG	√	√	√	8-24
Encrypted_PIN_Translate	CSNBPTR	√	√	√	8-29
Encrypted_PIN_Verify	CSNBPVR	√	√	√	8-34
SET_Block_Compose	CSNDSBC	√	√	√	8-40
SET_Block_Decompose	CSNDSBD	√	√	√	8-44

Appendix G. Access Control Request Function Codes

The following table lists all of the access control points for the functions in the Cryptographic Coprocessor. Each access control point corresponds to one primitive function, which can be enabled or disabled in a role.

Each code is two bytes in length, for a maximum of 65,536 possible codes. Any codes not listed in the table are reserved for future use.

Figure G-1 (Page 1 of 3). Access control point codes

Code	**Function Name**
X'000E'	Encipher
X'000F'	Decipher
X'0010'	Generate MAC
X'0011'	Verify MAC
X'0012'	Re-encipher To Master Key
X'0013'	Re-encipher From Master Key
X'0018'	Load First Master Key Part
X'0019'	Combine Master Key Parts
X'001A'	Set Master Key
X'001B'	Load First Key Part
X'001C'	Combine Key Parts
X'001D'	Compute Verification Pattern
X'001F'	Translate Key
X'0020'	Generate Random Master Key
X'0032'	Clear New Master Key Register
X'0033'	Clear Old Master Key Register
X'0040'	Generate Diversified Key
X'008C'	Generate Key Set
X'008E'	Generate Key
X'0090'	Re-encipher To Current Master Key
X'00A0'	Generate Clear 3624 PIN
X'00A4'	Generate Clear 3624 PIN Offset
X'00AB'	Verify Encrypted 3624 PIN
X'00AC'	Verify Encrypted GBP PIN
X'00AD'	Verify Encrypted VISA PVV
X'00AE'	Verify Encrypted InterBank PIN
X'00AF'	Format and Encrypt PIN
X'00B1'	Generate Formatted and Encrypted GBP PIN
X'00B2'	Generate Formatted and Encrypted InterBank PIN
X'00B3'	Translate PIN with No Format-Control to No Format-Control
X'00B7'	Reformat PIN with No Format-Control to No Format-Control

© Copyright IBM Corp. 1997-98

Figure G-1 (Page 2 of 3). Access control point codes

Code	Function Name
X'00BB'	Generate Clear VISA PVV Alternate
X'00C3'	Encipher Under Master Key
X'00D7'	Generate Key Set Extended
X'00DB'	Replicate Key
X'0100'	Digital Signature Generate
X'0101'	Digital Signature Verify
X'0102'	Key Token Change
X'0103'	PKA Key Generate
X'0104'	PKA Key Import
X'0105'	Symmetric Key Export
X'0106'	Symmetric Key Import
X'0109'	Data Key Import
X'010A'	Data Key Export
X'010B'	Compose SET Block
X'010C'	Decompose SET Block
X'010D'	PKA92 Symmetric Key Generate
X'010E'	NL-EPP-5 Symmetric Key Generate
X'010F'	Reset Intrusion Latch
X'0110'	Set Clock
X'0111'	Reinitialize Device
X'0112'	Initialize access control system roles and profiles
X'0113'	Change the expiration date in a user profile
X'0114'	Change the authentication data (e.g. passphrase) in a user profile
X'0115'	Reset the logon failure count in a user profile
X'0116'	Load Roles and Profiles
X'0117'	Delete a User Profile
X'0118'	Delete a Role
X'0119'	Load Function Control Vector
X'011A'	Clear Function Control Vector
X'011B'	Force User Logoff
X'011C'	Set EID (Environment Identifier)
X'011D'	Initialize Km Cloning Control
X'0200'	Register PKA Public Key Hash
X'0201'	Register PKA Public Key, with Cloning
X'0202'	Register PKA Public Key
X'0204'	PKA Clone Key Generate
X'0211 to 21F'	Clone-information Obtain, 1-15

Figure G-1 (Page 3 of 3). Access control point codes	
Code	**Function Name**
X'0221 to 21F'	Clone-information Install, 1-15
X'0203'	Delete Retained Key
X'0230'	List Retained Key
X'0231'	Generate Clear NL-PIN-1 Offset
X'0232'	Verify Encrypted NL-PIN-1
X'0235'	PKA92 Symmetric Key Import
X'0236'	PKA92 PIN Key Import

List of Abbreviations

ac alternating current

ANSI American National Standards Institute

ACF/VTAM Advanced Communications Function for the Virtual Telecommunications Access Method

AIX Advanced Interactive Executive operating system

APF Authorized Program Facility

API Application Programming Interface

ASCII American National Standard Code for Information Interchange

AS/400 Application System/400

BCD Binary Coded Decimal

BTU British Thermal Unit

C Celsius

CBC Cipher-Block Chaining

CCA Common Cryptographic Architecture

CDMF Commercial Data Masking Facility

cfm cubic feet per minute

CICS Customer Information Control System

CKDS Cryptographic Key Data Set

cm centimeter

COBOL Common Business-Oriented Language

CTC Channel To Channel

CPRB Connectivity Programming Request Block

CUSP Cryptographic Unit Support Program

CV Control Vector.

CVC Card-Verification Code.

CVV Card-Verification Value

DCI Data Channel Interlock

DEA Data Encryption Algorithm

DES Data Encryption Standard

DMA Direct Memory Access

DOS Disk Operating System

EBCDIC Extended Binary Coded Decimal Interchange Code

EC Engineering Change

ECB Electronic Code Book

EEPROM Electrically Erasable, Programmable Read-Only Memory

EIA Electronics Industries Association

EMS Expanded Memory Specification.

EPO Emergency Power Off

ESCON Enterprise Systems Connection

ESS Establish Secure Session

F Fahrenheit

FBSS Financial Branch System Services

FCC Federal Communications Commission

FEPROM Flash Erasable, Programmable Read-Only Memory

FIPS Federal Information Processing Standard

ft foot

GTF Generalized Trace Facility

HCD Hardware Configuration Definition

Hz Hertz

IBM International Business Machines

ICRF Integrated Cryptographic Facility

ICSF Integrated Cryptographic Service Facility

ICSF/MVS Integrated Cryptographic Service Facility/Multiple Virtual Storage

IMS Information Management System

in. inch

I/O Input/Output

IOCP Input/Output Control Program

IPL Initial Program Load

ISO International Standards Organization

KB Kilobyte

KEK Key-Encrypting Key

KM Master key

kPa kilopascal

KSS Key Storage Synchronization

kVA kilovolt ampere

LAN Local Area Network

LANDP LAN Distributed Platform

LED Light-Emitting Diode

© Copyright IBM Corp. 1997-98

LU Logical Unit

MAU Multistation Access Unit

MB Megabyte

MCS Multiple Console Support

m meter

MAC Message Authentication Code

MBps Megabytes per second

MD5 Message Digest 5 Hashing Algorithm

MDC Modification Detection Code

MKVN Master Key Version Number

MVS Multiple Virtual Storage

MVS/DFP MVS/Data Facility Product

MVS/ESA MVS/Enterprise Systems Architecture

MVS/SP MVStorage/System Product

MVS/XA MVS/Extended Architecture

NEMA National Electrical Manufacturers Association

NIST National Institute of Science and Technology (USA).

OEM Original Equipment Manufacturer

OLTS Online Test System

OS/VS Operating System/Virtual Storage

OS/2 Operating System/2

OS/400 Operating System/400

Pa Pascal

PC Personal Computer

PC DOS Personal Computer Disk Operating System

PCF Programmed Cryptographic Facility

pH A measure of acidity or alkalinity

PIN Personal Identification Number

PKA Public Key Algorithm

POS Point Of Sale

POST Power-On Self Test

PROM Programmable Read-Only Memory. (A)

PRPQ Program Request for Price Quotation

PS/2 Personal System/2

RACF Resource Access Control Facility

RAM Random Access Memory

RISC Reduced Instruction-Set Computer

ROM Read-Only Memory

RPQ Request for Price Quotation

RSA Rivest, Shamir, and Adleman

RU Request Unit

SAA Systems Application Architecture

SAF System Authorization Facility

SHA Secure Hashing Algorithm

SM Service Memorandum

SNA Systems Network Architecture

SRIU Service request/reply interchange unit

SRPI Server-Requester Programming Interface

TSO Time Sharing Option

TSR Terminate and Stay Resident

TSS Transaction Security System

UCW Unit Control Word

UKPT Unique-Key-Per-Transaction

UL/CSA Underwriters Laboratory/Canadian Standards Association

V Volt

VGA Video Graphics Adapter

WCS Workstation Cryptographic Services

VM Virtual Machine

WSSP Workstation Security Services Program

Glossary

This glossary includes some terms and definitions from the *IBM Dictionary of Computing*, New York: McGraw Hill, 1994. This glossary also includes some terms and definitions from:

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

access. A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

access control. Ensuring that the resources of a computer system can be accessed only by authorized users in authorized ways.

access method. (1) A technique for moving data between main storage and input/output devices. (2) In the Transaction Security System products, the part of the IBM Network Security Processor MVS Support Program that supports the Application Program Interfaces, the cross-memory server, the request manager, and that sends cryptographic requests to the appropriate Network Security Processor.

adapter. A printed circuit card that modifies the system unit to allow it to operate in a particular way.

address. (1) In data communication, the unique code assigned to each device or workstation connected to a network. (2) A character or group of characters that identifies a register, a particular part of storage, or some other data source or data destination. (A) (3) To refer to a device or an item of data by its address. (A) (I)

Advanced Communications Function for the Virtual Telecommunications Access Method. ACF/VTAM is an IBM-licensed program that controls communication and the flow of data in an SNA network.

Advanced Interactive Executive (AIX) operating system. IBM's implementation of the UNIX** operating system.

alternating current (ac). An electric current that reverses its direction at regularly recurring intervals.

American National Standard Code for Information Interchange (ASCII). The standard code (8 bits including parity a bit), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

American National Standards Institute (ANSI). An organization, consisting of producers, consumers, and general interest groups that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. (A)

Application System/400 system (AS/400). AS/400 is one of a family of general purpose midrange systems with a single operating system, Operating System/400, that provides application portability across all models.

assembler language. A source language that includes symbolic machine language statements in which there is a one-to-one correspondence between the instruction formats and the data formats of the computer.

authentication. (1) A process used to verify the integrity of transmitted data, especially a message. (T) (2) In computer security, a process used to verify the user of an information system or protected resources.

authorization. (1) The right granted to a user to communicate with or make use of a computer system. (T) (2) The process of granting a user either complete or restricted access to an object, resource, or function.

authorize. To permit or give authority to a user to communicate with or make use of an object, resource, or function.

** UNIX is a trademark of UNIX Systems Laboratories, Incorporated.

© Copyright IBM Corp. 1997-98

Authorized Program Facility (APF). APF is a facility that permits identification of programs authorized to use restricted functions.

B

batch file. A file that contains multiple DOS commands that are processed sequentially whenever you type the name of the batch file and press the Enter key.

batch initialization utility. In the Transaction Security System, one of the utility programs supplied with the Workstation Security Services Program. It enables you to initialize the hardware access controls and the cryptographic key registers in the Cryptographic Adapter, the Security Interface Unit, and the Personal Security Card.

Binary-Coded Decimal (BCD). BCD notation is asystem of binary coding where which each decimal digit is represented by a binary numeral; for example, in BCD notation, the number “twenty-three” is represented by the binary digits 0010 0011 (compare its representation 10111 in the pure binary numeration system).

bus. In a processor, a physical facility along which data is transferred.

bus in. A unidirectional data bus that is part of the System/360 or System/370 Parallel Channel Interface. This bus passes data from the control unit to the host.

bus out. A unidirectional data bus that is part of the System/360 or System/370 Parallel Channel Interface. This bus passes data from the host to the control unit.

byte. (1) A binary character operated on as a unit and usually shorter than a computer word. (A) (2) A string that consists of a number of bits, treated as a unit, and representing a character. (3) A group of eight adjacent binary digits that represents one EBCDIC character.

C

Card-Verification Code (CVC). See *Card-Verification Value*.

Card-Verification Value (CVV). CVV is a cryptographic method, defined by VISA, for detecting forged magnetic-striped cards. This method cryptographically checks the contents of a magnetic stripe. This process is functionally the same as MASTERCARD’s Card-Verification Code (CVC) process.

Commercial Data Masking Facility (CDMF). CMDF is an alternate algorithm for data confidentiality applications, based on the DES algorithm with an effective 40 bit key strength.

channel. A path along which signals can be sent; for example, a data channel or an output channel. (A)

channel adapter. A communication controller hardware unit used to attach the controller to a System/370 data channel.

channel-attached. (1) Pertaining to attachment of devices directly by data channels (I/O channels) to a computer. (2) Pertaining to devices attached to a controlling unit by cables rather than by telecommunication lines.

channel-interface assembly. An assembly that attaches to the Network Security Processor with a flat-ribbon cable so that a channel attachment can be made. The channel-interface assembly includes bus and tag sockets.

channel speed. The rate at which data is transferred between a host computer and a channel-attached device. Channel speed is dependent on the type of sub-channel defined by the channel-attached device.

ciphertext. Text that results from the encipherment of plaintext. See also *plaintext*.

Cipher Block Chaining (CBC). CBC is a mode of operation that cryptographically connects one block of ciphertext to the next plaintext block.

clear data. (1) Data that is not enciphered.

cleartext. Text that has not been altered by a cryptographic process. Synonym for plaintext. See also *ciphertext*.

Common Cryptographic Architecture (CAA) API. The CCA API is the programming interface described in the *Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*.

Common Cryptographic Architecture Services/400. This IBM PRPQ runs in an AS/400 system under the OS/400 operating system to support a xryptographic co-processor. PRPQ 5700 XBI also enables the use of a Security Interface Unit and Personal Security Card on an AS/400 system.

concatenation. An operation that joins two characters or strings in the order specified, forming one string whose length is equal to the sum of the lengths of its parts.

configuration. (1) The manner in which the hardware and software of an information processing system are organized and interconnected. (T) (2) The physical and

logical arrangement of devices and programs that consitiutes a data processing system.

configuration vector. In the Transaction Security System, a public-key hardware data structure that specifies the security levels under which the user requires the system to operate, and the key-management protocol that determines the use of each type of public key. The configuration vector is stored in the security module on the cryptographic adapter.

Connectivity Programming Request Block (CPRB). The CPRB is an interface control block used by requesters and servers to communicate information over the Server-Requester Programming INterface (SRPI).

controller. A device that coordinates and controls the operation of one or more input/output devices, such as workstations, and synchronizes the operation of such devices with the operation of the system as a whole.

control program. (1) A computer program designed to schedule and to supervise the programs running in a computer system. (A) (I) (2) In the Transaction Security System, the IBM 4753 Network Security Processor Control Program.

control vector (CV). In the Transaction Security System, a 16-byte string that is exclusive-ORd with a master key or a Key-Encrypting Key to create another key that is used to encipher and decipher data or data keys. A control vector determines the type of key and the restrictions on the use of that key.

cross-memory server. The part of the access method that receives the request from the security API and exits to the System Authorization Facility interface.

cryptographic adapter. The 4755 is an expansion board that provides a comprehensive set of cryptographic functions for the Network Security Processor and the workstation.

Cryptographic Key Data Set (CKDS). CKDS is a data set containing the encrypting keys used by an installation.

Cryptographic Key Data Set Conversion Utility. The CKDS Conversion utility is that part of the IBM Network Security Processor MVS Support Program that converts PCF/CUSP cryptographic key data sets to Network Security Processor key data sets.

cryptographic processor. An AS/400 I/O processor that uses the Cryptographic Adapter and &CCAS4. to provide a comprehensive set of DES and RSA-based cryptographic services for an AS/400 system.

cryptographic services. In the Transaction Security System, the part of the security server that processes requests from an application program or the HIKM utility and sends the requests to the cryptographic hardware for processing.

Cryptographic Unit Support Program (CUSP). CUSP is an IBM licensed program (program number 5740-XY6) that supports the creation and management of cryptographic keys. This program interacts with the IBM 3848 Cryptographic Unit to encipher and decipher data.

cryptography. The transformation of data to conceal its meaning.

CUSP/PCF. An interface between the ACF/VTAM program and the Network Security Processor MVS Support Program.

CUSP/PCF transform. That part of the access method that contains the code to transform the CUSP/PCF cryptographic requests to a format that the security API stub can use.

Customer Information Control System (CICS). CICS is an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases.

D

data. (1) A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. Data includes constants, variables, arrays, and character strings. (2) Any representations such as characters or analog quantities to which meaning is or might be assigned. (A)

Data Channel Interlock (DCI). DCI is a protocol for transmitting data on a channel. In this protocol, the sender raises and maintains a signal on the channel until the receiver acknowledges receipt of the signal.

data-encrypting key. (1) A key used to encipher, decipher, or authenticate data. (2) Contrast with *Key-Encrypting Key*.

Data Encryption Algorithm (DEA). DEA is a 64-bit block cipher that uses a 64-bit key, of which 56 bits are used to control the cryptographic process and 8 bits are

used for parity checking to ensure that the key is transmitted properly.

Data Encryption Standard (DES). DES is the National Institute of Standards and Technology Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46. which allows only hardware implementations of the data-encryption algorithm.

data set. The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

data streaming. An uninterrupted transfer of information over an interface in order to achieve high data transfer rates. (A)

decipher. (1) To convert enciphered data into clear data. (2) Synonym for *decrypt*. (3) Contrast with *encipher*.

decode. (1) To convert data by reversing the effect of some previous encoding. (A) (I) (2) In the Transaction Security System products, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard (DES). (3) Contrast with *encode*.

decrypt. (1) To decipher or decode. (2) Synonym for *decipher*. (3) Contrast with *encrypt*.

device ID. In the Transaction Security System products, a user-defined field in the global configuration-data that can be used for any purpose the user specifies. For example, it can be used to identify a particular device, by using a unique ID similar to a serial number.

diagnostic. Pertaining to the detection and isolation of errors in programs, and faults in equipment.

directory server. A server that manages key records in key storage by using an Indexed Sequential Access Method.

Disk Operating System (DOS). DOS is an operating system for computers that use disks and diskettes for the auxiliary storage of programs and data.

driver. A program that contains the code needed to attach and use a device.

dump file. In the IBM 4753, a file that contains a record of dump information for the selected servers.

E

Electronic Code Book (ECB). ECB is a mode of operation used with block cipher cryptographic algorithms in which plaintext or ciphertext is placed in the input to the algorithm and the result is contained in the output of the algorithm.

Electronics Industries Association (EIA). EIA is an organization of electronics manufacturers that advances the technological growth of the industry, represents the views of its members, and develops industry standards.

encipher. (1) To scramble data or to convert data to a secret code that masks the meaning of the data to unauthorized recipients. (2) Synonym for *encrypt*. (3) Contrast with *decipher*. (4) See also *encode*.

enciphered data. Data whose meaning is concealed from unauthorized users or observers. See also *ciphertext*.

encode. (1) To convert data by the use of a code in such a manner that reconversion to the original form is possible. (T) (2) In the Transaction Security System products, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard. (3) Contrast with *decode*. (4) See also *encipher*.

encrypt. (1) Synonym for *encipher*. (T) (2) To convert clear text into ciphertext. (3) Contrast with *decrypt*.

engineering change (EC) level. A number that indicates the hardware version.

Erasable Programmable Read-Only Memory (EPROM). EPROM is a PROM that can be erased by a special process and reused. (T)

ESCON. The data processing environment having an Enterprise Systems Connection channel-to-control-unit I/O interface that uses optical cables as the transmission medium.

Establish Secure Session (ESS). ESS describes the way by which the hardware components establish authenticity with each other.

exit routine. In the Transaction Security System products, a user-provided routine that acts as an extension of the cross-memory server in the IBM Network Security Processor MVS Support Program.

Expanded Memory Specification (EMS). EMS is a software interface for accessing additional memory in personal computers that use the disk operating system (DOS).

expansion board. In an IBM personal computer, a panel the user can install in an expansion slot to add memory or special features.

EXPORTER key. (1) In the Transaction Security System, a type of DES Key-Encrypting Key that can encipher a key at a sending node. (2) Contrast with *IMPORTER key*.

F

facility. (1) An operational capability, or the means for providing such a capability. (T) (2) A service provided by an operating system for a particular purpose; for example, the checkpoint/restart facility.

feature. A part of an IBM product that can be ordered separately.

Federal Communications Commission (FCC). The FCC is a board of commissioners, appointed by the President under the Communications Act of 1934, and having the power to regulate all interstate and foreign communications by wire and radio originating in the United States.

Federal Information Processing Standard (FIPS). FIPS is a standard published by the US National Institute of Science and Technology.

Financial Branch System Services (FBSS). FBSS is an IBM licensed program that provides extended services for application programs, communication, token-ring interconnection, and device support.

financial PIN. (1) A Personal Identification Number used to identify an individual in some financial transactions. To maintain the security of the PIN, processes and data structures have been adopted for creating, communicating, and verifying PINs used in financial transactions. (2) See also *Personal Identification Number*.

Flash-Erasable Programmable Read-Only Memory (FEPROM). FEPROM is a PROM that has to be erased before it can be changed.

frequency. The rate of signal oscillation, expressed in hertz (cycles per second).

G

Generalized Trace Facility (GTF). GTF is an optional Operating System/Virtual Storage (OS/VS) service program that records significant system events, such as supervisor calls and start I/O operations, for the purpose of problem determination.

global configuration data. Information that specifies general configuration characteristics of the cryptographic hardware components, such as the number of Key-Encrypting Keys, number of data keys, log size, and so forth.

guest profile. (1) In the Transaction Security System products, profile data that is downloaded from the Personal Security Card into the other hardware components. The guest profile temporarily redefines the user's capabilities for that component. (2) See also *Profile*.

H

hardware. The equipment, as opposed to the programming, of a system.

Hardware Initialization and Key Management Utilities. The part of the Workstation Security Services Program that enables you to customize the system, display the status of components, reinitialize the components, manage the command configuration data, manage the profiles, manage the clear cryptographic keys, manage the keys and key storage, manage the signatures, manage the initialization batch file, and perform miscellaneous functions.

hertz (Hz). A unit of frequency equal to one cycle per second.

Note: In the United States, line frequency is 60 Hz or a change in voltage polarity 120 times per second; in Europe, line frequency is 50 Hz or a change in voltage polarity 100 times per second.

holiday table. Information that specifies up to 16 dates on which the cryptographic hardware components cannot be fully used.

host. (1) In this publication, same as host computer or host processor. (2) In a computer network, the computer that usually performs network-control functions and provides end-users with services such as computation and database access. (T) (3) The primary or controlling computer in a multiple-computer installation. (4) A processor that controls all or part of a user-application network. (T) (5) In a network, the processing unit where the access method resides.

host-communication interface. The part of the IBM 4753 control program that permits communication between the 4753 and the System/370 host through the channel adapter.

host-connection data. In the Transaction Security System products, information about the channel between the &BUulwarknm. and the MVS host. For the 4753 Model 1, this data includes the channel address, the number of channel pairs, the channel speed, and

the data transfer mode. For the IBM 4753 Models 2 and 12, this data includes the data transfer mode, channel transfer speed, and the subchannel starting address.

I

IMPORTER key. (1) In the Transaction Security System, a type of DES Key-Encrypting Key that can decipher a key at a receiving mode. (2) Contrast with *EXPORTER key*.

Information Management System. IMS is an IBM licensed program that is an operation on the operating system; this operation provides information management services.

initialize. (1) In programming languages, to give a value to a data object at the beginning of its lifetime. (I) (2) To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine. (A)

Initial Program Load (IPL). (1) IPL is the initialization procedure that causes an operating system to commence operation. (2) The process by which a configuration image is loaded into storage at the beginning of a work day or after a system malfunction. (3) The process of loading system programs and preparing a system to run jobs.

Input/Output (I/O). (1) I/O Pertains to a device whose parts can perform an input process and an output process at the same time. (I) (2) Pertaining to a functional unit or channel involved in an input process, output process, or both, concurrently or not, and to the data involved in such a process.

Integrated Cryptographic Service Facility (ICSF). ICSF is an IBM licensed program that supports the cryptographic hardware feature for the high-end System/390 processor running in an MVS environment.

Interconnect Control Program (ICP). ICP is a communication control program that the 3172 uses.

interface. (1) A shared boundary between two functional units, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate. The concept includes the specification of the connection of two devices having different functions. (T) (2) Hardware, software, or both, that links systems, programs, or devices.

International Organization for Standardization (ISO). ISO is an organization of national standards bodies established to promote the development of standards to facilitate the international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity.

ISA bus. A Personal computer industry standard architecture. The expansion board bus introduced with the IBM Personal Computer and subsequently extended to a 16-bit data bus. See also *Micro Channel* bus.

J

jumper. A wire that joins two unconnected circuits on a printed circuit board.

K

key. In computer security, a sequence of symbols used with a cryptographic algorithm to encrypt or decrypt data.

Key-Encrypting Key (KEK). (1) A KEK is a key used for the encryption and decryption of other keys. (2) Contrast with *data-encrypting key*.

key storage. In the Transaction Security System products, a data file that contains cryptographic keys.

key-storage synchronization. A process that ensures that every Network Security Processor accessed by the application program contains the same key-storage records.

key-synchronization server. The part of the Network Security Processor control program that maintains current and compatible cryptographic keys to be used by multiple network security processors connected on a token-ring network.

key token. In the Transaction Security System security API, a data structure that can contain a cryptographic key, a control vector, and other information related to the key.

Kilobyte (KB). a Kkilobyte is equal to 1024 bytes. See *byte*.

Kilopascals (kPa). Is equal to one thousand pascals. See *pascal*.

Kilovolt ampere (kVA). Kilovolt is a unit of power.

L

LAN/Distributed Processing (LAN/DP). An IBM-licensed program product.

Light-Emitting Diode (LED). A semiconductor chip that gives off visible or infrared light when activated.

link. (1) The logical connection between nodes including the end-to-end control procedures. (2) The combination of physical media, protocols, and programming that connects devices on a network. (3) In computer programming, the part of a program, in some cases a single instruction or an address, that passes control and parameters between separate portions of the computer program. (A) (I) (4) To interconnect items of data or portions of one or more computer programs. (T) (5) In SNA, the combination of the link connection and link stations joining network nodes.

local area network (LAN). A LAN is a computer network located on the user's premises within a limited geographical area. Communication within a Local Area Network is not subject to external regulations; however, communication across the LAN boundary may be subject to some form of regulation.

logical unit (LU). An LU is a port through which an end user accesses the SNA network in order to communicate with another end user, and through which the end user accesses the functions provided by System Services Control Points (SSCPs). An LU can support at least two sessions, one with an SSCP and one with another LU; it can be capable of supporting many sessions with other logical units.

M

make file. A composite file that contains either device configuration data or individual user profiles.

master key (KM). In computer security, the top-level key in a hierarchy of key-encrypting keys.

megabyte (MB). A megabyte is equal to 1 048 576 bytes.

merge file. A file containing information for each of several Personal Security Cards that the Batch Initialization utility initializes.

Message Authentication Code (MAC). (1) A number or value derived by processing data with an authentication algorithm, (2) The cryptographic result of block cipher operations on text or data using a cipher block chaining (CBC) mode of operation, (3) A digital signature code.

Micro Channel bus. A type of bus is used in IBM PS/2 computer Models 50 and higher. This term is used to distinguish these computers from personal computers using a PC I/O channel.

migrate. (1) To move data from one hierarchy of storage to another. (2) To move to a changed operating environment, usually to a new release or a new version of a system.

Modification Detection Code (MDC). In cryptography, the MDC is a number or value that interrelates all bits of a data stream so that, when enciphered, modification of any bit in the data stream results in a new MDC.

Multiple Virtual Storage (MVS). MVS implies MVS/370, the MVS/XA product, and the MVS/ESA product.

Multiple Virtual Storage/Extended Architecture (MVS/XA). The MVS/XA product, consists of MVS/System Product Version 2 and the MVS/XA Data Facility Product, operating on a System/370 processor in the System/370 extended-architecture mode. The MVS/XA product allows virtual storage addressing up to two gigabytes.

multiplexer. (1) A device that takes several input signals and combines them into a single output signal; the output signal allows each of the input signals to be recovered. (T) (2) A device capable of interleaving the events of two-or-more activities, or capable of distributing the events of an interleaved sequence to their respective activities. (A)

Multi-station Access Unit (MAU). An MAU is an IBM token-ring unit that can be used to connect as many as 16 Network Security Processors on a single token ring.

multi-tasking supervisor. The part of the Network Security Processor that manages the system functions and system states, and schedules the software tasks for the network security processor.

multi-user environment. A computer system that provides terminals and keyboards for more than one user at the same time.

N

National Institute of Science and Technology (NIST). This is the current name for the US National Bureau of Standards.

network. (1) A configuration of data-processing devices and software programs connected for information interchange. (2) An arrangement of nodes and connecting branches. (T)

Network Security Processor (IBM 4753). The IBM 4753 is a processor that uses the Data Encryption Algorithm to provide cryptographic support for systems requiring secure transaction processing (and other cryptographic services) at the host computer.

Network Security Processor Control Program. A program that runs in the IBM 4753 Network Security Processor to enable it to process cryptographic commands from the host computer.

IBM Network Security Processor MVS Support Program. An IBM-licensed program that runs in the System/370 host under the MVS/370, MVS/XA, or MVS/ESA operating systems to enable host applications to request cryptographic services in the Network Security Processor.

Network Security Processor Support Utility. Network Security Processor Support Utility is the part of the workstation security services program to support functions relating directly to the Network Security Processor.

Network Security Processor Utilities. The Network Security Processor Utilities are the parts of the Network Security Processor Control Program that enable you to do the following: install the program, customize the system, manage key storage, manage the signon list, and vary the host adapter offline and online.

node. In a network, a point at which one-or-more functional units connect channels or data circuits. (I)

node address. The address of an adapter on a LAN.

O

Online Test System (OLTS). OLTS is a system that allows the user to test I/O devices concurrently with program execution. Tests can be run to diagnose I/O errors, and verify repairs and engineering changes, or run to check devices periodically.

Operating System/2 (OS/2). OS/2 is an operating system for the IBM Personal System/2 computers.

Operating System/400 (OS/400). OS/400 is an operating system for the IBM Application System/400 computers.

Operating System/Virtual Storage (OS/VS). OS/VS is a family of operating systems that controls IBM System/360* and System/370 computing systems. OS/VS includes VS1, VS2, MVS/370, and MVS/XA.

operations log. In the IBM 4753, a file that contains a record of operator activitiesperformed on the 4753.

P

panel. The complete set of information shown in a single image on a display station screen.

parameter. In the security API, one of the values passed to a verb to address a variable exchanged between an application program and the verb.

Pascal (Pa). The stress resulting when a force of one Newton is applied evenly and perpendicularly to an area of one square meter.

password. (1) In computer security, a string of characters known to the computer system and a user; the user must specify it to gain full or limited access to a system and to the data stored within it. (2) In the Transaction Security System products, a string of characters that a user must enter when signing on to a system that uses the Cryptographic Adapter.

path. (1) In a network, any route between any two nodes. A path may include more than one branch. (T) (2) The route traversed by the information exchanged between two attaching devices in a network. (3) A command in IBM Personal Computer Disk Operating System (PC DOS) and IBM Operating System/2 (OS/2) environments that specifies directories to be searched for commands or batch files that are not found by a search of the current directory.

PC-bus. A type of bus that is used in the following IBM Personal Computers: PC/XT, AT, PS/2 Model 25, PS/2 Model 30, and PS/2 Model 30 286.

Personal Identification Number (PIN). (1) In the Transaction Security System, the PIN is the secret number that is used to authenticate the user to the Personal Security Card and the Cryptographic Adapter. (2) In some financial-transaction-authentication systems, the PIN is the secret number given to a consumer with an identification card. This number is selected by the consumer, or it is assigned by the financial institution.

Personal Security Card. An ISO-standard "smart card" with a microprocessor that enables it to perform a variety of DES-based cryptographic functions, such as identifying and verifying users and determining which functions the users can perform. The Security Interface Unit reads and writes information on the Personal Security Card.

physical device ID. In a Transaction Security System public-key implementation, a 16-byte, user-defined field that is stored in and identifies the public-key hardware.

* Trademark of IBM

profile ID. In the Transaction Security System products, one of the four profiles that the Personal Security card contains.

plaintext. (1) Data that has nor been altered by a cryptographic process. (2) Synonym for *cleartext*. See also *ciphertext*.

plug. (1) A connector designed to insert into a receptacle or socket. (2) To insert a connector into a receptacle or socket.

Point-Of-Sale (POS) device. A POS records sales data on machine-readable media at the time a sale is made. (A)

Power-On Self Test (POST). POST is a series of diagnostic tests run automatically by a device when the power is turned on.

private key. (1) In computer security, a key that is known only to the owner and used together with a public-key algorithm to decipher data. The data is enciphered using the related public key. (2) Contrast with *public key*. (3) See also *public-key algorithm*.

procedure call. In programming languages, a language construct for invoking execution of a procedure. (I) A procedure call usually includes an entry name and possible parameters.

profile. Data that describes the significant characteristics of a user, a group of users, or one-or-more computer resources.

profile ID. In the Transaction Security System products, one of the four profiles that the Personal Security Card contains.

profile vector. Transaction Security System public-key implementation, a software data structure that contains default values and configuration information used by various public-key verbs.

profile 0, profile 1, profile 2, profile 3. (1) In the Transaction Security System products, profile data that identifies one of the users of the Cryptographic Adapter or the Security Interface Unit. (2) See also *Profile*.

Programmed Cryptographic Facility (PCF). PCF is an IBM licensed program that provides facilities for enciphering and deciphering data and for creating, maintaining, and managing cryptographic keys.

protocol. (1) A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I) (2) In SNA, the meanings of and the sequencing rules for requests and responses used to manage the network, transfer data, and synchronize the states of network components. (3) A specification for the format and relative timing of information exchanged between communicating parties.

public key. (1) In computer security, a key that is widely known, and used with a public-key algorithm to encrypt data. The encrypted data can be decrypted only with the related private key. (2) Contrast with *private key*. (3) See also *public-key algorithm*.

Public-Key Algorithm (PKA). (1) In computer security, PKA is an asymmetric cryptographic process that uses a public key to encrypt data and a related private key to decrypt data. (2) Contrast with *Data Encryption Algorithm* and *Data Encryption Standard algorithm*. (3) See also *Rivest-Shamir-Adleman algorithm*.

public-key hardware. That portion of the security module in a Cryptographic Adapter containing the microcode and registers for the public-key functions. Depending on the adapter, the hardware can be installed in a workstation or in a Network Security Processor.

public profile. (1) In the Transaction Security System products, profile data that contains the default characteristics for the Cryptographic Adapter and for the Security Interface Unit; the defaults are available when a specific profile is not active. (2) See also *profile*.

R

rack. A free-standing framework that holds equipment.

Random Access Memory (RAM). RAM is a storage device into which data are entered and from which data are retrieved in a non-sequential manner.

Read-Only Memory (ROM). ROM is memory in which stored data cannot be modified by the user except under special conditions.

reason code. (1) A value that provides a specific result as opposed to a general result. (2) Contrast with *return code*.

receptacle. Electrically, a fitting equipped to receive a plug and used to complete an electrical path.

Reduced Instruction-Set Computer (RISC). A RISC computer uses a small, simplified set of frequently used instructions for rapid processing.

request manager. The part of the Access Method that sends cryptographic requests to one or more Network Security Processors.

Request Unit (RU). In SNA, the RU is a message unit containing control information such as a request code or

function management headers, or end-user data, or both.

Resource Access Control Facility (RACF). RACFis an IBM licensed program that enables access control by identifying and verifying the users to the system, authorizing access to protected resources, logging detected unauthorized attempts to enter the system, and logging detected accesses to protected resources.

return code. (1) A code used to influence the execution of succeeding instructions. (A) (2) A value returned to a program to indicate the results of an operation requested by that program. (3) In the Transaction Security System products, a value that provides a general result as opposed to a specific result. (4) Contrast with *reason code*.

Rivest-Shamir-Adleman (RSA) algorithm. RSA is a public-key cryptography process developed by R. Rivest, A. Shamir, and L. Adleman.

RS-232. A specification that defines the interface between data terminal equipment and data circuit-terminating equipment, using serial binary data interchange.

RS-232C. A standard that defines the specific physical, electronic, and functional characteristics of an interface line that uses a 25-pin connector to connect a workstation to a communication device.

RSA algorithm. Rivest-Shamir-Adleman encryption algorithm.

S

security. The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

security API stub. The part of the access method that contains a set of code for each security API verb.

security application programming interface. In Transaction Security System, the security API is the interface through which an application program interacts with an access method or with a workstation interface between an application program and the security server. The interface consists of procedure calls for services (verbs).

Security Interface Unit (IBM 4754). The IBM 4754 is a free-standing device that controls data communication between the Network Security Processor and the Personal Security Card or between the workstation and the Personal Security Card. The Security Interface Unit reads and writes data on the Personal Security Card.

security server. In the Transaction Security System, the part of the Network Security Facility Control Program and the Workstation Security Services Program that provides cryptographic services and key-storage services.

server. On a Local Area Network, a data station that provides facilities to other data stations; for example, a file server, a print server, a mail server. (A)

Server-Requester Programming Interface (SRPI). The SRPI is an Application Programming Interface (API) used by requester and server programs to communicate with the personal computer or host routers.

service clearance. The minimum space required to allow working room for the person installing or servicing a unit.

session. (1) In network architecture, for the purpose of data communication between functional units, all the activities that take place during the establishment, maintenance, and release of the connection. (T) (2) The period of time during which a user of a terminal can communicate with an interactive system (usually, the elapsed time between logon and logoff).

Session-Level Encryption (SLE). SLE is a Systems Network Architecture (SNA) protocol that provides a method for establishing a session with a unique key for that session. This protocol establishes a cryptographic key and the rules for deciphering and enciphering information in a session.

signature verification module. An optional module on the Cryptographic Adapter that provides support for signature verification.

signature verification pen. A pen attached by a cable to the Security Interface Unit for the purpose of identifying and verifying users.

signon list. In the Transaction Security System products, a list that contains the profile IDs and the card IDs of the only users allowed to log onto the IBM 4753. You can create this list with the IBM 4753 support utility.

software configuration utility. One of the utilities supplied with the Workstation Security Services Program that enables you to configure security servers and device drivers by specifying combinations of verbs and functions. By using this utility, you can minimize memory requirements.

string. A sequence of elements of the same nature, such as characters, considered as a whole. (T)

subsystem. A secondary or subordinate system, usually capable of operating independently of, or asynchronously with, a controlling system. (T)

supervisor router. The part of the Workstation Security Services Program that schedules each task for the program.

system. In data processing, a collection of people, machines, and methods organized to accomplish a set of specific functions. (A) (I)

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

System Authorization Facility (SAF). SAF is a program that provides access to the resource access control facility or its equivalent.

system error log. In the IBM 4753, a file containing a record of error messages that have been displayed.

Systems Network Architecture (SNA). SNA describes logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks. **Note:** The layered structure of SNA allows the ultimate origins and destinations of information, that is, the end users, to be independent of and unaffected by the specific SNA network services and facilities used for information exchange.

T

tag in. A unidirectional control line bus that is part of the System/360 or System/370 Parallel Channel Interface. This bus passes control signals from the control unit to the host.

tag out. A unidirectional control line bus that is part of the System/360 or System/370 Parallel Channel Interface. This bus passes control signals from the host to the control unit.

Terminate and Stay Resident (TSR). A TSR is a program that remains in memory after it has run and returned control to the operating system. A TSR program can be started and stopped by another program without disturbing that program's processing.

throughput. (1) A measure of the amount of work performed by a computer system over a given period of time; for example, number of jobs per day. (A) (I) (2) A measure of the amount of information transmitted over a network in a given period of time; for example, a network's data-transfer-rate is usually measured in bits per second.

Time Sharing Option (TSO). TSO is an IBM licensed program that is an option on the operating system; for a System/370 processor, the option provides interactive time sharing from remote terminals.

token. (1) In a Local Area Network, the symbol of authority passed successively from one data station to another to indicate the station is temporarily in control of the transmission medium. (T) (2) A string of characters treated as a single entity.

token-ring adapter. The circuit card and its associated software that enables a communicating device to communicate over a local area network.

token-ring network. (1) A ring network that allows unidirectional data transmission between data stations, by a token passing-procedure, such that the transmitted data return to the transmitting station. (T) (2) A network that uses a ring technology, in which tokens are passed in a circuit from node to node. A node that is ready to send can capture the token and insert data for transmission.

trace file. In the IBM 4753, a file that contains a record of trace information for the selected servers.

U

Unique Key Per Transaction (UKPT). UKPT is a cryptographic process that can be used to decipher PIN blocks in a transaction.

user exit. That point in an IBM supplied program at which a user-exit routine can be given control.

user-authorization table. In the Transaction Security System products, information that specifies the options the operator is permitted to access.

user-exit routine. A user-written routine that receives control at predefined user-exit points.

user ID. User identification.

userid. A string of characters that uniquely identifies a user to the system.

utility program. A computer program in general support of computer processes. (T)

V

verb. A function that has an entry-point-name and a fixed-length parameter list. The procedure call for a verb uses the standard syntax of a programming language.

virtual machine (VM). A functional simulation of a computer and its associated devices. Each virtual machine is controlled by a suitable operating system. VM controls concurrent execution of multiple virtual machines on one host computer.

VISA. A financial institution consortium which defines four PIN block formats and a method of PIN verification.

W

workstation. A terminal or microcomputer, usually one that is connected to a mainframe or to a network, at which a user can perform applications.

Workstation Cryptographic Services Program. An IBM-licensed program that runs in the workstation under OS/2, AIX, or an equivalent product to support the Cryptographic Adapter, the Security Interface Unit, and the Personal Security Card.

Numerics

3172. IBM 3172 Interconnect Controller. The 4753 is based on the 3172.

4707 display. (1) A monochrome display. (2) In an IBM 4753 environment, the 4707 display shows the 4753 messages and codes, including the diagnostic and control program messages and codes.

4753. IBM 4753 Network Security Processor.

4754. IBM 4754 Security Interface Unit.

4755. IBM 4755 Cryptographic Adapter.

Index

Special Characters

(CSNBMKP) Master_Key_Process 2-46
(PKA_Key_Record_Delete) CSNDKRD 7-13
(Retained_Key_Delete) CSNDRKD 7-21
(Retained_Key_List) CSNDRKL 7-22

A

Access Control Initialization (CSUAACI) 2-13
Access Control Maintenance (CSUAACM) 2-16
Access Control, CCA 2-2
Access_Control_Initialization (CSUAACI) 2-13
Access_Control_Maintenance (CSUAACM) 2-16
American National Standards Institute (ANSI)
  X3.106 (CBC) method D-3
  X9.19 method D-7
  X9.23 method D-5
  X9.9 method D-7
asymmetric keys 5-5
attributes 5-7
automated teller machine 8-2

C

calculation methods, PIN 8-7
carriage return (CR) B-16
chaining vector 6-3
chaining vector record format B-13
ciphering
  DES key verification algorithm D-1
  keys 5-7
  methods
    3624 PIN E-2, E-4
    3624 PIN offset E-3
    ANSI X3.106 (CBC) D-3
    German Bank Pool Institution PIN E-5
    Interbank PIN E-7
    message authentication code (MAC) D-7
    NL-PIN-1 E-4
    VISA PIN validation value (PVV) E-6
clear keys 5-12
coding procedure calls 1-7
common parameters 1-10
confidentiality, data 6-1
control vectors (CVs)
  bit map
    EXPORT bit C-6
    format C-4
    gks bits C-6
    IMPORT bit C-6
    Key-part bit C-8
    parity bits C-8
    PIN-block format bits C-7
control vectors (CVs) *(continued)*
  bit map *(continued)*
    XLATE bit C-6
  Changing
    pre-exclusive-OR technique C-11
  checking 5-4
  default values 5-6
  description 5-3
  determining values C-5
  key form bits, fff C-5
  key separation 5-3
  keywords 5-7
  multiply deciphering keys C-8
  multiply enciphering keys C-8
  specifying values C-5
CR (carriage return) B-16
cryptographic engine 1-3
Cryptographic_Facility_Control (CSUACFC) 2-22
Cryptographic_Facility_Query (CSUACFQ) 2-26
CSNBCKI (Clear_Key_Import) 5-16
CSNBCPA (Clear_PIN_Generate_Alternate) 8-18
CSNBCPE (Clear_PIN_Encrypt) 8-12
CSNBDEC (Decipher) 6-4
CSNBDKG (Diversified_Key_Generate) 5-20
CSNBDKM (Data_Key_Import) 5-18
CSNBDKX (Data_Key_Export) 5-17
CSNBENC (Encipher) 6-7
CSNBEPG (Encrypted_PIN_Generate) 8-24
CSNBKEX (Key_Export) 5-23
CSNBKGN 5-13
CSNBKGN (Key_Generate) 5-25
CSNBKIM (Key_Import) 5-31
CSNBKPI (Key_Part_Import) 5-33
CSNBKRC (DES_Key_Record_Create) 7-4
CSNBKRL (Key_Record_List) 7-7
CSNBKRR (Key_Record_Read) 7-9
CSNBKRW (Key_Record_Write) 7-10
CSNBKTB (Key_Token_Build) 5-38
CSNBKTC (Key_Token_Change) 5-41
CSNBKTR (Key_Translate) 5-43
CSNBKYT (Key_Test) 5-35
CSNBMGN (MAC_Generate) 6-10
CSNBMVR (MAC_Verify) 6-13
CSNBOWH (One_Way_Hash) 4-10
CSNBPGN (Clear_PIN_Generate) 8-15
CSNBPTR (Encrypted_PIN_Translate) 8-29
CSNBPVR (Encrypted_PIN_Verify) 8-34
CSNBRNG (Random_Number_Generate) 5-45
CSNDDSG (Digital_Signature_Generate) 4-4
CSNDDSV (Digital_Signature_Verify) 4-7
CSNDKRC (PKA_Key_Record_Create) 7-11

© Copyright IBM Corp. 1997-98

CSNDKRL (PKA_Key_Record_List) 7-15
CSNDKRR (PKA_Key_Record_Read) 7-17
CSNDKRW (PKA_Key_Record_Write) 7-19
CSNDKTC (PKA_Key_Token_Change) 3-18
CSNDPKB (PKA_Key_Token_Build) 3-12
CSNDPKG (PKA_Key_Generate) 3-6
CSNDPKH (PKA_Public_Key_Hash_Register) 3-22
CSNDPKI (PKA_Key_Import) 3-10
CSNDPKR (PKA_Public_Key_Register) 3-24
CSNDPKX (PKA_Public_Key_Extract) 3-20
CSNDSBC (SET_Block_Compose) 8-40
CSNDSBD (SET_Block_Decompose) 8-44
CSNDSYG (PKA_Symmetric_Key_Generate) 5-49
CSNDSYI (PKA_Symmetric_Key_Import) 5-52
CSNDSYX (PKA_Symmetric_Key_Export) 5-47
CSUAACI 2-13
CSUAACI (Access_Control_Initialization) 2-13
CSUAACM 2-16
CSUAACM (Access_Control_Maintenance) 2-16
CSUACFC (Cryptographic_Facility_Control) 2-22
CSUACFQ (Cryptographic_Facility_Query) 2-26
CSUALCT 2-38
CSUALCT (Logon_Control) 2-38
CSUAMKD 2-42
CSUAMKD (Master_Key_Distribution) 2-42

D

DASD (direct access storage device) B-14
data
  confidentiality 6-1
    ensuring 6-1
  integrity 6-1, 6-2
  segmented 6-3
  validation 8-7
DATA-class keys 5-6
deactivating keys 3-18, 7-5, 7-13, 7-21
decimalization table 8-7
defaults, control vectors 5-6
DES key storage initialization 2-36
DES_Key_Record_Delete (CSNBKRD) 7-5
DES_Key_Record_List(CSNBKRL) 7-7
DES_Key_Record_Read (CSNBKRR) 7-9
DES_Key_Record_Write (CSNBKRW) 7-10
device key 1-4
direct access storage device (DASD) B-14

E

entry-point names 1-7
environment identifier 2-10
EX (exportable) keys 5-3
exit_data parameter 1-10
exit_data_length parameter 1-10
exportable (EX) keys 5-3
exporting, description 5-14, C-11
external
  key tokens
    building 5-38
    format B-4
    Key_Token_Build verb 5-38
  key tokens, description 5-10
  keys 5-3, 5-14
extraction methods, financial PIN 8-10

F

financial personal identification number (PIN)
  3624 PIN (CSNBPVR) 8-34
  blocks
    3624 8-9, E-8
    and PIN calculation methods E-1
    description 8-5, 8-9, E-8
    format control 8-9
    ISO-0 8-9, E-9
    ISO-1 8-9, E-10
    ISO-2 8-9, E-11
    multiple 8-9
    profile 8-9
    reformatting 8-29
  calculation
    3624 PIN E-2, E-4
    3624 PIN Offset E-3
    descriptions 8-7, E-1
    German Bank Pool Institution PIN E-5
    Interbank PIN E-7
    supporting multiple PIN calculation methods 8-7
    VISA PVV E-6
  data array
    decimalization table 8-7
    transaction security data 8-8
    validation data 8-8
  description 8-2
  extraction methods 8-10
  format control 8-9
  generating clear PIN 8-15
  institution-assigned 8-34
  key types 8-6
  key-usage bits 8-6
  personal account number (PAN) 8-11
  PIN profile
    format control element 8-9
    pad digit element 8-9
    PIN-block format element 8-9
  processing
    description 8-2
    extraction methods 8-9
    security 8-5
    supporting multiple PIN calculation methods 8-7
    verbs 8-2
  reformatting 8-29

financial personal identification number (PIN) (continued)
security 8-5
verbs
CSNBCPA (Clear_PIN_Generate_Alternate) 8-18
CSNBCPE (Clear_PIN_Encrypt) 8-12
CSNBEPG (Encrypted_PIN_Generate) 8-24
CSNBPGN (Clear_PIN_Generate) 8-15
CSNBPTR (Encrypted_PIN_Translate) 8-29
CSNBPVR (Encrypted_PIN_Verify) 8-34
flag bytes B-4
format
chaining_vector record B-13
control, financial PIN 8-9
key record list data set B-16
key storage record B-14
key tokens
external B-4
internal B-3
null B-2

I

IM (importable) keys 5-3
importable (IM) keys 5-3
importing, description 5-14, C-11
initializing key storage 2-36
input/output (I/O) parameters 1-8
installing keys 5-11
intermediate PIN block (IPB) E-9
internal 5-10
key tokens
building 5-38
copying into application data storage 7-9
copying into key storage 7-10, 7-19
format B-3
Key_Token_Build verb 5-38
Introduction of master key parts 2-8
IPB (intermediate PIN block) E-9
ISO-0 PIN block format E-9
ISO-1 PIN block format E-10
ISO-2 PIN block format E-11

K

key shares 2-9
key storage
description 5-15
key-record-list data set
creating 7-7, 7-15
format B-16
verbs 5-11
key storage initialization 2-36
key tokens
assembling 5-38
key tokens (continued)
changing 3-18
contents 5-8
deleting 3-18, 7-13, 7-21
description 5-8
external 5-10
Key_Token_Build verb 5-38
PKA_Key_Record_Delete service 7-13
PKA_Key_Token_Change verb 3-18
flag byte 1 B-4
flag byte 2 B-4
format 5-8, B-1
internal 5-10
Key_Token_Build verb 5-38
PKA_Key_Record_Delete service 7-13
PKA_Key_Token_Change verb 3-18
Key_Token_Build verb 5-38
listing 7-22
null 5-10
Record-Validation Value (RVV) B-2
token-validation value (TVV) B-2
key-encrypting-key-class keys 5-6
key-export operation 5-14
key-import operation 5-14
key-management keys
Common Cryptographic Architecture support 5-1
key-processing and key-storage verbs 5-11
DES_Key_Record_Delete (CSNBKRD) 7-5
Key_Record_List (CSNBKRL) 7-7
Key_Record_Read (CSNBKRR) 7-9
Key_Record_Write (CSNBKRW) 7-10
PKA_Key_Record_Delete (CSNDKRD) 7-13
PKA_Key_Record_List (CSNDKRL) 7-15
PKA_Key_Record_Read (CSNDKRR) 7-17
PKA_Key_Record_Write (CSNDKRW) 7-19
Retained_Key_Delete (CSNDRKD) 7-21
Retained_Key_List (CSNDRKL) 7-22
Key_DES_Key_Record_Delete (CSNBKRD) 7-5
Key_Generate (CSNBKGN) 5-13
Key_Record_List (CSNBKRL) 7-7
Key_Record_List (CSNDKRL) 7-15
Key_Record_Read (CSNBKRR) 7-9
Key_Record_Write (CSNBKRW) 7-10
Key_Storage_Initialization (&vblcksi.) 2-36
Key_Token_Build (CSNBKTB) 5-38
keys
activating 3-18
asymmetric 5-5
ciphering 5-7
clear 5-12
control vectors 5-3
deactivating 3-18
deleting 3-18, 7-13, 7-21
double-length 5-6
exportable (EX) 5-3

keys *(continued)*
exporting 5-14
external 5-3
generating 5-12
identifiers 5-10
importable (IM) 5-3
importing 5-14
installing 5-11
key management 5-1, 5-2
key storage initialization 2-36
key-usage keywords 5-7
labels
definition 5-10
length 5-27
listing 7-22
managing 5-1, 5-2
master key loading 2-46
multiply-deciphered 5-14
multiply-deciphered the 5-3
multiply-enciphered 5-3, 5-12
operational (OP) 5-3
parity 5-3
parts
generating 5-12
secure 5-11
processing
verbs 5-11
re-enciphering 3-18
records
deleting 7-5, 7-13, 7-21
DES_Key_Record_Deleteservice 7-5
Key_Record_List service 7-7
Key_Record_Read service 7-9
Key_Record_Write service 7-10
listing 7-7, 7-15, 7-22
PKA_Key_Record_Delete service 7-13
PKA_Key_Record_List service 7-15
PKA_Key_Record_Read service 7-17
PKA_Key_Record_Write service 7-19
reading 7-9, 7-17
Retained_Key_Delete service 7-21
Retained_Key_List service 7-22
writing 7-10, 7-19
separation 5-3
storing 5-15
symmetric 5-5
types
and verbs 5-6
asymmetric 5-5
DATA 5-6
DATA-class keys 5-6
description 5-5
EXPORTER 5-6
IKEYXLAT 5-6
IMPORTER 5-6
IPINENC 8-6
key-encrypting-key-class keys 5-6
keys *(continued)*
types *(continued)*
key-usage keywords 5-7
MAC 5-6
MAC-class keys 5-6
MACVER 5-6
OKEYXLAT 5-6
one-way key distribution channels 5-5
OPINENC 8-6
PIN security 8-6
PINGEN 8-6
PINVER 8-6
symmetric 5-5
usage
bits 8-6
key form 5-13
key type 5-13
keywords 5-7
verification pattern 5-11
verifying 5-11
keywords, key-usage 5-7

L

LF (line feed) B-16
line feed (LF) B-16
listing keys 7-22
loading a master key 2-46
Logging on and logging off 2-7
Logon Control (CSUALCT) 2-38
Logon_Control (CSUALCT) 2-38

M

m-of-n master key shares 2-9
MAC_Generate (CSNBMGN) 6-3
MAC_Verify (CSNBMVR) 6-3
MACVER key type, MAC_Verify verb 5-6
managing
DES keys
Common Cryptographic Architecture 5-1
master key 1-4
current master key 2-8
environment identifier 2-10
Introduction of master key parts 2-8
m-of-n 2-9
master key cloning 2-9
new master key 2-8
old master key 2-8
Random generation of a new master key 2-9
shares 2-9
Understanding and managing master keys 2-8
master key cloning 2-9
master key loading 2-36, 2-46
master key verification pattern 2-8

Master_Key_Distribution (CSUAMKD) 2-42
Master_Key_Process (CSNBMKP) 2-46
multiple PIN calculation methods 8-7
multiply-deciphered keys 5-3, 5-14
multiply-enciphered keys 5-3, 5-12

N

null key token 5-10, B-2

O

OCV (output chaining value) D-3
OP (operational) keys 5-3, 5-14
operating environments 1-7
operational (OP) keys 5-14
operational keys (OP) 5-3
output chaining value (OCV) D-3

P

pad digit 8-9
PAN (personal account number) 8-11
parity, key 5-3
personal account number (PAN) 8-11
PIN block-encrypting key 8-6
PKA_Key_Record_Delete (CSNDKRD) 7-13
PKA_Key_Record_List(CSNDKRL) 7-15
PKA_Key_Record_Read (CSNDKRR) 7-17
PKA_Key_Record_Write (CSNDKRW) 7-19
PKA_Key_Token_Change (CSNDKTC) 3-18
PKA_PKA_Key_Record_Delete (CSNDKRD) 7-13
PKA_Retained_Key_Delete (CSNDRKD) 7-21
PKA_Retained_Key_List (CSNDRKL) 7-22
pre-exclusive-OR technique C-11
procedure calls 1-7
processing a master key 2-46
profiles
  activating
  Header 2-4
  Overview 2-3
  Passphrase verification protocol D-16
  Passphrases 2-7
  personal identification number (PIN)
    PIN profile 8-9
  Profile data structures B-21
  Verbs for initialization and management 2-5
pseudonyms 1-7, F-1

R

Random generation of a new master key 2-9
re-enciphering keys 3-18
reason codes A-1
reason_code parameter 1-10
record-validation value (RVV) B-2
Required Commands
  Description B-19
  List of access control point codes G-1
  Overview 2-3
Retained_Key_Delete (CSNDRKD) 7-21
Retained_Key_List (CSNDRKL) 7-22
return_code parameter 1-10
Roles, access control
  Default role 2-3
  Overview 2-2
  Role data structures B-18
  Verbs for initialization and management 2-5
rule_array parameter description 1-11
RVV (record-validation value) B-2

S

segmented data 6-3
symmetric keys 5-5

T

token-validation value (TVV) 5-9, B-2
trial pin 8-2
TVV (token-validation value) 5-9, B-2

U

Understanding and managing master keys 2-8
  current master key 2-8
  new master key 2-8
  old master key 2-8

V

validation data 8-7
verbs
  common parameters
    exit_data 1-10
    exit_data_length 1-10
    reason_code 1-10
    return_code 1-10
    rule_array 1-11
  data confidentiality 6-1
  data integrity 6-1
  descriptions 1-7
  direction 1-9
  entry-point names 1-7
  list of 1-7
  parameters 1-9
  procedure calls 1-7
  processing A-1
  pseudonyms 1-7, F-1
  reason codes A-1
  return codes A-1
  supported environments 1-7
  type 1-10

verbs *(continued)*
  variables 1-9
verification pattern 5-11

X

X3.106 (CBC) method D-3

Printed in U.S.A.