IBM

CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors

Releases 2.53, 2.54, 3.20, and 3.23

Fifteenth edition, July 2005

This edition describes the IBM Common Cryptographic Architecture (CCA) Basic Services API for Releases 2.53, 2.54, 3.20, and 3.23.

Contents

Figures

Tables

About this document

This document is intended for systems analysts, applications analysts, and application programmers who evaluate or create programs that employ the IBM® Common Cryptographic Architecture (CCA) application programming interface (API). The IBM CCA API implementations described in this document are listed in the following table. IBM also offers a CCA implementation on the IBM eServer™ zSeries® that is described in other publications.

IBM only provides the IBM 4764 Cryptographic Coprocessor on IBM eServer i5 running IBM i5/OS™. i5/OS is the next generation of IBM OS/400®.

In this document the terms iSeries™ and OS/400 are synonymous with eServer i5 and i5/OS, respectively.

You might find it useful to review the *IBM 4758 Cryptographic Coprocessor General Information Manual*. It discusses these topics important to the understanding of the information presented in this document:
- The IBM 4758 Cryptographic Coprocessor
- The IBM 4764 Cryptographic Coprocessor
- An overview of cryptography
- Supported cryptographic functions
- System hardware features and software
- Organization of the relevant publications

Revision history

Fifteenth edition, July 2005, Common Cryptographic Architecture Support Program, Releases 2.53, 2.54, 3.20, and 3.23

This edition describes the IBM CCA Basic Services API for Releases 2.53, 2.54, 3.20, and 3.23. Table 1 describes the APIs discussed in this document.

Table 1. CCA API release availability matrix

Release	Coprocessor product	Software platform and offering	Replaces release...
3.23	• IBM 4764-001 PCI-X Cryptographic Coprocessor • IBM eServer i5 hardware feature code 4806	IBM i5/OS Option 35 Cryptographic Service Provider, version 5, release 3	Release 3.20 (see "Other Release 3.23 changes" on page xiv for additional release 3.23 changes to the CCA API
3.20	• IBM 4764-001 PCI-X Cryptographic Coprocessor (Release 3.20 is the first release to support this coprocessor) • IBM eServer i5 hardware feature code 4806	IBM i5/OS Option 35 Cryptographic Service Provider, version 5, release 3	Release 2.54 (see "Other Release 3.20 changes" on page xiv for additional release 3.20 changes to the CCA API)

Table 1. CCA API release availability matrix (continued)

2.54	• IBM 4758-023 PCI Cryptographic Coprocessor on IBM eServer iSeries • IBM eServer iSeries hardware feature 4801	Either of the following: • IBM i5/OS version 5, release 3 • Option 35 Cryptographic Service Provider, version 5, release 2	Release 2.52
2.53	• IBM 4758-002 PCI Cryptographic Coprocessor • IBM 4758-023 PCI Cryptographic Coprocessor	Windows® 2000	Release 2.42
	• IBM 4758-002 PCI Cryptographic Coprocessor • IBM 4758-023 PCI Cryptographic Coprocessor • IBM eServer hardware feature code 4963	• IBM AIX® 4.3.3 • IBM AIX 5.3	

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

Other Release 3.23 changes

Release 3.23 changes to the CCA API include:

- The Key-Encryption-Translate (CSNBKET) verb is included. (This verb was in Release 2.54 but was not in Release 3.20.)
- Addition of the **DUKPT-IP**, **DUKPT-OP**, and **DUKPT-BH** keywords to the Encrypted_PIN_Translate verb and **DUKPT-IP** keyword to the Encrypted_PIN_Verify verb to perform UKPT processing using triple DEA.
- Corrections to “Reason codes that accompany return code 4” on page 336.
- Other minor editorial changes.

Fourteenth edition, April 2005, Common Cryptographic Architecture Support Program, Releases 2.53, 2.54, and 3.20

This edition describes the IBM CCA Basic Services API for Releases 2.53, 2.54, and 3.20. Table 1 on page xiii describes the APIs discussed in this document.

Other Release 3.20 changes

Release 3.20 changes to the CCA API include:

- The Key_Encryption_Translate (CSNBKET) verb is not included (it was in Release 2.54).
- The Cryptographic_Facility_Control verb extended with the **ERRINJ1** keyword. iSeries does not support this function.
- The Cryptographic_Facility_Query verb reports the coprocessor operating system identification and level with keyword **DBREGS1**.

- The hash length in the Digital_Signature_Generate verb is restricted when formatting with **ZERO-PAD** and employing a key-management capable private key.
- Addition of the Prohibit_Export_Extended (CSNBPEXX) verb to extend the functionality of the Prohibit_Export verb to external keys.
- Addition of the **ADJUST** and **NOADJUST** keywords to the Key_Test verb so keys can be validated independent of, or dependent on, the setting of the parity bits in a DES key.
- Addition of the Key_Test_Extended (CSNBKYTX) verb to extend the functionality of the Key_Test verb to external keys.
- Support for a 19-digit PAN in the CVV_Generate (CSNBCSG) and CVV_Verify (CSNBCSV) verbs with keyword **PAN-19**.

Thirteenth edition, February 2005, Common Cryptographic Architecture Support Program, Release 2.54

This edition of the *IBM PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide Release 2.53: IBM xSeries® and pSeries® PCICC Feature*, replaces Release 2.53 of December 2004. Release 2.54 is only available for the IBM eServer iSeries. Changes include:

- The addition of the Key_Encryption_Translate (CSNBKET) verb. This verb translates an encrypted double-length external DATA key (with an all-zero control vector) into the following formats:
  - CBC encryption to CCA key-encryption
  - CCA key-encryption to CBC encryption
- A new appendix, “Observations on Secure Operations,” containing security pointers adapted from the *CCA Support Program Installation Manual*.

Twelfth edition, December 2004, Common Cryptographic Architecture Support Program, Release 2.53

This edition of the *IBM PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide Release 2.52: IBM iSeries PCICC Feature* replaces Release 2.52 of April 2004. This release improves security in two ways:

- To create a particular RSA private-public key pair using regeneration data, you must authorize a new control point. This is described in the PKA_Key_Generate verb section on “Required Commands.”
- If you attempt to use an RSA private key with the CLONE attribute, the following verbs terminate with return code 8, reason code 64 (decimal):
  - PKA_Decrypt
  - PKA_Symmetric_Key_Import
  - SET_Block_Decompose

Eleventh edition, April, 2004, Common Cryptographic Architecture Support Program, Release 2.52

This revision to the February 2004 edition of the *IBM 4758 CCA Basic Services Reference and Guide for the IBM 4758 Models 002 and 023*, Release 2.52, replaces the February 2004 Release 2.51 edition. Release 2.52 is only available for the IBM eServer iSeries. Incorporated changes include:

- Addition of a second set of issuer-master key parameters with revised processing in the PIN_Change/Unblock (CSNBPCU) verb. The processing changes are further described in “Visa and EMV-related smart card formats and processes” on page 437.
- Documentation of the **RESETBAT** rule-array keyword in the Cryptographic_Facility_Control (CSUACFC) verb you use to reset the indication of a low battery. This capability was added with Release 2.41.
- In Appendix A, removal of return code 12, reason code 093 (decimal).

Tenth edition, February 2004, Common Cryptographic Architecture Support Program, Release 2.51

The tenth edition of the *IBM 4758 CCA Basic Services Reference and Guide Release 2.51 for the IBM 4758 Models 002 and 023* describes the *Common Cryptographic Architecture* (CCA) application programming interface (API) that is supported by the PCI Cryptographic Coprocessor feature available with IBM eServer iSeries and OS/400 Option 35, CCA CSP.

The document also includes updates and corrections to the previous editions for Release 2.50, Release 2.41 and earlier. The revision bar, as shown at the left, marks important changes and extensions to material previously published in the ninth edition of the *Basic Services* document.

Release 2.51 for the IBM eServer iSeries includes these additional and modified EMV smart-card-related capabilities enhancing the earlier Release 2.50:

1. Addition of the *tree format key-diversification system* to the Diversified_Key_Generate and PIN_Change/Unblock verbs defined in Book 2 of the *EMV 2000 Integrated Circuit Card Specification for Payment Systems*, Annex A1.3.
2. The double-length issuer master-key in the Diversified_Key_Generate and PIN_Change/Unblock verbs must have unequal halves.
3. The issuer master-key control-vector encoding is extended to support use of the **DALL** combination in the PIN_Change/Unblock verb.
4. The key-generating key control-vector encoding is extended to support use of **DDATA**, **DMAC**, and **DMV** encodings provided the control vector for the generated key has a conforming control vector.
5. Extension of the Message Authentication Code (MAC) MAC_Generate and MAC_Verify verbs to support EMV-required post-padding of a message.
6. Corrected the order of the parameters on the Secure_Messaging_for_PINs verb. The PIN_encrypting_key_identifier follows the input_PIN_block parameter.

Release 2.50 incorporated these capabilities and changes:

1. Functions to support EMV-compatible smart cards.
   - Support of the PIN Change/Unblock function described in Version 1.4.0 of the *Visa Integrated Circuit Card Specification Manual*, Section C.11
   - Support of the key-generation function used for secure messaging described in Version 1.4.0 of the *Visa Integrated Circuit Card Specification Manual*, Section B.4
   - Encryption of PINs and keys for inclusion in smart-card transactions with EMV-compatible smart cards

   This support is provided through:
   - A new verb, PIN_Change/Unblock (CSNBPCU), to create a PIN block to change the PIN accepted by a smart card

   - An extension to the Diversified_Key_Generate (CSNBDKG) verb enabling session-key generation for secure messaging
   - A new verb, Secure_Messaging_for_Keys (CSNBSKY), to encrypt a key under a session key
   - A new verb, Secure_Messaging_for_PINs (CSNBSPN), to encrypt a PIN under a session key
   - The next item relating to ISO 9796-2 digital signature verification
2. An extension to the PKA_Encrypt (CSNDPKE) verb enabling verification of digital signatures with any hash formatting method (for example, ISO 9796-2) through the public-key enciphering of data in the zero-pad format.

Ninth edition, revised September, 2003, Common Cryptographic Architecture Support Program, Release 2.41

The *revised* Release 2.41 document, dated September, 2003, contains minor editorial changes and these corrections:

- Figure 24 on page 389 is changed to indicate that a SECMSG key is always double length (“fff” bits changed to “FFF”).
- Figure 24 on page 389 is changed to reflect that key-encrypting keys, bits 35-37, must be B'000'. The text in item 2 of section “Specifying a control-vector-base value” on page 391 which previously described these bits has been removed. Testing for these control vector bits had not been implemented.
- The padding for a Current Key Serial Number must be four bytes of X'00' rather than four space characters as previously stated in “Current key serial number” on page 274.

Ninth edition, revised August 2002, Common Cryptographic Architecture Support Program, Release 2.41

This *revised* Release 2.41 document incorporates corrected information about the name for a Retained RSA key and other minor editorial changes.

Eighth edition, revised, Common Cryptographic Architecture Support Program, Release 2.41

The *revised* Release 2.41 document incorporates additional information concerning access controls (see “Using CCA access-control” on page 16) and other minor editorial changes.

Eighth edition, Common Cryptographic Architecture Support Program, Release 2.41

The major items changed, extended, or added in Release 2.41 include:

- The Key_Export, Key_Import, Data_Key_Export, and Data_Key_Import verbs now require the exporter or importer key to have unique key-halves when importing or exporting a key with unequal halves. You can regress to less-secure operation which does not enforce the restriction by activating an additional access control command point.
- The Key_Part_Import verb has been modified in two ways:
  – For double-length keys, unless a new access-control point is enabled in the governing role, the previously accumulated key-value and the resulting key-value must both have equal, replicated key-halves or both have unequal key-halves. This test is ignored if the previously accumulated key has all key

bits other than parity bits set to zero. This increases security by guaranteeing that the strength of the key is not modified when combining the new key part.

Replicated key-halves means that the first part (half) and the last part of a double-length data-encryption standard (DES) key have equal values and thus perform as though the key were single length.

– Additional keywords are added to the rule_array that permit enforcing separation between individuals who can update the accumulated key and one who can make the key operational (that is, switch off the control-vector key-part bit). The Cryptographic Node Management utility is not updated to take advantage of this extension.

- The Encrypted_PIN_Generate verb (CSNBEPG) has been extended to include support of the 3624 PIN-calculation method through use of the **IBM-PIN** keyword.
- The Encrypted_PIN_Verify verb (CSNBPVR) has been extended to optionally enforce ensuring that PINs are four digits in length when using the VISA-PVV calculation method using the **VISAPVV4** keyword.
- Host-side key caching, which has been performed since Release 2.10, can be switched off using an environment variable. This can be important where a key can be updated by one process and used by one or more other concurrent processes. See “Host-side key caching” on page 6.
- Fixes have been applied to the Diversified_Key_Generate, Encrypted_PIN_Translate, and Encrypted_PIN_Verify verbs. The control-vector checking is corrected to properly account for non-default control-vector values. The Encrypted_PIN_Translate verb now returns reason code 154 instead of 43 (decimal).
- In Windows NT® and Windows 2000 environments, the code is repaired to permit multi-threaded support of multiple coprocessors.
- New drivers are supplied for AIX which support 32-bit and 64-bit environments.
- The Cryptographic Node Management (CNM) utility is modified to prohibit use of key lengths greater than 1024 bits when performing master-key cloning. You can create an application to clone keys having any of the CSS, CSR, and SA keys longer than 1024-bits. See “Establishing master keys” on page 27.
- The PKA_Key_Token_Change verb now returns return code 0 and reason code 0 if you request to update a key token that contains only a public key. A key token containing only a public key is legitimate, but the PKA_Key_Token_Change verb has no effect on such a key token. The verb used to return reason code 8 if the token only contained public-key information.
- The command names listed in this book, in the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*, and in the Cryptographic Node Management utility have been made the same.
- The Key_Token_Change and DES_Key_Record_Create verbs now work correctly with master keys having 3 unique parts (the CCA master keys are triple length).
- The diagnostic trace facility has been removed from the “SECY” DLL/shared-library. If tracing is required in the future for diagnostic purposes, IBM can supply tracing code upon customer agreement to install such code.

How this document is organized

The document includes these topics:

- Chapter 1, “Introduction to programming for the IBM Common Cryptographic Architecture” 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 section also explains the master-key concept and administration, and introduces CCA DES key-management.
- Chapter 3, “RSA key-management” 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, “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 MAC services are described.
- Chapter 7, “Key-storage mechanisms” explains how to use key labels and how to employ key storage.
- Chapter 8, “Financial services support” explains services for the financial services industry including PIN-processing services and credit-card validation functions.
- Appendix A, “Return codes and reason codes” describes the return codes and reason codes issued by the coprocessor.
- Appendix B, “Data structures” describes the various data structures for key tokens, 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 system verbs calculation methods and data formats” describes processes and formats implemented by the PIN-processing support.
- Appendix F, “Verb list” lists the supported CCA verbs.
- Appendix G, “Access-control-point codes” lists the access-control points used by the various verbs and suggests appropriate security settings and considerations.
- Appendix H, “Observations on secure operations,” on page 449 describes observations on secure operations.
- “Notices” on page 459 provides legal notices.
- “Trademarks” on page 463 provides trademark information.
- “List of Abbreviations” on page 465 provides a list of abbreviations.
- “Glossary” on page 467 contains definitions of terms.

Related publications

In addition to the documents listed below, you might want to refer to other CCA product publications which might be of use with applications and systems you might develop for use with the IBM 4758 and IBM 4764. While there is substantial commonality in the API supported by the CCA products, and while this document seeks to guide you to a common subset supported by all CCA products, other individual product publications might provide further insight into potential issues of compatibility.

IBM 4758 PCI Cryptographic Coprocessor

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
The General Information Manual is suggested reading prior to reading this document.

IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide, Release 2.52
Describes the predecessor CCA API supporting the IBM 4758.

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

IBM 4758 PCI Cryptographic Coprocessor Installation Manual
Describes the physical installation of the IBM 4758 and the battery-changing procedure.

Building a High-Performance Programmable, Secure Coprocessor
A research paper describing the security aspects and code loading controls of the IBM 4758.

Custom Programming for the IBM 4758
The Library portion of the IBM 4758 Web site also includes programming information for creating applications that perform within the IBM 4758. See the reference to Custom Programming under the Publications heading. The IBM 4758 Web site is located at http://www.ibm.com/security/cryptocards.

Cryptography publications

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

- *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, Bruce Schneier, John Wiley & Sons, Inc. ISBN 0-471-12845-7 or ISBN 0-471-11709-9
- *IBM Systems Journal* Volume 30 Number 2, 1991, G321-0103
- *IBM Systems Journal* Volume 32 Number 3, 1993, G321-5521
- *USA Federal Information Processing Standard (FIPS):*
  - *Data Encryption Standard,* 46-1-1988
  - *Secure Hash Algorithm,* 180-1, May 31, 1994
  - *Cryptographic Module Security,* 140-1
- *PKCS #10: RSA Cryptography Standard*, RSA Laboratories, October 1, 1998. Obtain from http://www.rsasecurity.com/rsalabs/pkcs.
- *ISO 9796 Digital Signal Standard*
- *Internet Engineering Taskforce RFC 1321*, April 1992, MD5
- *Secure Electronic Transaction Protocol Version 1.0*, May 31, 1997

Chapter 1. Introduction to programming for the IBM Common Cryptographic Architecture

This section introduces the IBM *Common Cryptographic Architecture* (CCA) application programming interface (API). The section explains basic concepts and describes how you can obtain cryptographic and other services from the IBM 4764 and IBM 4758 Cryptographic Coprocessors and CCA.

Section topics include:
- Available CCA services
- CCA functional overview
- Security API programming fundamentals
- How the API services are organized in the remainder of the book

Available Common Cryptographic Architecture verbs

CCA products provide a variety of cryptographic processes and data-security techniques. Your application program can call verbs to perform the following functions:

- Encrypt and decrypt information, typically using the DES algorithm in the cipher block chaining (CBC) mode to enable data confidentiality
- Hash data to obtain a digest, or process the data to obtain a message authentication code (MAC), that is useful in demonstrating data integrity
- Create and validate digital signatures to demonstrate both data integrity and form the basis for non-repudiation
- Generate, encrypt, translate, and verify finance industry personal identification numbers (PINs) and transaction validation codes with a comprehensive set of finance-industry-specific services
- Manage the various DES and RSA keys necessary to perform the above operations
- Control the initialization and operation of CCA

Subsequent sections group the many available services by topic. Sections list the services in alphabetical order by name.

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

Common Cryptographic Architecture functional overview

Figure 1 on page 2 provides a conceptual framework for positioning the CCA security API which you use to access a CCA. Application programs make procedure calls to the CCA 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 receives a synchronous response; that is, your application program loses control until the access layer returns a response after processing your request.

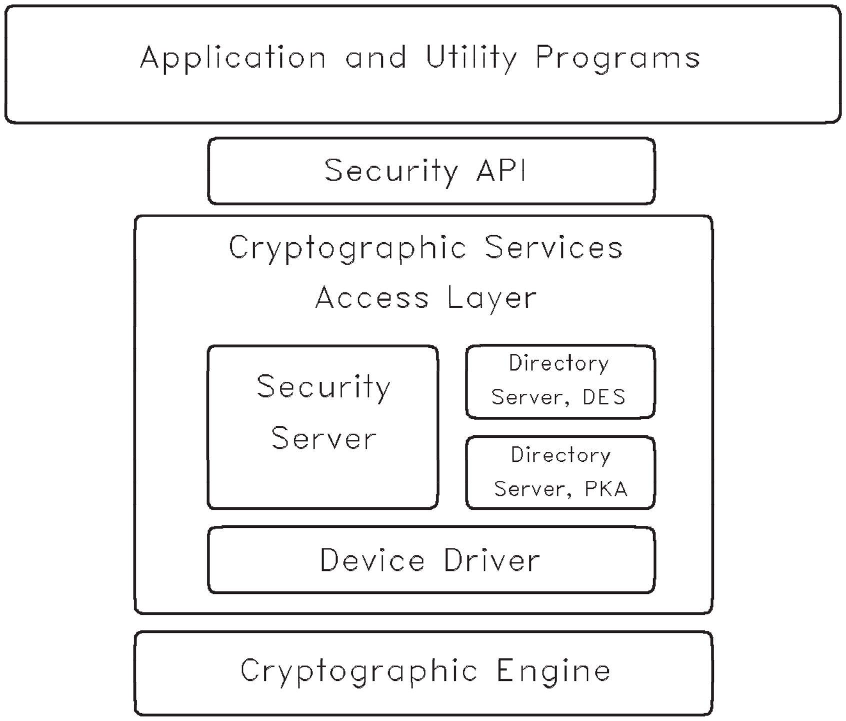


Figure 1. CCA security API, access layer, and cryptographic engine

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

CCA software support: The software consists of application development and runtime software components.

- The application development software primarily consists of language bindings that can be included in new applications to assist in accessing services available at the API. Language bindings are provided for the C programming language. The i5/OS Option 35, CCA Cryptographic Service Provider feature also provides language bindings for COBOL, RPG, and CL.

  Note: For availability of the various i5/OS code levels, see the IBM iSeries Information Center (http://www.ibm.com/eserver/iseries/infocenter).
- The runtime software can be divided into the following categories:
  - Service-requesting programs, including utility programs and application programs.
  - The Security API, an agent function that is logically part of the calling application program or utility.
  - The Cryptographic Services Access Layer: an environment-dependent request routing function, key-storage support services, and device driver to access one or more hardware cryptographic engines.
  - The cryptographic engine software that gives access to the cryptographic engine hardware.

    The cryptographic engine is implemented in the hardware of the IBM 4764 and IBM 4758 Cryptographic Coprocessors. Security-sensitive portions of CCA are implemented in the cryptographic engine software running in the coprocessor-protected environment.

– Utility programs and tools provide support for administering the CCA access-controls, administering DES and public-key cryptographic keys, and configuring the software support.

  On the IBM iSeries platform, the utility functions to set up and administer the coprocessor are accessible through a Web-based configuration utility. Refer to the Cryptographic Coprocessor section of the iSeries Information Center (http://www.ibm.com/eserver/iseries/infocenter). Also, i5/OS provides sample programs for your consideration that administer CCA access-controls, and manage DES and public-key cryptographic keys.

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

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

IBM 4764 and IBM 4758 Cryptographic Coprocessors: The coprocessors provide a secure programming and hardware environment wherein DES and RSA processes are performed. The coprocessors include a general-purpose processor, non-volatile storage, and specialized cryptographic electronics. These components are encapsulated in a protective environment to enhance security. The CCA Support Program enables applications to employ a set of DES- and RSA-based cryptographic services utilizing the coprocessor hardware. Services include:
- RSA key-pair generation
- Digital signature generation and verification
- Cryptographic key wrapping and unwrapping
- Data encryption and MAC generation/verification
- PIN processing for the financial services industry
- Other services, including DES key-management based on CCA's control-vector-enforced key separation

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

CCA access control: Each CCA node has an access-control system enforced by the hardware and protected software. This access-control system permits you to determine whether programs and persons can use the cryptographic and data-storage services. Although your computing environment might be considered open, the specialized processing environment provided by the cryptographic engine can be kept secure because 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 cannot be subverted by rogue code that might run on the main computing platform.

Coprocessor certification: After quality checking a newly manufactured coprocessor, IBM loads and certifies the embedded software. Following the loading of basic, authenticated software, the coprocessor generates an RSA key-pair and retains the private key within the cryptographic engine. The associated public key is signed by a certification key securely held at the manufacturing facility, and then the certified device key is stored within the coprocessor. The manufacturing facility key has itself been certified by a securely held key unique to the IBM 4758 and IBM 4764 Cryptographic Coprocessor product lines.

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

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

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

Establishing a CCA master key: To protect working keys, the master key must be generated and initialized in a secure manner. One method uses the internal random-number generator for the source of the master key. In this case, the master key is never external to the node as an entity (unless, out of the 2^{168} possible values, another node randomly generates the same master-key data), and no other node has the same master key 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 master-key-establishment method enables authorized users to enter multiple, separate, 168-bit key parts into the cryptographic engine. As each part is entered, that part is exclusive-ORed with the contents of the new master-key register. When all parts have been accumulated, a separate command is issued to promote the contents of the current master-key register to the old master-key register, and to promote the contents of the new master-key register to the current master-key register.

A master key can be cloned (copied) from one coprocessor node to another coprocessor node through master-key-shares distribution. This process is protected using digital certificates and authorizations. In this process, the master key can be reconstituted in one or more additional coprocessors by transporting encrypted shares of the master key. "Understanding and managing master keys" on page 26 provides additional detail about master-key management.

CCA verbs: Application and utility programs called requestors obtain service from the CCA Support Program by issuing service requests (verb calls or procedure calls) to the runtime subsystem. 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, roles, profiles: In order to ensure that only designated individuals (or programs) can run sensitive commands such as master-key loading, each command processor interrogates one or more *control-point* values within the cryptographic engine access-control system for permission to perform the request.

The access-control system includes roles. Each role defines the permissible control points for users associated with that role. The access-control system also supports user profiles that are referenced by a user ID. Each profile associates the user ID with a role, logon verification method and authentication information, and a logon session-key. Within a host process, one and only one profile, and thus role, can be logged on at a time. In the absence of a logged-on user, a default role defines the permitted commands (using the control points in the role) that a process can use.

For the coprocessor, there can be multiple logons by different users from different host processes. There can also be requests from multiple threads within a single host process.

A user is logged on and off by the Logon_Control verb. During logon, the Logon_Control verb establishes a logon session key. This key is held in user-process memory space and in the cryptographic engine. All verbs append and verify a MAC based on this key on verb control information exchanged with the cryptographic engine. Logoff causes the cryptographic engine to destroy its copy of the session key and to mark the user profile as not active.

"Using CCA access-control" on page 16 provides a further explanation of the access-control system, and "Logon_Control (CSUALCT)" on page 67 provides details about the Logon_Control verb.

How application programs obtain service

Application programs and utility programs obtain services from the security product by issuing service requests to the runtime subsystem of software and hardware. Use a procedure call according to the rules of your application language. The available services are collectively described as the security API. All of the software and hardware accessed through the security API should be considered an integrated subsystem.

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

The way application programs and utilities are linked to the API services depends on the computing environment. In the AIX and Microsoft® Windows 2000 environments, the operating systems dynamically link application security API requests to the subsystem DLL code (AIX: shared library; i5/OS: service program). In the i5/OS environment, the CCA API is implemented in a set of 64-bit entry-point service programs, one for each security API verb. Details for linking to the API on the i5/OS platform can be found by following the i5/OS link from the CCA support section of the product Web site, *http://www.ibm.com/security/cryptocards*.

Details for linking to the API are covered in the individual software product documentation. For AIX and Windows 2000, see the *IBM 4758 CCA Support Program Installation Manual*.

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

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

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

Overlapped processing

Calls to the CCA API are synchronous, that is, your program loses control until the verb completes. Multiple processing-threads can make concurrent calls to the API. CCA for Windows 2000 restricts the number of concurrent outstanding calls for a coprocessor to 32.

Note: The limitation of 32 concurrent API calls does not apply to AIX implementations.

You can maximize throughput by organizing your application or applications to make multiple, overlapping calls to the CCA API. You can also increase throughput by employing multiple coprocessors, each with CCA (see "Understanding multi-coprocessor capabilities" on page 25).

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

Host-side key caching

CCA , on other than IBM iSeries, provide caching of key records obtained from key storage within the CCA host code. However, the host cache is unique for each host process. If different host processes access the same key record, an update to a key record caused in one process does not affect the contents of the key cache held for other processes. Caching of key records within the key storage system can be suppressed so that all processes access the most current key records. The techniques used to suppress key-record caching are discussed in the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*.

Security API programming fundamentals

You obtain CCA cryptographic services from the coprocessor through procedure calls to the CCA security application programming interface (API). Most of the services that are provided are considered an implementation of the IBM Common Cryptographic Architecture (CCA). Most of the extensions that differ from other IBM CCA implementations are in the area of the access-control services. If your application program is used with other CCA products, compare the product literature for differences.

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

The security application programming interface is designed for use with high-level languages, such as C, COBOL (i5/OS), or RPG (i5/OS), 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 is described in the reference material and provides an explanation of the characteristics of the security API.

Each callable verb has an entry-point name and a fixed-length parameter list. The reference material describes each verb and includes the following information for each verb:
- Pseudonym
- Entry-point name
- Valid environments
- Description
- Restrictions
- Format
- Parameters
- Required commands

Each verb has a pseudonym (also called a general-language name) and an entry-point name (known as a computer-language name). The entry-point name is used in your program to call the verb. Each verb's seven-character, entry-point name begins with one of the following:

CSNB	Generally the DES services
CSND	RSA public-key services
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.

Supported environments: Each verb description begins with a table that describes which CCA releases use the verb. For example:

Coprocessor	AIX	i5/OS	Windows 2000
IBM CCA 4758	2.53	2.54	2.53
IBM CCA 4764		3.20	
		3.23	

The table indicates which coprocessors support the verb and which operating system platforms are valid. A CCA release number indicates that the verb is supported for that release.

Description: The verb is described in general terms. Be sure to read the parameter descriptions as these add additional detail. A **Related information** section appears at the end of the verb material for some verbs.

Restrictions: Any restrictions are noted.

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

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	In/Output	Integer	
exit_data	In/Output	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 9. 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.

Parameter 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.

Data type: Data 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, two's-complement binary number. In the i5/OS environment, integer values are presented in 4 bytes in the sequence high-order to low-order.

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

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

Array
An array of values, which can be integers or strings. Only one-dimensional arrays are permitted. For information about the parameters that use arrays, see "Rule_array and other keyword parameters" on page 11.

Data length: This is the length, in bytes, of the data identified by the parameter being described. This length can be expressed as a specific number of bytes or it might be expressed in terms of the contents of another variable.

For example, the length 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, however, is an array whose elements are 8-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.

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 storage.

entry_point_name			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	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* value. The *exit_data_length* parameter should point to a value of 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 user exit.

Restriction: You cannot use the IBM 4758 and IBM 4764 CCA Support Programs with user exits. The *exit_data_length* and *exit_data* variables must be declared in the parameter list. The *exit_data_length* parameter should be set to 0.

Return code and reason code overview: The *return code* 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. 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	Indicates normal completion; a few nonzero reason codes are associated with this return code.
4	Indicates 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 prematurely stopped processing. Generally the application programmer needs to investigate the problem and to know the associated reason code.
12	Indicates that the verb prematurely stopped processing. The reason is most likely related to a problem in the setup of the hardware or in the configuration of the software.

Value	Meaning
16	Indicates that the verb prematurely 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 might 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 a variable number of data elements that contain keywords that direct specific details of the verb process. Almost all keywords, in a rule array or otherwise, are 8 bytes in length, and should be uppercase, left-aligned, and padded with space characters. While some implementations can fold lowercase characters to uppercase, you should always code the keywords in uppercase.

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

In some cases, a rule_array is used to convey information other than keywords between your application and the server. This is, however, an exception.

Key tokens, key labels, and key identifiers

Essentially all cryptographic operations employ one or more keys. In CCA, keys are retained within a structure called a *key token*. A verb parameter can point to a variable that contains a key token. Generally you do not need to be concerned with the details of a key token and can deal with it as an entity. See “Key tokens” on page 347 for a detailed description of the key-token structures.

Keys are described as either internal, operational, or external, as follows:

Internal
A key that is encrypted for local use. The cryptographic engine decrypts an internal key to use the key in a local operation. Once a key is entered into the system it is always encrypted 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 have a flag set that indicates the key information is incomplete.

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

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

Internal key tokens can be stored in a file 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 used to locate a key record in key storage.
Key identifier	The variable must contain either a key token or a key label. The first byte in the variable 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 is processed as a key label. There are additional restrictions on the value of a key label. See “Key-label content” on page 242. The first byte in all key-token structures is in the range of X'01' to X'1F'. X'00' indicates a DES null key-token. X'FF' as the first byte of a key-related variable passed to the API raises an error condition.

API verb organization in the remainder of this document

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 the organization of the cryptographic engine and node. There are four topics:
  - Access-control administration
  - Controlling the cryptographic facility
  - Multi-coprocessor support
  - Master-key administration
  - Cryptographic key-storage initialization

  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 cannot be retrieved as an entity from the engine.

  As you examine the verb descriptions throughout this book, you see references 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 must be authorized for use. Access-control administration concerns managing these authorizations.
- Chapter 3, “RSA key-management” explains how you can generate and protect an RSA key-pair. The section 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 use a double-length DES key-encrypting key, a *transport key*. It is useful to have a general understanding of the DES key-management concepts found in section Chapter 5, “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 using a digital signature. The section 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, “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 key security. 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 section 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 mechanisms” explains how you can label, store, retrieve, and locate keys in the cryptographic-services access-layer-managed *key storage*.
- Chapter 8, “Financial services support” explains three groups of verbs of especial use in finance industry transaction processing:
  - A suite of verbs for processing personal identification numbers (PIN) in various phases of automated teller machine and point-of-sale transaction processing
  - Processing keys and information related to the *Secure Electronic Transaction (SET™)* protocol
  - Verbs to generate and verify credit-card and debit-card validation codes

Chapter 2. CCA node management and access control

This section discusses:

- The access-control system that you can use to control who can perform various sensitive operations, and at what times
- Controlling the cryptographic facility
- Understanding multi-coprocessor capabilities
- Understanding and managing master keys
- Initializing cryptographic key-storage
- Using the CCA node, access control, and master-key management verbs

Table 2 lists the verbs you use to accomplish these tasks. See "Using the CCA node, access control, and master-key management verbs" on page 34 for a detailed description of these verbs.

Table 2. CCA node, access control, and master-key management verbs

Verb	Page	Service	Entry point	Service location
Access_Control_Initialization	35	Initializes or updates access-control tables in the coprocessor.	CSUAACI	E
Access_Control_Maintenance	38	Queries or controls installed roles and user profiles.	CSUAACM	E
Cryptographic_Facility_Control	44	Reinitializes the CCA application, sets the adapter clock, resets the intrusion latch, sets the CCA 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, initiates a hardware error (for testing purposes).	CSUACFC	E
Cryptographic_Facility_Query	48	Retrieves information about the coprocessor and the state of master-key-shares distribution processing.	CSUACFQ	E
Cryptographic_Resource_Allocate	59	Connects subsequent calls to an alternative cryptographic resource (coprocessor).	CSUACRA	S
Cryptographic_Resource_Deallocate	61	Reverts subsequent calls to the default cryptographic resource (coprocessor).	CSUACRD	S
Key_Storage_Designate	63	Specifies the key-storage file used by the process.	CSUAKSD	S

Table 2. CCA node, access control, and master-key management verbs (continued)

Verb	Page	Service	Entry point	Service location
Key_Storage_Initialization	65	Initializes one of the key-storage files that can store DES or RSA (public or private) keys.	CSNBKSI	S/E
Logon_Control	67	Logs on or off the coprocessor.	CSUALCT	E
Master_Key_Distribution	70	Supports the distribution and reception of master-key shares.	CSUAMKD	E
Master_Key_Process	74	Enables the introduction of a master key into the coprocessor, the random generation of a master key, and the setting and clearing of the master-key registers.	CSNBMKP	E
Random_Number_Tests	79	Enables tests of the random-number generator and performance of the FIPS-mandated known-answer tests.	CSUARNT	E
E=Cryptographic Engine, S=Security API software				

Using 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 CCA services or commands of the coprocessor are available to a user at any given time.

Note: At the end of each CCA verb description, you find a list of commands that must be enabled to use specific capabilities of the CCA verb.

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 might be limited by parameters such as the time of day or the day of the week.

See Appendix H, "Observations on secure operations," on page 449 for more information.

Role-based access control

This CCA implementation 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 is enrolled by defining a user profile, which maps the user to one of the available roles. Profiles are described in "Understanding profiles" on page 18.

Note: For 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
A 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 to modify the access rights of those users who are already enrolled

Typically, only a few users are associated with the Key-Management Officer role, but a large population of users associated with General User role. The Access-Control Administrator role is typically limited to a single super user, because he can make any change to the access control settings. In some cases, once the system is set up, it is desirable to delete all profiles linked to Access-Control Administrator roles to prevent further changes to the access controls.

A role-based system is more efficient than one in which the authority is assigned individually for each user. In general, users can be segregated 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.

Permissions and characteristics of 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.

Required user-authentication strength level
The access-control system is designed to allow a variety of user authentication mechanisms. The one that is available for use is passphrase authentication.

All user-authentication mechanisms are given a strength rating, namely 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
The times of the day and the days of the week when the users with this role are permitted to log on. If the current time is outside the values defined for the role, logon is not allowed. You can choose values that let users log on at any time on any day of the week.

Notes:

1. Times are specified in Greenwich Mean Time (GMT).
2. If you physically move a coprocessor between time zones, remember that you must resynchronize the CCA-managed clock with the host-system clock.

Permitted commands
: A list defining which commands the user is allowed to perform in the coprocessor. Each command corresponds to one of the primitive functions that collectively comprise the CCA implementation.

Comment
: A 20-byte comment can be incorporated into the role.

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 370.

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 can operate with the capabilities and restrictions defined in the default role.

Requirement: Because 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 and defined by the customer. For simple applications, the default role by itself might 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 user profile defines a specific user to the CCA implementation. Each profile contains the following information:

User ID
: The name used to identify the user to the coprocessor. The user ID is an 8-byte value, with no restrictions on its content. Although it is typically an unterminated ASCII (or EBCDIC on i5/OS) character string, any 64-bit string is acceptable. A utility program is used to enter the user ID. That utility might restrict the ID to a specific character set.

Comment
: A 20-byte comment can be incorporated into the profile.

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

Role ID
: A character string that 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
: Values that define the first and last dates 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.

Authentication data

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 CCA implementation, user identification is accomplished by means of a passphrase supplied to the Logon_Control verb.

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, might have different strengths.

The structure of the authentication data is described in "Authentication data structure" on page 375.

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 374.

Profiles are stored in non-volatile memory inside the secure module of the coprocessor. When a user logs on, his stored profile is used to authenticate the information presented to the coprocessor. In most applications, the majority of the users operate under the default role, and do not have user profiles. Only the security officers and other special users need profiles.

Initializing and managing the access-control system

Before you can use a coprocessor with newly loaded or initialized CCA support, initialize roles, profiles, and other data. You might also need to update some of these values from time to time. Access-control initialization and management are the processes you use to accomplish this.

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

- You can use the IBM-supplied utility program for your platform:
  - Cryptographic Node Management utility program (CNM) (not for i5/OS)

    Note: The Cryptographic Node Management utility is described in the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*.
  - i5/OS Cryptographic Coprocessor Web-based configuration utility.
- You can write programs that use the access-control verbs described in this section.

Use the verbs to write programs that do more than the utility program. If your needs are simple, the utility program might do everything you need.

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

Note: See "Access_Control_Initialization (CSUAACI)" on page 35 for additional information about this verb.

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
- Getting a list of the users who are logged on to the coprocessor

Note: See "Access_Control_Maintenance (CSUAACM)" on page 38 for additional information about this verb.

See "Access-control data structures" on page 370 for additional information about data structures.

Permitting changes to the configuration

You can set up the coprocessor so no one is authorized to perform *any* functions, including further initialization. It is also possible to set up 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 set up the coprocessor with no roles having the authority to perform initialization functions.

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 configure the coprocessor so that initialization is not allowed, you can recover.

Important: To recover from this condition, all the data stored in the coprocessor, including master keys, retained keys, roles, and profiles, will be destroyed.

On an iSeries software platform, instructions on how to reinitialize the coprocessor can be found in the Troubleshooting article of the Cryptographic Coprocessor topic in the iSeries Information Center (http://www.ibm.com/eserver/iseries/infocenter). On a software platform other than iSeries, use the coprocessor load utility (CLU) and the file CRSxxyy.CLU to reset the coprocessor CCA software. This deletes all information previously loaded. Then use CLU and the file CNWxxxyy.CLU to load a new copy of the coprocessor CCA software. This restores the coprocessor's CCA function to its initial state.

Configuration and Greenwich Mean Time (GMT)

CCA 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.

All users have operating time limits, based on values in their roles and profiles. 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 4 hours during the part of the year Daylight Savings Time is in effect. At noon EDT, it is 4:00 PM GMT. At 8:00 PM EDT, 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 EDT on Tuesday, December 30, it is 11:00 PM, Tuesday, December 30 to the coprocessor. Two hours later, however, at 9:00 PM EDT, Tuesday, December 30, 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 can be obtained from your local workstation, but this information cannot 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.

Notes:

1. During the portions of the year when Daylight Savings Time is not in effect, the time difference between EST and GMT is 5 hours.
2. In the i5/OS environment, no translation is provided for Role and Profile names. The coprocessor initializes the default role name to DEFAULT encoded in ASCII. Users of the i5/OS CCA need to consider the encoding of Role and Profile names.

Logging on and logging off

A user must log on to the coprocessor in order to activate a user profile and the associated role. 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 "Logon_Control (CSUALCT)" on page 67.

When you successfully log on, the CCA implementation establishes a session between your application program and the coprocessor's access-control system. Security Application Program Interface (SAPI) stores the logon context information, which contains the session information needed by the host computer to protect and validate transactions sent to the coprocessor. As part of that session, a randomly derived session key, generated in the coprocessor, is subsequently used to protect information you interchange with the coprocessor. This protection is described in "Protecting your transaction information" on page 23. The logon process and its algorithms are described in "Passphrase verification protocol" on page 422.

With AIX and Windows 2000, the logon context information resides in memory associated with the process thread that performed the Logon_Control verb. With i5/OS, the logon context information resides in memory owned by the process in which the application runs. Host-side logon context information can be saved and shared with other threads, processes, or programs; see "Using logon context information" on page 22.

Thus, with AIX and Windows 2000, each thread in any process can log on to the CCA access control system within a specific CCA coprocessor. Because the coprocessor code creates the session key, and the session key is stored in the active context information, a thread cannot concurrently be logged on to more than one coprocessor.

In order to log on, you must prove the user's 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 meets the following criteria:

- Is not too short.
- Contains a mixture of alphabetic characters, numeric characters, and special symbols such as the asterisk (*), the plus sign (+), exclamation point (!), and others.
- Is not comprised of familiar words or other information which someone might be able to guess.

When you log on, no part of the 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 logon 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.

Using logon context information

The Logon_Control verb offers the capability to save and restore logon context information through the **GET-CNTX** and **PUT-CNTX** rule-array keywords.

The **GET-CNTX** keyword is used to retrieve a copy of your active logon context information, which you can then store for subsequent use. The **PUT-CNTX** keyword is used to make active previously stored context information. The coprocessor is unaware of what thread, program, or process has initiated a request. The host CCA supplies session information from the active context information in each request to the coprocessor. The coprocessor attempts to match this information with information it has retained for its active sessions. Unmatched session information causes the coprocessor to reject the associated request.

As an example, consider a simple application that contains two programs, LOGON and ENCRYPT:

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

LOGON

1. Log the user on to the coprocessor using the 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 is available to the ENCIPHER program. This can be as simple as a disk file, or it can be something more complicated such as shared memory or a background process.

ENCRYPT

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

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

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. Because 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.

Controlling the cryptographic facility

You can call six verbs to manage aspects of the coprocessor. One of these, the Key_Storage_Designate verb, is unique to the i5/OS implementation and allows you to select among key-storage files.

The Cryptographic_Facility_Query verb enables you to obtain the status of the CCA node. You specify one of several status categories, and the verb returns that category of status. Status information you can obtain includes:

- The condition of the master-key registers: clear, full, and so forth. The extended CCA status returns information about both the symmetric and the asymmetric master-key-register sets.
- The role name in effect for your processing thread.
- Information about the coprocessor hardware including the unique 8-byte serial number. This serial number is also printed on the label on the coprocessor's mounting bracket.
- Information about the coprocessor's operating system.
- The state of the coprocessor's battery: OK or change the battery within at most two weeks.
- Various tamper indications. This information is also returned in X'8040xxxx' status messages, for example, when you use the coprocessor load utility.

- Time and date from the coprocessor's internal clock.
- The environment ID (EID), which is a 16-byte identifier used in the PKA92 key encryption scheme and in master-key cloning. You assign an EID to represent the coprocessor.
- Diagnostic information that could be of value to product development in the event of malfunction.

The Cryptographic_Facility_Control verb enables you to:
- Reinitialize (zeroize) the CCA node. This is a two-step process that requires your application compute an intermediate value as insurance against any inadvertent reinitialization action.
- Set parameters into the CCA node, other than those related to the access-control system, including: the date and time, the function control vector used to establish the maximum strength of certain cryptographic functions, the EID, and the maximum number of master-key-cloning shares, and the minimum number of shares needed to reconstitute a master key.
- Reset the intrusion latch.

  Note: This capability is not supported on iSeries.
  The intrusion latch circuit can be set by breaking an external circuit connected to jack on the coprocessor. Normally the pins of this jack are connected to each other with a jumper. Contact IBM support for more information about possible use of the intrusion latch. Setting the intrusion latch does not cause zeroing of the coprocessor. To determine the intrusion latch state, use the **STATDIAG** rule-array option of the Cryptographic_Facility_Query verb. See "Cryptographic_Facility_Query (CSUACFQ)" on page 48 for more information.
- Reset the battery-low indicator (latch). The coprocessor electronics sets the battery-low indicator when the reserve power in the battery falls below a predetermined level as tested during power-up. You acknowledge and reset the battery-low condition using the **RESETBAT** rule-array keyword. If the battery has not been replaced, expect the low-battery-power condition to return.

The Key_Storage_Initialization verb is used to establish a fresh symmetric or asymmetric (DES or PKA) key-storage data set. The data file that holds the key records is initialized with header records that contain a verification pattern for the master key. Any existing key records in the key storage are lost if you initialize the key storage file. The index file associated with the key-record file is also initialized. The file names and paths for the key storage and its index file are obtained from different sources depending on the operating system:
- The AIX ODM registry
- The Windows registry

See the *CCA Support Program Installation Manual* for information.

Restriction: The i5/OS does not use an index file or provide master-key verification information in the key-storage file.

The Cryptographic_Resource_Allocate and Cryptographic_Resource_Deallocate verbs allow your application to steer requests to one of possibly multiple coprocessors.

Understanding multi-coprocessor capabilities

Multi-coprocessor capabilities allow you to employ more than one coprocessor, some or all of which might be loaded with the CCA application. When more than one coprocessor with CCA is installed, an application program can explicitly select which cryptographic resource (coprocessor) to use, or it can optionally employ the default coprocessor. To explicitly select a coprocessor, use the Cryptographic_Resource_Allocate verb. This verb allocates a coprocessor loaded with the CCA software. Once allocated, CCA requests are routed to it until it is deallocated. To deallocate an allocated coprocessor, use the Cryptographic_Resource_Deallocate verb. When a coprocessor is not allocated (either before an allocation occurs or after the cryptographic resource is deallocated), requests are routed to the default coprocessor.

To determine the number of CCA coprocessors installed in a machine, or in an i5/OS partition, use the Cryptographic_Facility_Query verb with the **STATCARD** rule-array keyword. The verb returns the number of coprocessors running CCA software. The count includes any coprocessors loaded with CCA user defined function (UDX) code.

When using multiple coprocessors, consider the implications of the master keys in each of the coprocessors. See “Master-key considerations with multiple CCA coprocessors” on page 31. You must also consider the implications of a logged-on session. See “Logging on and logging off” on page 21.

When you log on to a coprocessor, the coprocessor creates a session key and communicates this to the CCA, which saves the key in a session-context memory area. If your processing alternates between coprocessors, be sure to save and restore the appropriate session context information.

Multi-coprocessor CCA host implementation

Using a multi-coprocessor CCA in i5/OS host systems varies from that in the other environments. The following sections describe each approach:
- i5/OS multi-coprocessor
- AIX and Windows multi-coprocessor

i5/OS multi-coprocessor support

The kernel-level code detects all new coprocessors at IPL time and assigns them a resource name in the form of CRP01, CRP02, and so forth. In order to use a coprocessor, a user must create a cryptographic device description object. When creating the device description object, the user specifies the cryptographic resource name. The name of the device description object itself is completely arbitrary. For example, a user might call the object *BANK1CRYPTO*, or *CRP01*. The device-description-object name has no bearing on which resource it names. A user might create a device-description-object named CRP01 that internally names the CRP03 resource. (Unless you are intentionally renaming a resource, such a practice would likely lead to confusion.) With the Cryptographic_Resource_Allocate and Cryptographic_Resource_Deallocate verbs, you specify a device-description-object name (and not an i5/OS resource name). If no device has been allocated, the CCA defaults to use of the object named *CRP01*, if any. If no such object exists, the verb ends abnormally.

Note: The scope of the Cryptographic_Resource_Allocate verb and the Cryptographic_Resource_Deallocate verb is a process. All threads within the process employs the same coprocessor. Carefully consider the implications to additional threads if you employ these verbs.

AIX and Windows multi-coprocessor support

With the first call to CCA from a process, CCA associates coprocessor designators CRP01, CRP02, and so on with specific coprocessors. The host determines the total number of coprocessors installed through a call to the coprocessor device driver. The device driver designates the coprocessors using numbers 0, 1, ..., 7. The number assignment is based on the design of the BIOS in a machine. BIOS routines walk the bus to determine the type of device in each PCI slot. Adding, removing, or relocating coprocessors can alter the number associated with a specific coprocessor. The host then polls each coprocessor in turn to determine which ones contain the CCA application. As each coprocessor is evaluated, the CCA host associates the identifiers CRP01, CRP02, and so forth to the coprocessors with CCA. Coprocessors loaded with a UDX extension to CCA are also assigned a CRP0*x* identifier.

In the absence of a specific coprocessor allocation, the host code employs the device designated CRP01 by default. You can alter the default designation by explicitly setting the CSU_DEFAULT_ADAPTER environment variable. The selection of a default device occurs with the first CCA call to a coprocessor. Once selected, the default remains constant throughout the life of the thread. Changing the value of the environment variable after a thread uses a coprocessor does not affect the assignment of the default coprocessor.

If a thread with an allocated coprocessor ends without first deallocating the coprocessor, excess memory consumption results. It is not necessary to deallocate a cryptographic resource if the process itself is end; it is only suggested if individual threads end while the process continues to run.

Note: The scope of the Cryptographic_Resource_Allocate and the Cryptographic_Resource_Deallocate verbs is operating-system dependent. For AIX and Windows, these verbs are *scoped to a thread*, a process in which each of several threads within a process can allocate a specific coprocessor.

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

Understanding and managing master keys

In a CCA node, the master key is used to encrypt or 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 secured 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 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 also has an old master-key register. When a new master key is transferred to the current master-key register, the preexisting contents, if any, of the current master-key register are transferred to the old master-key register. With CCA, 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 (return code 0, reason code 10001) 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 and PKA_Key_Token_Change verbs). Whenever a working key is encrypted for local use, it is encrypted using the current master-key.

Symmetric and asymmetric master keys

CCA incorporates a second set of master-key registers. One register set is used to encrypt DES (symmetric) working-keys. The second register set is used to encrypt PKA (asymmetric) private working-keys. The verbs that operate on the master keys permit you to specify a register set (with keywords **SYM-MK** and **ASYM-MK**). If your applications that modify the master-key registers never explicitly select a register set, the master keys in the two register sets are modified in the same way and contain the same keys. However, if at any time you modify only one of the register sets, your applications thereafter needs to manage the two register sets independently.

Using cryptographic node management (CNM) utility results in operating as though there were only a single set of registers. If you use another program to modify a register in only one of the register sets, the CNM utility is no longer usable for updating the master keys.

With the IBM eServer zSeries CCA, you can use a symmetric-key master-key that has an effective double-length (usually master keys are triple length). To accomplish this, use the same key value for the first and third 8-byte portion of the key.

Establishing master keys

Master keys are established in one of three ways:

- From clear key parts (components)
- Through random generation internal to the coprocessor
- Copying encrypted master-key shares, called *cloning*

Establishing a master key from clear information

Individual key parts are supplied as clear information, and 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 installing the master key, individuals trusted to not share the key-part information retrieve the parts and enter the information into the cryptographic engine. Use the Master_Key_Process verb for 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. This requires that roles exist that enforce this separation of responsibility.

Setting the master key uses a unique command with its own control point. You can set up the access-control system to require the participation of at least three individuals or three groups of individuals.

You can check the contents of any of the master-key registers, and the key parts as they are entered into the new master-key register, using the Key_Test verb. The verb performs a one-way function on the key-of-interest, the result of which is either returned or compared to a known correct result.

Establishing a master key from an internally generated random value

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. Randomly generating 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 reestablish 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 exposing the value of the master key. Do this by cloning the master key, splitting the master key into *n* shares, of which *m* shares, $1 \le m \le n \le 15$, 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.

For this secure master-key cloning process, use the CNM utility. See Section 5 and Appendix F of the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*. The utility can hold the certificates and shares in a database that you can transport on diskette between the various nodes:
- The certifying node public-key certificate
- The coprocessor master-key share-source node public-key certificate
- The coprocessor master-key share-receiving node public-key certificate
- The master-key shares

You establish the *m* and *n* values using the Cryptographic_Facility_Control verb.

Shares of the current master-key are obtained using the Obtain mode of the Master_Key_Distribution verb. The Receive mode of the Master_Key_Distribution verb is used to enter an individual share into the receiving (target) cryptographic engine. When sufficient shares have been entered, the verb returns status (return code 4, reason code 1024) that indicates the cloned master-key is now complete within the new master-key register of the target cryptographic 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. A certified public-key from the target cryptographic-engine is validated, and the share-encrypting key is wrapped using the public key from the certificate.

At the target cryptographic-engine, an encrypted share and the 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.

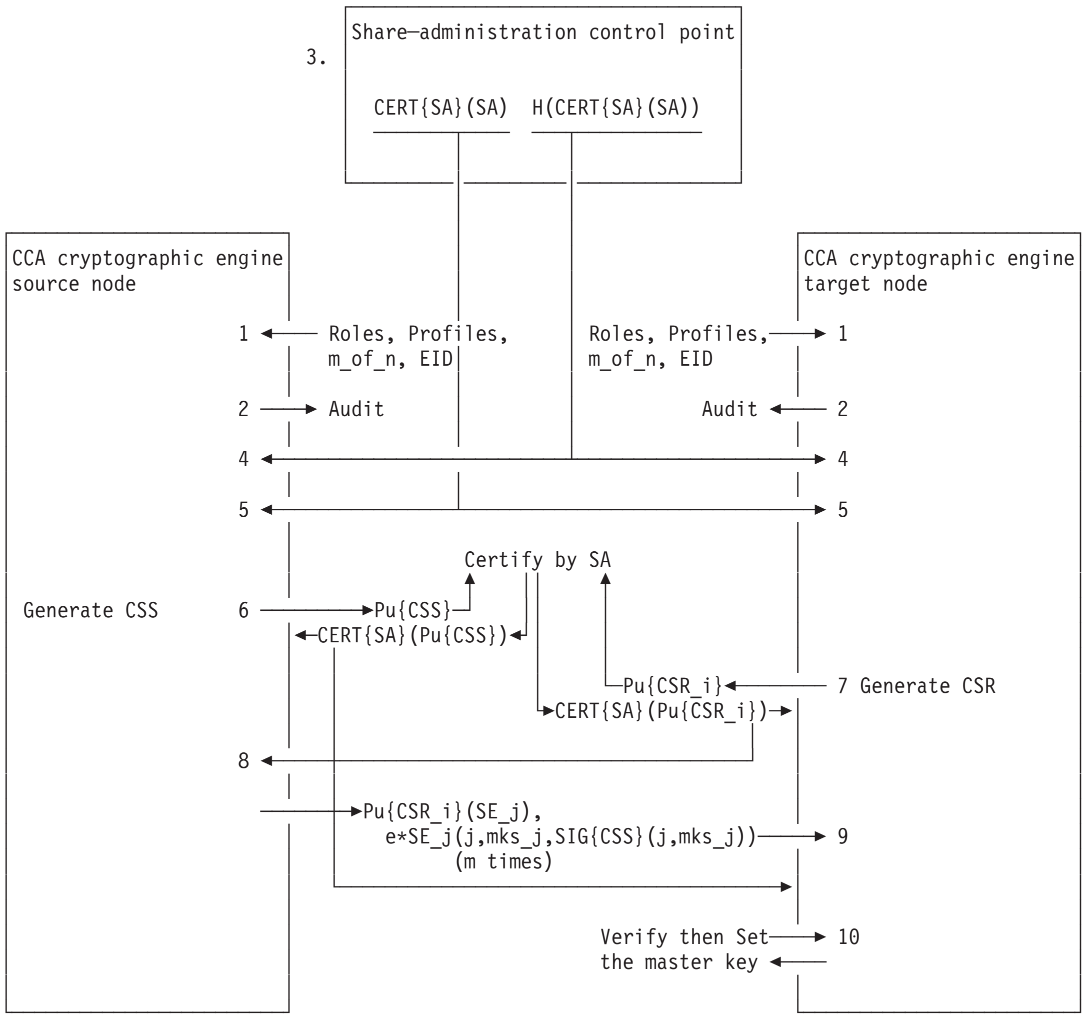
The certificates used to validate the share-signing public key and the target-engine public key used to wrap the share-encrypting key are validated by the cryptographic engines using a retained public key. A retained public key is introduced into a cryptographic engine in a two-part process using the PKA_Public_Key_Hash_Register and PKA_Public_Key_Register verbs. This allows you to establish two distinct roles to enforce dual control. 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 using 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.

The certifying node can be any of the following:
- Share source node
- Share target node
- Independent node that might be located in a cryptographic control center

You must initialize the target coprocessor with its retained private key and have 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.



Share–administration control point
3.
CERT{SA}(SA)
H(CERT{SA}(SA))
CCA cryptographic engine source node
CCA cryptographic engine target node
1
Roles, Profiles, m_of_n, EID
Roles, Profiles, m_of_n, EID
1
2
Audit
Audit
2
4
4
5
5
Certify by SA
Generate CSS
6
Pu{CSS}
CERT{SA}(Pu{CSS})
Pu{CSR_i}
7 Generate CSR
CERT{SA}(Pu{CSR_i})
8
Pu{CSR_i}(SE_j),
e*SE_j(j,mks_j,SIG{CSS}(j,mks_j))
(m times)
9
Verify then Set the master key
10

Figure 2. Coprocessor-to-coprocessor master-key cloning

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

1. Install appropriate access-control roles and profiles, *m*-of-*n*, and EID values. Have operators change their profile passwords. Ensure that the roles provide the degree of responsibility-separation that you require.
2. Audit the setup of the share administration, share source, and share receiving nodes.
3. Generate a retained RSA private key, the Share-Administration (SA) key. This key is used to certify the public keys used in the scheme. Self-certify the SA key. Distribute the hash of this certificate to the source and share-receiving nodes under dual control.
4. Install (register) the hash of the SA public-key in both the source and receiving nodes. Two different roles can be used to permit this and the next step to aid in ensuring dual control of the cloning process.
5. Install (register) the SA public-key in both the source and receiving nodes.
6. In the source node, generate a retained key usable for master-key administration, the coprocessor share signing (CSS) key, and have this key certified by the SA key.

7. In the target node, generate a retained key usable for master-key administration, the coprocessor share receiving (CSR) key, and have this key certified by the SA key.
8. Once a master key has been established in the source node, for example through random master-key generation, obtain shares of the master key. Also obtain master-key verification information for use in step 10 using the Key_Test verb. Generally, fewer shares are required to reconstitute the master key than that which can be obtained from the source node.
9. Deliver and install the master-key shares.
10. Use the Key_Test verb to verify that the new master-key in the target node has the proper value. Then set the master key.

Master-key considerations with multiple CCA coprocessors

Master keys are used to wrap working keys (as opposed to clear keys or keys wrapped by key-encrypting keys or RSA keys). Master-key-wrapped keys are either stored in the CCA key storage, or are held and managed by your application or applications. When multiple coprocessors are installed, it is a responsibility of the using organization to ensure that appropriate current and old master-keys, both symmetric and asymmetric, are installed in the multiple coprocessors. The most straightforward approach is to ensure that when you change master keys on one CCA coprocessor, you also change the master keys (both asymmetric and symmetric) on the other coprocessor.

The approach to multiple coprocessors differs in detail between i5/OS and the workstation environments. Each type of environment is discussed:
- i5/OS
- AIX and Windows

i5/OS multi-coprocessor master-key support

Load all CCA coprocessors with the same current and the same old master-keys, especially if your applications perform load balancing among the coprocessors or if the coprocessors are used for SSL.

With i5/OS, multiple key-storage files can exist. To avoid confusion, keep all keys in the key-storage files encrypted by a common, current master-key. The master-key verification pattern is not stored in the header record of any key-storage file. Therefore, it is important that when you change the master key, you reencipher all of the keys in all of your key-storage files. The organization that manages all users of the coprocessors must arrange procedures for keeping all key-storage files up to date with the applicable current master-key. The person changing the master key might not have authorization to, or knowledge of, all key-storage files on the system.

The order for loading and setting of the master key between coprocessors is not significant. However, be sure that after all coprocessor master keys have been updated that you then update all key-storage files. Remember that if you import a key or generate a key, it is returned encrypted by the current master key within the coprocessor used for the task.

AIX and Windows multi-coprocessor master-key support

All of the CCA coprocessors within the system should use the same current and old master keys. When setting a new master-key, it is essential that all of the changes are performed by a single program running on a single thread. If the thread-process is ended before all of the coprocessor master-keys are changed, significant complications can arise. It is suggested that you start the CNM utility and use it to make all of the changes before you end the utility.

If you fail to change all of the master keys with the same program running on the same thread, either because there is an unplanned interruption, or perhaps because you intend to have different master keys between coprocessors, you need to understand the design of the CCA host code that is described next.

CCA host code design (AIX and Windows)

CCA keeps a copy of the symmetric or the asymmetric current master-key verification pattern in the key-storage header records. This information is used to ensure that a given key-storage file is associated with a coprocessor having the same current master-key. This can prevent accessing an out-of-date key-storage backup file. The verification pattern is written into the header record when key storage is initialized, and when the current master-key is changed in a coprocessor.

CCA keeps two flags in memory associated with a host-processing thread. If there are multiple threads, each thread has its own set of flags. The flags, symmetric-directory-open (SDO) and asymmetric-directory-open (ADO), are set to false when CCA processing begins on the thread.

When a CCA verb is called and a key storage is referenced, and if the associated flag (SDO or ADO) is false, CCA obtains the verification pattern for the current master-key and compares this to the header-record information. If the patterns match, the flag is set to true, and processing continues. If the existing patterns do not match, processing ends with an error indication. If there is no current master key or if key storage has not been initialized, processing continues although, depending on the CCA verb, other error conditions might arise.

A key-storage reference occurs in two cases:
- When the verb call employs a key label
- When the **SET** master-key option is used on the Master_Key_Process verb

Situations to consider

When you employ multiple coprocessors with CCA, consider the following cases in regard to master keys. Remember that if you explicitly manage the symmetric or the asymmetric master keys (using the **SYM-MK** or **ASYM-MK** keywords on the Master_Key_Process verb), you have both master keys and both key storages to consider. If you do not explicitly manage the two classes of master keys, then CCA operates as though there is a single set of master keys. The CNM utility provided with the CCA program does not explicitly manage the two sets of keys, and the program design assumes that the master keys have always been managed without explicit reference to the symmetric or the asymmetric keys.

Setting master keys in multiple coprocessors

If you keep the master keys the same in all of the CCA coprocessors, and you set the master key in each of the coprocessors from a single program running on the same thread, the following steps take place:

- When all of the coprocessors are newly initialized, that is, their current master-key registers are empty, install the same master key in each of the new-master-key registers. Then set the master key in each of the coprocessors. Finally, if you are going to use key storage, initialize key storage.
- If all of the coprocessors have the same current master-key, when you undertake to set the master key in the first coprocessor, the code attempts to set the directory-open flags (SDO and ADO). This succeeds if

you have the proper key-storage files (or key storage is not initialized). The verification pattern in the key-storage header is changed as soon as the first master key is set.

When you set the master key in the additional coprocessors, because the directory-open flags are already set, no check is made to ensure that the verification patterns in key storage and for the current-master-key match (and they would not match because the header was updated when the first coprocessor master-key was set). As soon as the master key is set, its verification pattern is copied to the header in key storage.

The key in the new-master-key register is not verified. You might want to confirm the proper and consistent contents of these registers using the key-test service prior to undertaking setting of the master keys.

Setting the master key in a coprocessor after other coprocessors are successfully in operation

If you have one or more coprocessors in operation, and then wish to do the following:

- Add an additional coprocessor
- Set its current, and possibly old, master keys to the keys already in the other coprocessors

Consider the following cases:

1. If the new coprocessor has a current master key that is not the same as that in the other coprocessors, and if key storage is initialized for use with the other coprocessors, when you start a new thread and attempt to set the master key, the action fails unless you take precautions. Because the directory-open flags are initially set to false, the CCA host code compares the verification pattern for the current master-key in the coprocessor and in the key-storage header record. This comparison fails and processing ends with an error indication.
2. If the new coprocessor did not have a key in the current master-key register, the set-master-key operation proceeds. The verification pattern for this master key are copied to an initialized key-storage header record.

A solution to the first situation is to proceed as follows:

- Allocate a coprocessor that has the desired current master key or keys
- Perform a DES_Key_Record_List or other action that causes the key-storage-valid flags to be set.
- Deallocate the coprocessor
- Allocate the new coprocessor
- Set the master key

You might need to install two master keys into the new coprocessor in order have both the current and the old master keys agree with those in the other coprocessors.

Intentionally using different master keys in a set of coprocessors

This situation becomes very complicated if you are using key storage with a subset of the coprocessors. If you are not using key storage and have not initialized key storage files, just load and set the master keys as you would in a single-coprocessor situation.

While you are changing master keys in a multiple-coprocessor arrangement, it might be undesirable to continue other cryptographic processing. Consider the following possibilities:

- Keys generated or imported and returned enciphered with the latest master key are not usable with other coprocessors until they too have been updated with the latest master key. Existing keys might still be usable because the previous master key in the updated coprocessors are in the old master-key register and CCA can use this to recover the working keys.
- The header record in the key-storage file might have been altered to an undesirable value–refer to the earlier discussion.
- If you set the master key without specifically mentioning symmetric or asymmetric keys, and if you are using key storage, you need to have both the symmetric and the asymmetric key storage files initialized, even if you do not place keys in one or both of the key storages files.

Initializing cryptographic key-storage

Key storage is a repository of keys that you access by key label, using labels that you or the applications define. DES keys and PKA or RSA keys are held in separate storage systems. The coprocessor has limited internal storage for RSA keys.

Note: Keys stored in the coprocessor are not considered part of key storage.

| Use the Key_Storage_Initialization (CSNBKSI) verb to initialize a DES or RSA (public or private) key-storage file. Before using the Key_Storage_Initialization verb, you must establish the master keys.

Using the CCA node, access control, and master-key management verbs

Access_Control_Initialization (CSUAACI)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Access_Control_Initialization verb is used to initialize or update parameters and tables for the access-control system in the IBM 4758 and IBM 4764 Cryptographic Coprocessors.

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 function 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	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1, 2, or 3
rule_array	Input	String array	rule_array_count * 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

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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 1, 2, or 3 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

The rule_array keywords are shown in the following table:

Keyword	Meaning
Function to perform (one required)	
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 changes 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 failure count permits a user to log on again, after being locked out due to too many failed consecutive attempts.
Options (one or two, optional)	
PROTECTD	Specifies to operate in *protected* mode. Data sent to the coprocessor is protected by encrypting the data with the user's session key, K_S. If the user has not successfully logged on, there is no session key in effect, and the **PROTECTD** keyword results in a not-logged-on error.
REPLACE	Specifies that a new profile can replace an existing 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 profile which already exists are 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 coprocessor.

verb_data_1_length

The *verb_data_1_length* parameter is a pointer to an integer variable containing the number of bytes of data in 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 variable is used differently depending on the function being performed.

Rule-array keyword	Contents of *verb_data_1* variable
INIT-AC	The variable contains a list of zero or more user profiles to be loaded into the coprocessor. See "Profile structure" on page 374.
CHGEXPDT, **CHG-AD**, or **RESET-FC**	The variable contains the eight-character profile ID for the user profile that is to be modified.

verb_data_length_2

The *verb_data_length_2* parameter is a pointer to an integer variable containing the number of bytes of data in the verb_data_2 variable.

verb_data_2

The verb_data_2 parameter is a pointer to a string variable containing data used by the verb. Authentication data structures are described in "Access-control data structures" on page 370.

The manner in which this variable is used depends on the function being performed.

Rule-array keyword	Contents of *verb_data_2* variable
INIT-AC	The variable contains a list of zero or more roles to be loaded into the coprocessor. See "Role structure" on page 370.
CHGEXPDT	The field contains the new expiration date to be stored in the specified user profile. The expiration date is an 8-character string, in the form YYYYMMDD.
CHG-AD	The field contains the new authentication-data, to be used in the specified user profile. 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 in the active role:

- Initialize Access-Control System (offset X'0112') with the **INIT-AC** keyword. See "Profile structure" on page 374.
- Change User Profile Expiration Date (offset X'0113') with the **CHGEXPDT** keyword.
- Change User Profile Authentication Data (offset X'0114') with the **CHG-AD** keyword.
- Reset User Profile Logon-Attempt-Failure Count (offset X'0115') with the **RESET-FC** keyword.

Access_Control_Maintenance (CSUAACM)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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:
- Retrieve 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 coprocessor
- Retrieve 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	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
name	Input	String	8 bytes
output_data_length	In/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 9.

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

rule_array
: The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are shown in the following table:

Keyword	Meaning
Function to perform (one required)	
LSTPROFS	Retrieves a list of the user profiles installed in the coprocessor. Keyword **Q-NUM-RP** shows how to determine how much data this request returns to the application program.
LSTROLES	Retrieves a list of the roles installed in the coprocessor. Keyword **Q-NUM-RP** shows how to determine how much data this request returns to the application program.
GET-PROF	Retrieves the non-secret part of a specified user profile.
GET-ROLE	Retrieves 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 is returned with the **LSTROLES** or **LSTPROFS** keywords.
Q-NUM-UR	Queries the number of users logged on to the coprocessor. This allows the application program to know how much data is returned with the **LSTUSERS** keyword. Users can log on or log off between the time you use **Q-NUM-UR** and the time you use **LSTUSERS**, so the list of users might not always contain exactly the number the coprocessor reported was logged on.
LSTUSERS	Retrieves a list of the profile IDs for all users who are logged on to the coprocessor.

name

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

The manner in which this variable is used depends on the function being performed.

Rule-array keyword	Contents of *name* variable
LSTPROFS, **LSTROLES**, **Q-NUM-RP**, **Q-NUM-UR**, or **LSTUSERS**	The *name* variable is unused.
GET-PROF or **DEL-PROF**	The *name* variable contains the 8-character profile ID for the user profile that is to be retrieved or deleted.

Rule-array keyword	Contents of *name* variable
GET-ROLE or **DEL-ROLE**	The *name* variable contains the 8-character role ID for the role definition 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* variable. The value must be a multiple of four bytes.

On input, the *output_data_length* variable must be set to the total size of the variable pointed to by the *output_data* parameter. On output, this variable contains the number of bytes of data returned by the verb in the *output_data* variable.

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* variable is in big-endian format; the high-order byte of the value is in the lowest-numbered address in storage. Authentication data structures are described in "Access-control data structures" on page 370.

This manner in which this variable is used depends on the function being performed.

Rule-array keyword	Contents of *output_data* variable
LSTPROFS	Contains a list of the profile IDs for all the user profiles stored in the coprocessor.
LSTROLES	Contains a list of the role IDs for all the roles stored in the coprocessor.

Rule-array keyword	Contents of *output_data* variable
GET-PROF	The variable 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 1-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 variable padded on the right with spaces, which describes the profile. This variable is not X'00' terminated. **Role** The 8-character name of the user's assigned role. **Logon failure count** A 1-byte integer containing the number of consecutive failed logon attempts by the user. **Pad** A 1-byte padding value containing X'00'. **Activation date** The first date on which the profile is valid. The date consists of a 2-byte integer containing the year, followed respectively by a 1-byte integer for the month and a 1-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 mechanisms information** For each authentication mechanism associated with the profile, the verb returns a series of three integer values: • The 2-byte Mechanism ID • The 2-byte Mechanism Strength • The 4-byte authentication data Expiration date, which has the same format as the Activation date described above The authentication data itself is not returned; only the IDs, strength, and expiration date of the data are returned.

Rule-array keyword	Contents of *output_data* variable
GET-ROLE	The variable contains the non-secret portion of the selected role. This includes the following data, in the order listed. **Role version** Two bytes containing 2 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 variable padded on the right with spaces, containing a comment which describes the role. This variable is not X'00' terminated. **Required authentication-strength level** A 2-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 1-byte integer values, a 1-byte hour, followed by a 1-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 1-byte variable 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 is set to 0. **Access-control-point list** The access-control-point bit map defines which functions a user with this role is permitted to run.
DEL-PROF or **DEL-ROLE**	The variable is empty. Its length is 0.
Q-NUM-RP	The variable contains an array of two 4-byte integers. The first integer is the number of roles loaded using the Access_Control_Initialization verb, while the second integer is the number of user profiles loaded with use of the same verb.
Q-NUM-UR	The variable contains a single integer value which indicates the number of users logged on to the coprocessor.
LSTUSERS	The variable contains an array of 8-character profile IDs, one for each user logged on to the coprocessor. The list is not in any specific order.

Required commands

The Access_Control_Maintenance verb requires the following commands to be enabled in the active role:

- Read Public Access-Control Information (offset X'0116') with the **LSTPROFS**, **LSTROLES**, **GET-PROF**, **GET-ROLE**, and **Q-NUM-RP** keywords.
- Delete User Profile (offset X'0117') with the **DEL-PROF** keyword.
- Delete Role (offset X'0118') with the **DEL-ROLE** keyword.

Cryptographic_Facility_Control (CSUACFC)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

Use the Cryptographic_Facility_Control verb to perform the following services:

- Reinitialize the CCA application of the coprocessor.
- Set the date and time in the coprocessor clock.
- Reset the coprocessor Intrusion Latch (see page 24).
- Reset the coprocessor Battery-Low Indicator (see page 24).
- Load or clear the function control vector, which defines limitations on the cryptographic functions available in the coprocessor.
- Establish the EID, which is a user-defined identifier. Once set, the EID can only be set again following a CCA reinitialization.
- 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.
- Force a coprocessor hardware error for testing purposes.

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

- Use only these characters in an EID: A...Z, a...z, 0...9, and these additional characters relating to different character symbols in the various national language character sets as listed in the following table:

ASCII systems	**EBCDIC systems**	**USA graphic (for reference)**
X'20'	X'40'	space character
X'26'	X'50'	&
X'3D'	X'7E'	=
X'40'	X'7C'	@

  The alphabetic and numeric characters should be encoded in the normal character set for the computing platform that is in use, either ASCII or EBCDIC.
- Forcing a hardware error using the **ERRINJ1** keyword is possible starting with the 3.20 release. This capability is not available on iSeries.

Format

Cryptographic_Facility_Control			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1 or 2
rule_array	Input	String array	rule_array_count * 8 bytes
verb_data_length	In/Output	Integer	
verb_data	In/Output	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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

This verb accepts two keywords in the rule array. One specifies the coprocessor for which the request is intended, the other specifies the function to perform. No rule-array elements are set by the verb. The rule_array keywords are shown in Table 3.

Table 3. Cryptographic_Facility_Control rule_array keywords

Keyword	Meaning
Coprocessor to use (optional)	
ADAPTER1	This keyword is ignored. It is accepted for backward compatibility.
Control function to perform (one required)	
RQ-TOKEN	Requests a random 8-byte token from the adapter, which is returned in the *verb-data* variable. This is the first step when reinitializing the coprocessor. The second step for reinitialization uses **RQ-REINT**.
RQ-REINT	Reinitializes the CCA application in the coprocessor. For **RQ-REINT**, you must set the *verb_data* variable to the one’s complement of the token that was returned by the coprocessor when you ran 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 coprocessor.
SETCLOCK	Sets the date and time of the coprocessor’s secure clock. You must put the date and time values in the *verb-data* variable.
RESET-IL	Clears the intrusion latch on the coprocessor.
RESETBAT	Clears the battery-low indicator (latch) on the coprocessor.
LOAD-FCV	Loads a new function control vector into the coprocessor.
CLR-FCV	Clears the function control vector from the coprocessor.
SET-EID	Sets an EID value.

Table 3. Cryptographic_Facility_Control rule_array keywords (continued)

Keyword	Meaning
SET-MOFN	Sets 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.
ERRINJ1	Causes a coprocessor hardware error. This capability can be used for testing purposes. Using this keyword results in return code 16, reason code 336. **Note:** See Restrictions

verb_data_length

The *verb_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *verb-data* variable. On input, specify the size of the variable. The verb updates the variable with the size of the returned data.

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 manner in which this variable is used differs depending on the value of the control function selected by a rule-array keyword.

- For **RESETBAT**, **RESET-IL**, and **ERRINJ1**, the *verb_data* variable is unused. Set the *verb-data-length* variable to zero.
- For **RQ-TOKEN**, *verb_data* is an output parameter. It receives an 8-byte randomly generated value, which the application uses with the **RQ-REINT** keyword on a subsequent call.

  On input, the *verb-data-length* variable must contain the length of the buffer addressed by the verb_data pointer. Allocate an 8-byte buffer and specify this length in the *verb-data-length* variable.
- 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. Allocate an 8-byte buffer and specify this length in the *verb-data-length* variable.
- For **SETCLOCK**, verb-data is an input variable. It must contain a character string which contains the current GMT date and time. Allocate a 16-byte buffer and specify this length in the *verb-data-length* variable. This string has the form YYYYMMDDHHmmSSWW, where these fields are defined as follows.

YYYY	The current year
MM	The current month, from 01 – 12
DD	The current day of the month, from 01 – 31
HH	The current hour of the day, from 00 – 23
mm	The current minutes past the hour, from 00 – 59
SS	The current seconds past the minute, from 00 – 59
WW	The current day of the week, where Sunday is represented as 01, and Saturday by 07
- For **LOAD-FCV**, verb data is an input variable. It must contain a character string which contains the function control vector (FCV) as described in "Function control vector" on page 383. Allocate a 204-byte buffer and specify this length in the *verb-data-length* variable.

- For **CLR-FCV**, no data is provided and the *verb-data-length* variable should be set to 0.
- For **SET-EID**, verb-data is an input variable. The variable contains a 16-byte EID value. This identifier is used in verbs such as PKA_Key_Generate and PKA_Symmetric_Key_Import. See “Restrictions” on page 44 for a list of valid characters in an EID. Allocate a 16-byte buffer and specify this length in the *verb-data-length* variable.
- 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. The *verb-data* variable contains a 2-element array of integers. The first element is the *m* minimum 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). Allocate an 8-byte buffer (two 4-byte integers) and specify this length in the *verb-data-length* variable.

Required commands

The Cryptographic_Facility_Control verb requires the following commands to be enabled in the active role:

- Reinitialize Device (offset X'0111') with the **RQ-TOKEN** or **RQ-REINT** keywords
- Set Clock (offset X'0110') with the **SETCLOCK** keyword
- Reset Intrusion Latch (offset X'010F') with the **RESET-IL** keyword
- Reset Battery-Low Indicator (offset X'030B') with the **RESETBAT** keyword
- Load Function-Control Vector (offset X'0119') with the **LOAD-FCV** keyword
- Clear Function-Control Vector (offset X'011A') with the **CLR-FCV** keyword
- Set EID (offset X'011C') with the **SET-EID** keyword
- Initialize Master Key Cloning (offset X'011D') with the **SET-MOFN** keyword
- Error Injection 1 (offset X'0304') with the **ERRINJ1** keyword

Cryptographic_Facility_Query (CSUACFQ)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The Cryptographic_Facility_Query verb is used to retrieve information about the coprocessor and the CCA application program in that coprocessor. This information includes the following:
- General information about the coprocessor, its operating system, and CCA application
- Status of master-key shares distribution
- EID
- Diagnostic information from the coprocessor
- Export-control information from the coprocessor
- Time and date information from the coprocessor

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

The verb returns information elements in the rule array and sets the *rule-array-count* variable to the number of returned elements.

Restrictions

- You cannot limit the number of returned rule-array elements. Table 4 on page 50 describes the number and meaning of the information in output rule-array elements.

  Tip: Allocate a minimum of 30 rule-array elements to allow for extensions of the returned information.
- Obtaining diagnostic information using the **BDGREGS1** keyword is possible starting with release 3.20 release. This capability is for use by the IBM Support Center. The returned information is non-sensitive and not publicly defined.

Format

CSUACFQ

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length
rule_array_count	In/Output	Integer	1 or 2 on input
rule_array	In/Output	String array	rule_array_count * 8 bytes
verb_data_length	In/Output	Integer	
verb_data	In/Output	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 9.

rule_array_count

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

On output, the verb sets the variable to the number of rule-array elements it returns to the application program.

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

On input, set the rule array to specify the type of information to retrieve. There are two input rule-array elements, as described in the following table.

Keyword	Meaning
Adapter to use (optional)	
ADAPTER1	This keyword is ignored. It is accepted for backward compatibility.
Information to return (one required)	
BDGREGS1	Obtains non-sensitive crypto-chip information. This option is for diagnostic purposes. The returned information is not publicly defined. See Restrictions.
STATCCA	Obtains CCA-related status information.
STATCCAE	Obtains CCA-related extended status information.
STATCARD	Obtains coprocessor-related basic status information.
STATDIAG	Obtains diagnostic information.
STATEID	Obtains the EID.
STATEXPT	Obtains function control vector-related status information.
STATMOFN	Obtains master-key shares distribution information.
TIMEDATE	Reads the current date, time, 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 **BDGREGS1**, **STATCCA**, **STATCCAE**, **STATCARD**, **STATDIAG**, **STATEID**, **STATEXPT**, **STATMOFN**, or **TIMEDATE**.

For rule-array elements that contain numbers, those numbers are represented by numeric characters which are left-aligned and padded on the right with space characters. For example, a rule-array element which contains the number 2 contains the character string **“2   ”** (the number 2 followed by 7 space characters).

On output, the rule-array elements can have the values shown in Table 4.

Table 4. Cryptographic_Facility_Query information returned in the rule array

Element number	**Name**	**Description**
Output rule-array for option DBGREGS1		
		Twelve rule-array elements are filled with non-sensitive information from the coprocessor's crypto-chip control registers. This information is for debugging purposes and is not publicly disclosed. See Restrictions.
Output rule-array for option STATCCA		
1	NMK status	The state of the new master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a partially complete key. • A value of 3 means the register contains a complete key.
2	CMK status	The state of the current master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a key.
3	OMK status	The state of the old master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a key.
4	CCA application version	A character string that identifies the version of the CCA application program that is running in the coprocessor.
5 (CCA Release 2.x)	CCA application build date	A character string containing the build date for the CCA application program that is running in the coprocessor.
6 (CCA Release 2.x)	User role	A character string containing the role identifier which defines the host application user's current authority.
5 (CCA Release 3.x)	Coprocessor operating system name	The word *Linux*®, left-aligned and padded with three spaces on the right.

Table 4. Cryptographic_Facility_Query information returned in the rule array (continued)

Element number	Name	Description
6 (CCA Release 3.x)	Coprocessor operating system version	Left-aligned release, version, and modification level, separated by periods. If inclusion of the modification level and preceding period exceeds eight characters, the modification level and period are truncated.
Output rule-array for option STATCCAE		
1	Symmetric NMK status	The state of the symmetric new master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a partially complete key. • A value of 3 means the register contains a complete key.
2	Symmetric CMK status	The state of the symmetric current master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a key.
3	Symmetric OMK status	The state of the symmetric old master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a key.
4	CCA application version	A character string that identifies the version of the CCA application program that is running in the coprocessor.
5	CCA application build date	A character string containing the build date for the CCA application program that is running in the coprocessor.
6	User role	A character string containing the Role identifier which defines the host application user's current authority.
7	Asymmetric NMK status	The state of the asymmetric new master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a partially complete key. • A value of 3 means the register contains a complete key.

Table 4. Cryptographic_Facility_Query information returned in the rule array (continued)

Element number	Name	Description
8	Asymmetric CMK status	The state of the asymmetric current master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a key.
9	Asymmetric OMK status	The state of the asymmetric old master-key register: • A value of 1 means the register is clear. • A value of 2 means the register contains a key.
Output rule-array for option STATCARD		
1	Number of installed adapters	A numeric character string containing the number of active coprocessors installed in the machine. This only includes coprocessors that have CCA software loaded (including those with CCA UDX software). Non-CCA coprocessors are not included in this number.
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	Coprocessor operating system name	A character string identifying the operating system firmware on the coprocessor.
6	Coprocessor operating system version	A character string identifying the version of the coprocessor's operating system firmware.
7	Coprocessor part number	A character string containing the 8-character part number identifying the version of the coprocessor.

Table 4. Cryptographic_Facility_Query information returned in the rule array (continued)

Element number	**Name**	**Description**
8	Coprocessor EC level	A character string containing the 8-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 numeric character string containing the operating speed of the microprocessor chip, in megahertz.
11	Adapter ID (see also element number 15)	A unique identifier manufactured into the coprocessor. The coprocessor adapter ID is an 8-byte **binary** value where the high-order byte is X'78' for an IBM 4758-001 and 4758-013, and is X'71' for an IBM 4758-002 and 4758-023. The remaining bytes are a random value.
12	Flash memory size	A numeric character string containing the size of the flash EPROM memory on the coprocessor, in 64-kilobyte increments.
13	DRAM memory size	A numeric character string containing the size of the dynamic RAM (DRAM) memory on the coprocessor, in kilobytes.
14	Battery-backed memory size	A numeric character string containing the size of the battery-backed RAM on the coprocessor, in kilobytes.
15	Serial number	A character string containing the unique serial number of the coprocessor. The serial number is factory installed and is also reported by the CLU utility in a coprocessor-signed status message.
Output rule-array for option STATDIAG		

Table 4. Cryptographic_Facility_Query information returned in the rule array (continued)

Element number	**Name**	**Description**
1	Battery state	A numeric character string containing a value which indicates whether the battery on the coprocessor needs to be replaced: • A value of 1 means that the battery is good. • A value of 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: • A value of 1 means that the latch is cleared. • A value of 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: • A value of 1 means that the error log is empty. • A value of 2 means that the error log contains abnormal termination data, but is not yet full. • A value of 3 means that the error log is full, and cannot hold any more 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: • A value of 1 means no intrusion has been detected. • A value of 2 means an intrusion attempt has been detected.

Table 4. Cryptographic_Facility_Query information returned in the rule array (continued)

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 might indicate an attempt to attack the security module: • A value of 1 means only acceptable voltages have been detected. • A value of 2 means a voltage has been detected below the low-voltage tamper threshold.
6	High voltage detected	A numeric character string containing a value indicates whether a power-supply voltage was above the maximum acceptable level. This might indicate an attempt to attack the security module: • A value of 1 means only acceptable voltages have been detected. • A value of 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 of the acceptable limits. This might indicate an attempt to attack the security module: • A value of 1 means the temperature is acceptable. • A value of 2 means the temperature has been detected outside of an acceptable limit.
8	Radiation detected	A numeric character string containing a value to indicate whether radiation was detected inside the secure module. This might indicate an attempt to attack the security module: • A value of 1 means no radiation has been detected. • A value of 2 means radiation has been detected.

Table 4. Cryptographic_Facility_Query information returned in the rule array *(continued)*

Element number	Name	Description
9, 11, 13, 15, 17	Last 5 commands run	These five rule-array elements contain the last five commands that were run by the coprocessor CCA application. They are in chronological order, with the most recent command in element 9. Each element contains the security API command code in the first four characters, and the subcommand code in the last four characters.
10, 12, 14, 16, 18	Last 5 return codes	These five rule-array elements contain the security API return codes and reason codes corresponding to the five commands in rule-array elements 9, 11, 13, 15, and 17. Each element contains the return code in the first four characters, and the reason code in the last four characters.
Output rule-array for option STATEID		
1, 2	EID	The two elements, when concatenated, provide the 16-byte EID value.
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: • A value of 0 means base CCA services are not available. • A value of 1 means base CCA services are available.
2	CDMF availability	A numeric character string containing a value to indicate whether CDMF encryption is available: • A value of 0 means CDMF encryption is not available. • A value of 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: • A value of 0 means 56-bit DES encryption is not available. • A value of 1 means 56-bit DES encryption is available.

Table 4. Cryptographic_Facility_Query information returned in the rule array (continued)

Element number	**Name**	**Description**
4	Triple-DES availability	A numeric character string containing a value to indicate whether Triple-DES encryption is available: • A value of 0 means Triple-DES encryption is not available. • A value of 1 means Triple-DES encryption is available.
5	SET services availability	A numeric character string containing a value to indicate whether SET (secure electronic transaction) services are available: • A value of 0 means SET services are not available. • A value of 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.
Output rule-array for option STATMOFN Elements 1 and 2 are treated as a 16-byte string, as are elements 3 and 4, with the high-order 15 bytes containing meaningful information and the 16th byte containing a space character. Each byte provides status information about the *i*th share, 1≤*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** Cannot 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** Cannot be received **1** Can be received **3** Has been received **4** Has been received more than once

Table 4. Cryptographic_Facility_Query information returned in the rule array (continued)

Element number	Name	Description
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 – 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 – 15, followed by six space characters.
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 number of bytes of data in the *verb-data* variable.

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

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

Required commands

| None

Cryptographic_Resource_Allocate (CSUACRA)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Cryptographic_Resource_Allocate verb is used to allocate a specific CCA coprocessor for use by the thread or process, depending on the scope of the verb. For the i5/OS, this verb is scoped to a process; for the other implementations, this verb is scoped to a thread. When a thread or process, depending on the scope, allocates a cryptographic resource requests are routed to that resource. When a cryptographic resource is not allocated, requests are routed to the default cryptographic resource.

You can set the default cryptographic resource. If you take no action, the default assignment is CRP01.

You cannot allocate a cryptographic resource while one is already allocated. Use the Cryptographic_Resource_Deallocate verb to deallocate an allocated cryptographic resource.

Be sure to review "Understanding multi-coprocessor capabilities" on page 25 and "Master-key considerations with multiple CCA coprocessors" on page 31.

Restrictions

None

Format

CSUACRA			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
resource_name_length	Input	Integer	
resource_name	Input	String	resource_name_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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule-array variable. The value must be 1 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	**Meaning**
Cryptographic resource (required)	
DEVICE	Specifies an IBM 4758 or IBM 4764 CCA Coprocessor.

resource_name_length
The *resource_name_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *resource-name* variable. The length must be 1 – 64.

resource_name
The *resource_name* parameter is a pointer to a string variable containing the name of the coprocessor to be allocated.

Required commands

None

Cryptographic_Resource_Deallocate (CSUACRD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Cryptographic_Resource_Deallocate verb is used to deallocate a specific CCA coprocessor that is allocated by the thread or process, depending on the scope of the verb. For the i5/OS, this verb is scoped to a process; for the other implementations, this verb is scoped to a thread. When a thread or process, depending on the scope, deallocates a cryptographic resource, requests are routed to the default cryptographic resource.

You can set the default cryptographic resource. If you take no action, the default assignment is CRP01.

Be sure to review “Understanding multi-coprocessor capabilities” on page 25 and “Master-key considerations with multiple CCA coprocessors” on page 31.

If a thread with an allocated coprocessor ends without first deallocating the coprocessor, excess memory consumption results. It is not necessary to deallocate a cryptographic resource if the process itself is ending, only if individual threads end while the process continues to run.

Restrictions

None

Format

CSUACRD			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
resource_name_length	Input	Integer	
resource_name	Input	String	resource_name_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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule-array* variable. The value must be 1 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	**Meaning**
Cryptographic resource (required)	
DEVICE	Specifies an IBM 4758 or IBM 4764 CCA Coprocessor.

resource_name_length
The *resource_name_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *resource_name* variable. The length must be 1 – 64.

resource_name
The *resource_name* parameter is a pointer to a string variable containing the name of the coprocessor to be deallocated.

Required commands

None

Key_Storage_Designate (CSUAKSD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758		2.54	
IBM 4764		3.20	
		3.23	

The Key_Storage_Designate verb specifies the key-storage file used by the process.

You select the type of key storage, for DES keys or for public keys, using a rule-array keyword.

Restrictions

None

Format

Key_Storage_Designate			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
key_storage_file_name_length	Input	Integer	≤ 64
key_storage_file_name	Input	String	key_storage_file_name_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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Key-storage type (one required)	
DES	Indicates that the file name applies to the DES key-storage specification.
PKA	Indicates that the file name applies to the public-key key-storage specification.

key_storage_file_name_length

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

key_storage_file_name
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 selected.

Required commands

None

Key_Storage_Initialization (CSNBKSI)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Key_Storage_Initialization verb initializes a key-storage file using the current symmetric or asymmetric master-key. The initialized key storage does not contain any preexisting key records. The name and path of the key storage data and index file are established differently in each operating environment. See the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual* for information on the key storage data and index files.

Restrictions

None

Format

CSNBKSI			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	2
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	24 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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Master-key source (required)	
CURRENT	Specifies that the current symmetric master-key of the default cryptographic facility is to be used for the initialization.
Key-storage selection (one required)	
DES	Initialize DES key-storage.
PKA	Initialize PKA key-storage.

key_storage_file_name_length

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

key_storage_file_name

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 variable containing the number of bytes of data in the *key_storage_description* variable.

key_storage_description

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

clear_master_key

The *clear_master_key* parameter is unused, but it must be declared and point to 24 data bytes in application storage.

Required commands

Except in the i5/OS environment, the Key_Storage_Initialization verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role. In the i5/OS environment, no commands are issued to the coprocessor and command authorization does not apply.

Logon_Control (CSUALCT)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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	In/Output	Integer	
exit_data	In/Output	String	exit_data_length
rule_array_count	Input	Integer	1 or 2
rule_array	Input	String array	rule_array_count * 8 bytes
user_id	Input	String	8 bytes
auth_parms_length	In/Output	Integer	
auth_parms	Input	String	auth_parms_length bytes
auth_data_length	In/Output	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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Keywords used to log on (zero or two)	
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 identify yourself to the coprocessor.
PPHRASE	Specifies that you are going to identify yourself using a *passphrase*.
Keywords used to log off (zero, one, or two)	
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. When you use the **FORCE** keyword, you must also use the **LOGOFF** keyword.
Keywords used to save and restore logon context information (zero or one)	
GET-CNTX	Obtains a copy of the logon context information that is active in your session. See "Using logon context information" on page 22.
PUT-CNTX	Restores the logon context information that was saved using the **GET_CNTX** keyword. See "Using logon context information" on page 22.

user_id

The *user_id* parameter is a pointer to a string variable containing the ID string which identifies the user to the system. The user ID must be exactly 8 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_parms_length

The *auth_parms_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *auth_parms* 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 variable is used differently depending on the authentication method specified in the rule array.

Rule-array keyword	Contents of *auth_parms* variable
PPHRASE	The authentication variable is empty. Its length is zero.

auth_data_length

The *auth_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *auth_data* variable.

On input, this variable contains the length (in bytes) of the *auth_data* variable. When no usage is defined for the *auth_data* parameter, set the length variable to zero.

On output, this variable 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 variable is used differently depending on the keywords specified in the rule array.

Rule-array keyword	**Contents of *auth_data* variable**
PPHRASE and **LOGON**	The authentication data variable contains the user-provided passphrase.
GET-CNTX	The authentication data variable receives the active logon context information. The size of the buffer provided for the *auth_data* variable must be at least 256 bytes.
PUT-CNTX	The authentication data variable contains your active logon context.

Required commands

The Logon_Control verb requires the Force User Logoff command (offset X'011B') to be enabled in the active role for use with the **FORCE** keyword.

Master_Key_Distribution (CSUAMKD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 and, when sufficient shares are received, reconstruct the master key in the new master-key register.

You choose which class of master key, either symmetric or asymmetric, to clone with the **SYM-MK** and the **ASYM-MK** rule-array keywords. If neither keyword is specified, the verb performs the same operation on both classes of registers, provided that the registers contain the same values.

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

Specify the following information when using the **OBTAIN** keyword:
- The share number, *i*, where $1 \le i \le 15$ and $i \le$ the maximum number of shares to be distributed as defined by 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. This key must have the CLONE attribute set at the time of key generation.
- 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* variable that receives the encrypted master-key share.
- The verb performs the following functions:
  - 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_info_encrypting_key
  - Encryption of the clone_info_encrypting_key
- The verb returns the following information:
  - Encrypted cloning information
  - Encrypted clone_info_encrypting_key

Specify the following information when using the **INSTALL** keyword:
- The share number, *i*, presented in this request.
- The private_key_name of the coprocessor-retained key used to decrypt the clone_info_encrypting_key. This key must have the CLONE attribute set at the time of key generation.
- 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_info* variable that provides the encrypted master-key share.
- The verb performs the following tasks:
  – Recovery of the clone_info_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 the following information:
  – A return code of 4 if the master key has been recovered into the new master-key register. A return code of 0 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 empty.

Format

CSUAMKD

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data _length bytes
rule_array_count	Input	Integer	1 or 2
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	certificate_length bytes
clone_info_encrypting_key_length	In/Output	Integer	
clone_info_encrypting_key	In/Output	String	clone_info_encrypting_key_length bytes
clone_info_length	In/Output	Integer	
clone_info	In/Output	String	clone_info_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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and

padded on the right with space characters. The *rule_array* keywords are shown in the following table:

Keyword	Meaning
Operation (one required)	
OBTAIN	Generate and output a master-key share and other cloning information.
INSTALL	Receive a master-key share and other cloning information.
Master-key register class (one, optional)	
SYM-MK	Operate with the symmetric master-key registers.
ASYM-MK	Operate with the asymmetric master-key registers.

share_index
The *share_index* parameter is a pointer to an integer variable containing the index number of the share to be generated or received by the coprocessor.

private_key_name
The *private_key_name* parameter is a pointer to a string variable containing the name of the coprocessor-retained private key used to sign the cloning information or recover the cloning-information encrypting key.

certifying_key_name
The *certifying_key_name* parameter is a pointer to a string variable containing the name of the coprocessor-retained public key used to verify the offered certificate.

certificate_length
The *certificate_length* parameter is a pointer to an integer variable containing the number of bytes of data in the certificate variable.

certificate
The *certificate* parameter is a pointer to a string variable containing the public-key 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 is a pointer to an integer variable containing the number of bytes of data in the *clone_info_encrypting_key* variable.

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

clone_info_length
The *clone_info_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *clone_info* variable.

clone_info
The *clone_info* parameter is a pointer 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
| in the active role based on the requested share-number, $1 \leq i \leq 15$:

- Clone-info (Share) Obtain (offset X'0210'+share_index, for example, for share 10, X'021A') with the **OBTAIN** keyword.
- Clone-info (Share) Install (offset X'0220'+share_index, for example, for share 12, X'022C') with the **INSTALL** keyword.

Master_Key_Process (CSNBMKP)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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

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

For IBM 4758 and IBM 4764 implementations, the master key is a triple-length, 168-bit, 24-byte value.

You can choose to process either the symmetric or asymmetric registers by specifying the **SYM-MK** and the **ASYM-MK** rule-array keywords. If neither keyword is specified, the verb performs the same operation on both classes of registers, provided that the registers already contain the same values.

Tip: Before starting to load new master-key information, ensure that the new master-key register is cleared. Do this by using the **CLEAR** keyword in the rule array.

To form a master key from key parts in the new master-key register, use the verb several times to complete the following tasks:

- Clear the register, if it is not already clear
- Load the first key part
- Load any middle key parts, calling the verb once for each middle key part
- Load the last key part

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 are 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.

For symmetric master-keys, the low-order bit in each byte of the key is used as parity for the remaining bits in the byte. Each byte of the key part must contain an odd number of one bits. If this is not the case, a warning is issued. The product maintains odd parity on the accumulated symmetric master-key value.

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

- If any two of the 8-byte parts of the *new* master-key have the same value, a warning is issued. *Do not ignore this warning.* Do not use a key with this property.
- The master-key verification pattern (MKVP) of the *new* master-key is compared against the MKVP of the *current* and the *old* master-keys. If they are the same, the service fails with return code 8, reason code 704.

- If any of the 8-byte parts of the *new* master-key compares equal to one of the weak DES-keys, the service fails with return code 8, reason code 703. See 77 for a list of these weak keys. A parity-adjusted version of the asymmetric master-key is used to look for weak keys.

Except in the i5/OS environment, as part of the SET process, if a DES or PKA key storage exists, the header record of each key storage is updated with the verification pattern of the new, current master-key. The i5/OS environment does not have master-key verification records in the key-storage data set.

Restrictions

None

Format

CSNBMKP

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1, 2, or 3
rule_array	Input	String array	rule_array_count * 8 bytes
key_part	Input	String	24 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 9.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 1, 2, or 3 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Cryptographic component (optional)	
ADAPTER	Specifies the coprocessor. This is the default.
Master-key register class (one, optional)	
SYM-MK	Specifies operation with the symmetric master-key registers.
ASYM-MK	Specifies operation with the asymmetric master-key registers.
Master-key process (one required)	
CLEAR	Specifies to clear the new master-key register.

Keyword	Meaning
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	Generates 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.

key_part

The key_part parameter is a pointer to a string variable containing a 168-bit (3x56-bit, 24-byte) clear key-part that is used when you specify one of the keywords **FIRST**, **MIDDLE**, or **LAST**.

If you use the **CLEAR**, **RANDOM**, 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 active role:

- To process the symmetric and asymmetric master keys when neither master-key register class is specified:
  - Clear New Master Key Register (offset X'0032') with the **CLEAR** keyword
  - Clear Old Master Key Register (offset X'0033') with the **CLR-OLD** keyword
  - Load First Master Key Part (offset X'0018') with the **FIRST** keyword
  - Combine Master Key Parts (offset X'0019') with the **MIDDLE** or **LAST** keyword
  - Generate Random Master Key (offset X'0020') with the **RANDOM** keyword
  - Set Master Key (offset X'001A') with the **SET** keyword
- To process the asymmetric master-keys:
  - Clear New Asymmetric Master Key Register (offset X'0060') with the **CLEAR** keyword
  - Clear Old Asymmetric Master Key Register (offset X'0061') with the **CLR-OLD** keyword
  - Load First Asymmetric Master Key Part (offset X'0053') with the **FIRST** keyword
  - Combine Asymmetric Master Key Parts (offset X'0054') with the **MIDDLE** or **LAST** keywords

- Generate Random Asymmetric Master Key (offset X'0120') with the **RANDOM** keyword
- Set Asymmetric Master Key (offset X'0057') with the **SET** keyword

Related information

The following are considered questionable DES keys:

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

Random_Number_Tests (CSUARNT)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53		2.53
IBM 4764			

The Random_Number_Tests verb invokes the USA NIST FIPS PUB 140-1 specified cryptographic operational tests. These tests, selected by a rule-array keyword, consist of:
- For random numbers: a monobit test, poker test, runs test, and long-run test
- Known-answer tests of DES, RSA, and SHA-1 processes

The tests are performed three times. If there is any test failure, the verb returns return code 4 and reason code 1.

Restrictions

None

Format

Random_Number_Tests			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	**Meaning**
Test selection (one required)	
FIPS-RNT	Perform the FIPS 140-1 specified test on the random number generation output.
KAT	Perform the FIPS 140-1 specified known-answer tests on DES, RSA, and SHA-1.

Required commands

None

Chapter 3. RSA key-management

This section describes the management of RSA public and private keys and how you can perform the following tasks:

- Generate keys with various characteristics
- Import keys from other systems
- Protect and move a private key from one node to another

The verbs listed in Table 5 are used to perform cryptographic functions and assist you in obtaining key-token data structures. See “Using verbs to perform cryptographic functions and obtain key-token data structures” on page 86 for detailed information on these verbs.

Table 5. Public key key-administration services

Verb	**Page**	**Service**	**Entry point**	**Service location**
PKA_Key_Generate	87	Generates a public-private key-pair.	CSNDPKG	E
PKA_Key_Import	91	Imports a public-private key-pair.	CSNDPKI	E
PKA_Key_Token_Build	93	Builds a public key architecture (PKA) key-token.	CSNDPKB	S
PKA_Key_Token_Change	99	Reenciphers a private key from the old asymmetric master-key to the current asymmetric master-key.	CSNDKTC	E
PKA_Public_Key_Extract	101	Extracts a public key from a public-private public-key token.	CSNDPKX	S
PKA_Public_Key_Hash_Register	103	Registers 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	105	Registers 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.	CSNDPKR	E
E=Cryptographic Engine, S=Security API software				

RSA key-management

With CCA, you can use 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 on page 82 shows the relationship among the services, the public-private key-token, and other data involved with supporting digital signatures and symmetric (DES) key exchange.

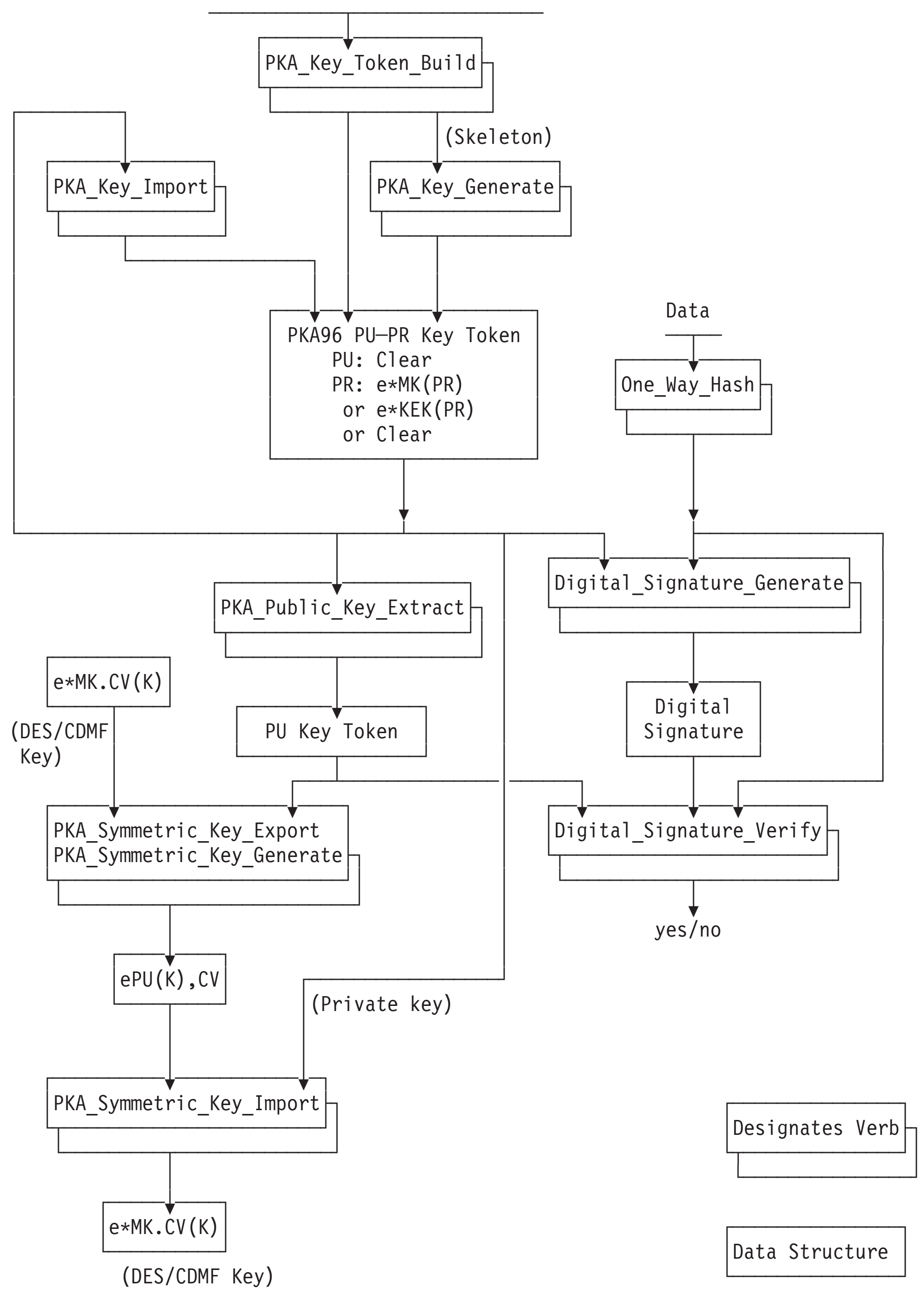


Figure 3. PKA96 verbs with key-token flow

These topics are discussed in this section:
- How to generate a public-private key pair
- How to import keys from other systems
- How to update a private key when the asymmetric master-key that protects a private key is changed
- How to use the keys and provide for private-key protection
- How to use a private key at multiple nodes
- How to register and retain a public key

Key generation

You generate RSA public-private key-pairs using the PKA_Key_Generate verb. You specify certain facts about the desired key in a skeleton key token that you can create using the PKA_Key_Token_Build verb.

When generating the key-pair you must determine:
- The key length
- How, or if, the private key should be encrypted
- If the key should be retained within the coprocessor, and if so, its name (label)
- The form of the private key: modular-exponent or Chinese Remainder
- A key name if access-control on the name is employed
- Whether the key should be usable in symmetric key-exchange operations
- Whether the key should be usable in digital signature generation operations

The PKA_Key_Generate verb either retains the generated private key within the coprocessor, or the verb produces output which is the generated private key in one of three forms, so you can control where the private key is deployed.

You can request that the generated private key be retained within the secure cryptographic-engine using 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 which you specify in the key-name section of the skeleton key-token.

If you do not retain the private key within the coprocessor, you select how to receive the private key:
- Clear text

  Both the 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. With this option, you 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 asymmetric master-key within the cryptographic engine. Because there is no service available to re-encrypt the private key other than by the current or a replacement master-key, the generated private key is effectively locked to the generating node, or other nodes that you establish with the same master key. Generally these would be backup nodes or parallel nodes for greater throughput.
- Enciphered by a transport key-encrypting-key

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

  The key-encrypting key can also 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 or nodes. For example, you could obtain a private key in this form for distribution to a zSeries large server's integrated RSA cryptographic processor.

  Related reading: EXPORTER and IMPORTER key-encrypting transport keys are discussed in Chapter 5, "DES key management."

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 receiving transport key-encrypting key to particular 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, such as z/OS® Security Server RACF®, the key name that you associate with a private key gives you the ability to enforce restricted key usage. These systems can determine whether 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.

For RSA keys, you decide whether the key should be returned in modular-exponent form or in Chinese-Remainder-Theorem (CRT) form. Generally the CRT form performs faster in services that use the private key. 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. Certain implementations such as the IBM zSeries (S/390®) server CMOS Cryptographic Coprocessor feature (CCF) cannot employ a private key in the CRT form generated by the PKA_Key_Generate verb.

For RSA keys, you also 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 (**KEY-MGMT**), or you can indicate that the key should be usable only for digital signature signing (**SIG-ONLY**), or only key decryption (**KM-ONLY**).

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.

You can also have a newly generated public key certified by a private key held within the coprocessor. You can obtain a self-signature, a signature from another key, or both. To obtain these signatures or certificates, you must extend the skeleton key-token yourself because the PKA_Key_Token_Build verb cannot do so.

The formats of the key tokens are described in “RSA PKA key tokens” on page 351. The key tokens are a concatenation of several sections with each section providing information pertaining to the key. All of the described formats can be input to any version, but only selected formats are output by Version 2 and later support.

Key import

To be secure and useful in services, a private key must be enciphered by an asymmetric master-key on the CCA node where it is used. (Of course, a private key generated as a retained private-key is also secure, but in this case PKA_Key_Import does not apply.) You can use the PKA_Key_Import verb to get a private key deciphered from a transport key and enciphered by the asymmetric 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 keys must be presented in a PKA external key-token (see “RSA PKA key tokens” on page 351). 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 returns the private key encrypted under the current asymmetric master-key along with the public key.

The coprocessor is designed to generate and employ RSA CRT-form keys having *p>q*. If you import a private key having *q>p*, the key is accepted. However, each time that you use such a key your application incurs substantial overhead to recalculate the inverse of the quantity *U*. See Table 50 on page 357 for the components of an RSA CRT key.

Reenciphering a private key under an updated master key

When the asymmetric 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 asymmetric 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 reencipherment of the private key.

Using the PKA keys

The public-private keys that you create or import can be used in these services:

For private keys:
- Digital_Signature_Generate
- PKA_Symmetric_Key_Import
- SET_Block_Decompose
- PKA_Decrypt
- Master_Key_Distribution

For public keys:
- Digital_Signature_Verify
- PKA_Symmetric_Key_Export
- PKA_Symmetric_Key_Generate
- SET_Block_Compose
- PKA_Encrypt
- Master_Key_Distribution

You must arrange appropriate protection for the private key. A CCA node can help ensure that the key remains confidential. However, you must ensure that the master key and any transport keys are protected, for example, 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. Generally, this limits the access other applications might have to the key. In systems with an access monitor, such as RACF on z/OS 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 asymmetric 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 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. 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.

Extracting a public key

CCA PKA key generation returns a public-private key-pair in a single key-token, provided your application is not retaining the private key within the coprocessor. You can obtain a key token with only the public-key information using the PKA_Public_Key_Extract verb.

If you use the public-private key token in verbs that only require the public key, the implementation might attempt to recover the private key which in the usual case would fail (because normally the private key should not be usable where use is being made of the public key).

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 registered public key stored within the engine with the confidence that the key has been entered under dual control. The Master_Key_Distribution verb makes use of registered RSA public keys in the master-key shares distribution scheme.

Using verbs to perform cryptographic functions and obtain key-token data structures

PKA_Key_Generate (CSNDPKG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 to the verb determines the following characteristics of the generated key-pair:

- The key type: RSA
- The key length (modulus size)
- The RSA public-key exponent: valued to 3, 2^{16}+1, or random
- Any RSA private-key optimization (modulus-exponent versus 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. See "PKA_Key_Token_Build (CSNDPKB)" on page 93.

Normally the output key is randomly generated. By providing regeneration data, a seed can be supplied so that the same value of the generated key can be obtained in multiple instances. This can 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 might vary between products. Therefore, do not expect to obtain 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 clear text form.
- Enciphered by the CCA asymmetric master-key.
- 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 corresponding IMPORTER key is installed.

Using the **RETAIN** rule-array keyword, you can cause the private key to be retained within the coprocessor. You incorporate the key label by which you later reference the newly generated key in the key name section of the skeleton key-token. Later, you use this label to employ the key in verbs such as Digital_Signature_Generate, PKA_Symmetric_Key_Import, Master_Key_Distribution, SET_Block_Decompose, and PKA_Decrypt. On output, the verb returns an external key-token containing the public key in the generated_key_identifier variable. The generated_key_identifier variable returned from the verb does not contain the private key.

Note: When using the **RETAINED** key option, the key label supplied in the skeleton key-token references the key storage within the coprocessor, and, in this case, must not reference a record in the host-system key-storage.

The rule-array keyword **CLONE** flags a generated and retained RSA private key as usable in an engine cloning process. *Cloning* is a technique for copying sensitive coprocessor information from one coprocessor to another. (See “Understanding and managing master keys” on page 26.)

If you include a public-key certificate section within the skeleton key token, the cryptographic engine signs a certificate with the key that is designated in the public-key certificate signature subsection. This technique causes 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 you include multiple signature subsections in the skeleton key token. See “RSA public-key certificate section” on page 360.

Tip: The verb returns a “section X'06'” private-key token format when you request a modulus-exponent internal key even though you have specified a type X'02' skeleton token.

Restrictions

1. Not all IBM implementations of CCA support a CRT form of the RSA private key; check the product-specific literature. The IBM 4764 and IBM 4758 implementations support an optimized RSA private key (a key in Chinese Remainder form). The formats vary between versions.
2. See “RSA PKA key tokens” on page 351 for the formats used when generating the various forms of key token.
3. When generating a key for use with ANSI X9.31 digital signatures, the modulus length must be 1024, 1280, 1536, 1792, or 2048 bits.
4. The key label used for a retained key must not exist in the external key storage held on the hard disk drive.

Format

CSNDPKG

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1 or 2
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	64 bytes
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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Private-key encryption (one required)	
MASTER	Enciphers the private key under the asymmetric master-key. The transport_key_token should specify a null key-token.
XPORT	Enciphers the private key under the IMPORTER or EXPORTER key-encrypting-key identified by the *transport_key_token* parameter.
CLEAR	Returns the private key in clear text.
RETAIN	Returns the private key within the cryptographic engine and returns the public key 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)	
CLONE	Flags, as usable, a retained private RSA key in a cryptographic engine cloning operation. This keyword requires the **RETAIN** keyword to also be specified.

regeneration_data_length
The *regeneration_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *regeneration_data* variable. This must be a value of 0, or in the range 8 – 256, inclusive. If the value is 0, the generated keys are based on a random-seed value. If this value is between 8 – 256, the regeneration data is hashed to form a seed value used in the key generation process to provide a means for recreating a public-private key pair.

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

skeleton_key_token_length
The *skeleton_key_token_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *skeleton_key_token* variable. The maximum length is 2500 bytes.

skeleton_key_token
The *skeleton_key_token* parameter is a pointer to a string variable 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 variable 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 **XPORT** rule_array keyword is not specified, this parameter points 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 the number of bytes of data in the generated_key_identifier variable. The maximum length is 2500 bytes. On output, and if the size is of sufficient length, the variable is updated with the actual length of the generated_key_identifier variable.

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 to be overwritten. If the key label identifies a key record in key storage, the generated key token replaces any key token associated with the label. If the first byte of the identified string does not indicate a key label (that is, not in the range X'20' to X'FE'), and the variable is of sufficient length to receive the result, then the generated key token is returned in the identified variable.

When generating a retained key, on output the verb returns the public key key-token in this variable.

Required commands

The PKA_Key_Generate verb requires the PKA Key Generate command (offset X'0103') to be enabled in the active role.

Also 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, PKA Clone Key Generate (offset X'0204')
- With the **CLEAR** rule-array keyword, PKA Clear Key Generate (offset X'0205')

PKA_Key_Import (CSNDPKI)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Key_Import verb is used to import a public-private key-pair. A private key must be accompanied by the associated public key. A source private-key can be in the clear or enciphered.

Generally, you obtain the key token from the PKA_Key_Generate verb. If the key originates in a non-CCA system, you can use the PKA_Key_Token_Build verb to create the source_key_token.

The verb deciphers the private key using the DES IMPORTER key identified by the *transport_key_identifier* when the source private-key is enciphered.

Imported keys are returned in an internal target_key_identifier with the private key enciphered by the asymmetric master-key.

Restrictions

- Not all IBM implementations of this verb might support an optimized form of the RSA private-key. Check the product-specific literature. The IBM 4758 and IBM 4764 implementations support an optimized RSA private key (a key in Chinese Remainder form).

  Beginning with Version 2, a clear, external RSA private-key in modulus-exponent format is presented in a key section type X'02'. When imported, the enciphered private-key is returned in a X'06' type private-key key-token section.
- 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	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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
source_key_token_length	Input	Integer	
source_key_token	Input	String	source_key_token_length bytes
transport_key_identifier	Input	String	64 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 9.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 0 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* 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 number of bytes of data in the *source_key_token* variable. 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-key and private-key information. The private key can be in clear text 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 is 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, meaning 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 number of bytes of data in the target_key_identifier variable. The maximum length is 2500 bytes. On output, and if the size is of sufficient length, the variable is updated with the actual length of the target_key_identifier variable.

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, or is other information that is overwritten with the imported key. If the key label identifies a key record in key storage, the returned key-token replaces any key token associated with the label. If the first byte of the identified string does not indicate a key label (that is, not in the range X'20' to X'FE'), and the variable is of sufficient length to receive the result, then the key token is returned in the identified variable.

Required commands

The PKA_Key_Import verb requires the PKA Key Import command (offset X'0104') to be enabled in the active role.

PKA_Key_Token_Build (CSNDPKB)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The PKA_Key_Token_Build verb constructs a public-key architecture (PKA) key-token from the supplied information.

This verb is used to create the following:
- A skeleton_key_token for use with the PKA_Key_Generate verb
- A key token with a public key that has been obtained from another source
- A key token with a clear private-key and the associated public key

Other than a skeleton key-token prepared for use with the PKA_Key_Generate verb, every PKA key-token contains a public-key value. A token optionally contains a private-key value.

See “RSA PKA key tokens” on page 351 for a description of the key token formats. You can create RSA private-key tokens for section types:

X'08'	using the **RSA-CRT** keyword to obtain a token format for a key usable with the Chinese-Remainder Theorem (CRT) algorithm.
X'02'	using the **RSA-PRIV** keyword to obtain a token format for a key in modulus-exponent form
X'04'	using the **RSA-PUBL** keyword to obtain a token format for a public key.

Specify the following factors:
- The token type:
  - **RSA-CRT** for an RSA CRT token
  - **RSA-PRIV** for an RSA modulus-exponent token
  - **RSA-PUBL** for an RSA public key-only token
- The usage limits for a private key:
  - If an RSA private-key can be allowed to import a symmetric key, and the key can also be used to create digital signatures, include the **KEY-MGMT** keyword in the rule array.
  - If a private key cannot be used in digital signature generation, include the **KM-ONLY** keyword in the rule array.
  - If an RSA private-key cannot be used in importing of DES keys, you can include the **SIG-ONLY** keyword in the rule array. This is the default.
- A key name when:
  - You need to specify the key-label for a retained private key in a skeleton key-token.

Restrictions

- The **RSA-OPT** rule-array keyword is not supported beginning with Version 2. Instead, use keyword **RSA-CRT** to obtain a X'08' private-key section type.
- The RSA key length is limited to the range of 512 to 2048 bits with specific formats restricted to 1024 bits maximum.

- When generating a key for use with ANSI X9.31 digital signatures, the key length must be 1024, 1280, 1536, 1792, or 2048 bits.

Format

CSNDPKB

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1 or 2
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	key_name_length bytes
reserved_1_length	Input	Integer	0
reserved_1	Input	String	null
reserved_2_length	Input	Integer	0
reserved_2	Input	String	null
reserved_3_length	Input	Integer	0
reserved_3	Input	String	null
reserved_4_length	Input	Integer	0
reserved_4	Input	String	null
reserved_5_length	Input	Integer	0
reserved_5	Input	String	null
token_length	In/Output	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 9.

rule_array_count

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

rule_array

The rule_array parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Token type (one required)	
RSA-CRT	Create a key token for an RSA public key and a key in Chinese-Remainder form.

Keyword	Meaning
RSA-OPT	**Note:** This keyword is not supported with Version 2 and later software.
RSA-PRIV	Create a key token for an RSA public and private key pair in modulus-exponent form.
RSA-PUBL	Create a key token for an RSA public key in modulus-exponent form.
RSA key-usage control (one, optional)	
SIG-ONLY	Selects a usage control to render the private key usable in digital-signature operations but not in (DES) key import operations. This is the default.
KEY-MGMT	Selects a usage control that allows an RSA private-key to be used in distribution of symmetric keys and in digital-signature services.
KM-ONLY	Selects a usage control to render the private key usable in (DES) key-import operations but not in digital-signature operations.

key_values_structure_length
The *key_values_structure_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *key_values_structure* variable. The maximum length is 2500 bytes.

key_values_structure
The *key_values_structure* parameter is a pointer to a string variable containing a structure of the lengths and data for the components of the key or keys. The contents of this structure are shown in Table 6, and sample data is described in "Related information" on page 98.

Note: The following table has these conditions:
- All length fields are in binary
- All binary fields (exponents, lengths, and so on) are stored with the high-order byte first (left, low-address, big endian, S/390 format)

Table 6. PKA_Key_Token_Build key-values-structure contents

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 1024 for RSA-PRIV, 512 to 2048 for RSA-PUBL).
002	002	Length of the modulus field, *n*, in bytes, *nnn*. This value must not exceed 256 for a 2048 bit-length 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.

Table 6. PKA_Key_Token_Build key-values-structure contents (continued)

008	*nnn*	Modulus, n, integer value, $1<n<2^{2048}$; $n=pq$ for prime p and prime q.
8+*nnn*	*eee*	Public exponent field, 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 0 (zero) to indicate that a full-random exponent should be generated. The exponent field can be a null-length field when preparing a skeleton_key_token.
8+*nnn* +*eee*	*ddd*	Private exponent, d, integer value, $1<d<n$, $d=e^{-1}\bmod(p-1)(q-1)$.
RSA key-values structure, Chinese Remainder form (**RSA-CRT**)		
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 is 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 field, e, 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	Reserved, binary zero.
008	002	Length of the prime number field, p, in bytes: *ppp*. (Can be zero in a skeleton_key_token.) The maximum value of *ppp+qqq* is 256 bytes.
010	002	Length of the prime number field, q, in bytes: *qqq*. (Can be zero in a skeleton_key_token.) The maximum value of *ppp+qqq* is 256 bytes.
012	002	Length of the d_p field, in bytes: *rrr*. (Can be zero in a skeleton_key_token.) The maximum value of *rrr+sss* is 256 bytes.
014	002	Length of the d_q field, in bytes: *sss*. (Can be zero in a skeleton_key_token.) The maximum value of *rrr+sss* is 256 bytes.
016	002	Length of the U field, in bytes: “uuu”. (Can be zero in a skeleton_key_token.) The maximum length of U is 256 bytes.
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 be 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.

Table 6. PKA_Key_Token_Build key-values-structure contents (continued)

018 +*nnn* +*eee* +*ppp*	*qqq*	Prime number, *q*.
018 +*nnn* +*eee* +*ppp* +*qqq*	*rrr*	$d_p = d \bmod(p\text{-}1)$.
018 +*nnn* +*eee* +*ppp* +*qqq* +*rrr*	*sss*	$d_q = d \bmod(q\text{-}1)$.
018 +*nnn* +*eee* +*ppp* +*qqq* +*rrr* +*sss*	*uuu*	$U = q^{-1}\bmod(p)$.

key_name_length
The *key_name_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *optional key_name* variable. 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 value must be 64 for this verb.

key_name
The *key_name* parameter is a pointer to a string variable containing the name of the key. The name of the key can consist of alphanumeric characters, the pound sign (#), the at sign (@), the dollar sign ($), and the period (.), and must begin with an alphabetic character. See “Key-label content” on page 242.

reserved_x_lengths
The *reserved_x_length* parameters are each a pointer to an integer variable containing the number of bytes of data in the corresponding *reserved_x* variable. These variables are reserved for future use, and each variable must contain zero.

reserved_xs
The *reserved_x* parameters are each a pointer to a string variable that is reserved for future use. Each of the *reserved_x* parameters must contain a null pointer.

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

token
The *token* parameter is a pointer to a string variable containing the assembled token returned by the verb.

Related information

Samples for the *key_values_structure* are shown below (and see the note following the examples).

Token type	Key length (bits)	Public exponent	Key-values structure (hexadecimal)	Structure length (bytes)
RSA-CRT	512	Random (0)	0200 0000 0000 0000 0000 0000 0000 0000 0000	18
RSA-CRT	512	3	0200 0000 0001 0000 0000 0000 0000 0000 0000 03	19
RSA-CRT	512	65537	0200 0000 0003 0000 0000 0000 0000 0000 0000 010001	21
RSA-CRT	768	Random (0)	0300 0000 0000 0000 0000 0000 0000 0000 0000	18
RSA-CRT	768	3	0300 0000 0001 0000 0000 0000 0000 0000 0000 03	19
RSA-CRT	768	65537	0300 0000 0003 0000 0000 0000 0000 0000 0000 010001	21
RSA-CRT	1024	Random (0)	0400 0000 0000 0000 0000 0000 0000 0000 0000	18
RSA-CRT	1024	3	0400 0000 0001 0000 0000 0000 0000 0000 0000 03	19
RSA-CRT	1024	65537	0400 0000 0003 0000 0000 0000 0000 0000 0000 010001	21
RSA-CRT	2048	Random (0)	0800 0000 0000 0000 0000 0000 0000 0000 0000	18
RSA-CRT	2048	3	0800 0000 0001 0000 0000 0000 0000 0000 0000 03	19
RSA-CRT	2048	65537	0800 0000 0003 0000 0000 0000 0000 0000 0000 010001	21
RSA-PRIV	512	Random (0)	0200 0000 0000 0000	8
RSA-PRIV	512	3	0200 0000 0001 0000 3	9
RSA-PRIV	512	65537	0200 0000 0003 0000 010001	11
RSA-PRIV	768	Random (0)	0300 0000 0000 0000	8
RSA-PRIV	768	3	0300 0000 0001 0000 3	9
RSA-PRIV	768	65537	0300 0000 0003 0000 010001	11
RSA-PRIV	1024	Random (0)	0400 0000 0000 0000	8
RSA-PRIV	1024	3	0400 0000 0001 0000 3	9
RSA-PRIV	1024	65537	0400 0000 0003 0000 010001	11

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)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The PKA_Key_Token_Change verb changes RSA private keys from encipherment with the old asymmetric master-key to encipherment with the current asymmetric 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

Not all CCA implementations support this verb.

Format

CSNDTKC			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
key_identifier_length	In/Output	Integer	
key_identifier	In/Output	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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Table 7. PKA_Key_Token_Change rule_array keyword

Keyword	Meaning
Encipherment type (required)	
RTCMK	Changes an RSA private key from encipherment with the old asymmetric master-key to encipherment with the current asymmetric master-key.

key_identifier_length

The *key_identifier_length* parameter is a pointer to an integer variable containing the number of bytes of data in the key_identifier variable. On output, the variable contains the length of the key token returned by the verb if a key token (not a key label) was specified. The maximum 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 private key within the token is securely reenciphered under the current asymmetric master-key.

Required commands

When you specify the reencipher option, the PKA_Key_Token_Change verb requires the Key Token Change command (offset X'0102') to be enabled in the active role.

PKA_Public_Key_Extract (CSNDPKX)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 can be in the clear or enciphered.

Restrictions

None

Format

CSNDPKX

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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
source_key_identifier_length	Input	Integer	
source_key_identifier	Input	String	source_key_identifier_length bytes
target_key_token_length	In/Output	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 9.

rule_array_count
The rule_array_count parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 for this verb.

rule_array
The rule_array parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array parameter is not presently used by this verb, but must be specified.

source_key_identifier_length
The source_key_identifier_length parameter is a pointer to an integer variable containing the number of bytes of data in the *source_key_identifier* variable. 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 PKA key-storage record or a PKA96 key-token.

target_key_token_length
The target_key_token_length parameter is a pointer to an integer variable containing the number of bytes of data in the *target_key_token* variable. On output, the variable contains the length of the key token returned by the verb. The maximum length is 2500 bytes.

target_key_token
The target_key_token parameter is a pointer to a string variable containing the PKA96 public-key token returned by the verb.

Required commands

None

PKA_Public_Key_Hash_Register (CSNDPKH)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

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	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1 or 2
rule_array	Input	String array	rule_array_count * 8 bytes
public_key_name	Input	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 9.

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

rule_array
The rule_array parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Hash type (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 public key is accessed.

hash_data_length
The *hash_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the hash_data variable.

hash_data
The *hash_data* parameter is a pointer to a string variable containing the SHA-1 hash of a public-key certificate that is 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 360.

Required commands

The PKA_Public_Key_Hash_Register verb requires the PKA Register Public Key Hash command (offset X'0200') to be enabled in the active role.

PKA_Public_Key_Register (CSNDPKR)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 is the key that is registered. A preregistered 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 PKA key tokens" on page 351 for more information.

Restrictions

None

Format

CSNDPKR			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0 or 1
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	public_key_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 9.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 or 1 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

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 public key is accessed.

public_key_certificate_length

The *public_key_certificate_length* parameter is a pointer to an integer variable containing the number of bytes of data in the public_key_certificate variable.

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 360.

Required commands

The PKA_Public_Key_Register verb requires the PKA Public Key Register command (offset X'0201') to be enabled in the active role.

If you specify the **CLONE** rule-array keyword, also enable the PKA Public Key Register with Cloning command (offset X'0202').

Chapter 4. Hashing and digital signatures

This section discusses the data hashing and the digital signature techniques you can use to determine data integrity. A digital signature might also be used to establish the nonrepudiation security property. (Another approach to data integrity based on DES 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.
- Nonrepudiation permits you to assert that the originator of a digital signature might 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 use both hashing and public-key cryptography.

The following table describes verbs used in hashing and digital signature services. See "Verbs used in hashing and digital signature services" on page 109 for a detailed description of the verbs.

Table 8. Hashing and digital signature services

Verb	Page	Service	Entry point	Service location
Digital_Signature_Generate	110	This verb generates a digital signature.	CSNDDSG	E
Digital_Signature_Verify	114	This verb verifies a digital signature.	CSNDDSV	E
MDC_Generate	117	This verb generates a hash using the Modification Detection Code (MDC) one-way function.	CSNBMDG	E
One_Way_Hash	120	This verb generates a hash using any of the SHA-1, MD5, or RIPEMD160 one-way hashing functions.	CSNBOWH	S/E
E=Cryptographic Engine, S=Security API software				

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 quantity called a *hash value* (also referred to as a hash, a message digest, or a fingerprint). Common hashing functions produce hash values of 128 or 160 bits. While many different strings supplied to a cryptographically-useful hashing function produce the same hash value, it is computationally infeasible to determine a modification to a data string that results in a desired hash-value.

Hash functions for data integrity applications have a one-way property: given a hash value, it is highly improbable that a second data string can be found that hashes 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 hash values. If both hash values are identical, there is a very high probability that the strings producing them are identical.

The CCA products use the following hash functions:

Secure Hash Algorithm-1 (SHA-1)
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.

RIPEMD-160
RIPEMD-160 is a 160-bit cryptographic hash function. It is intended to be used as a secure replacement for the 128-bit hash functions MD5 and RIPEMD.

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, general-purpose computers (for example, Intel).

Modification Detection Code (MDC)
MDC is based on the DES algorithm and produces a 16-byte, 128-bit hash value. This hashing algorithm is considered quite strong. However, it performs rapidly only when supported by DES-hardware units specifically designed for MDC. See “Modification detection code calculation methods” on page 411 for a description of the MDC algorithm.

Note: The SHA-1 method is specified in FIPS 180-1, May 31, 1994. The MD5 method is specified in RFC 1321, dated April 1992. The RIPEMD-160 method is an outgrowth of the EU project RIPE (RACE Integrity Primitives Evaluation); further information can be found on the Internet under “RIPEMD”.

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 wanting 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 rehash 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 section 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 not possible 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 value is called a *digital signature*, and relies on both public-key cryptography and hashing. To create a digital signature, hash the data and encrypt the results of the hash using your private key.

Anyone with access to your public key can verify your information by performing the following steps:

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.
4. An equal comparison confirms that the data they possess is the same as the data that you signed (on which you created the digital signature).

The Digital_Signature_Generate and the Digital_Signature_Verity verbs described in this section perform the hash encrypting and decrypting operations. The requirements are:

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

  Tip: Verifying public keys also involves using digital signatures.
- 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. Because 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. With the digital signature verbs (Verify and Generate), you can use several different hash-value-formatting approaches. The rule-array keywords for the digital signature verbs contain brief descriptions of these formatting approaches:
  - ANSI X9.31
  - ISO 9796-1
  - PKCS #1 block type 00
  - PKCS #1 block type 01 (RSA PKCS #1 v2.0 standard, RSASSA-PKCS-v1_5)
  - Padding with zero bits

The receiver of data signed using digital signature techniques can, in some cases, assert *non-repudiation* of the data. *Non-repudiation* means that the originator of the digital signature cannot later deny having originated the signature and, therefore, 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 section are the most common methods of digital signing.

Verbs used in hashing and digital signature services

Digital_Signature_Generate (CSNDDSG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Digital_Signature_Generate verb is used to generate a digital signature.

You supply the following information:
- A rule-array keyword to select the hash-formatting method
- The RSA private key
- For ANSI X9.31, the hashing method specification
- The hash value
- The address where the verb returns the digital signature

The hash quantity can be created using the One_Way_Hash or the MDC_Generate verb.

Restrictions

- A private key flagged as a key-management-only key (in the private-key section, offset 50, valued to X'C0') is not usable in this verb. See “PKA_Key_Token_Build (CSNDPKB)” on page 93 and “PKA_Key_Generate (CSNDPKG)” on page 87.
- Starting with Release 3, the hash length is restricted to less than or equal to 36 bytes when using a private key flagged as a key-management-capable key (in the private-key section, offset 50, high-order bit on). You can override this restriction with the use of the Override DSG Zero-Pad Length Restriction, command offset X'030C'.
- Not all IBM implementations of this verb might use an optimized form of the RSA private key, however, the IBM 4758 and IBM 4764 can use this verb with an optimized RSA private key in Chinese Remainder form.
- Not all CCAs use each formatting method.
- The modulus length of a key used with ANSI X9.31 digital signatures must be one of 1024, 1280, 1536, 1792, or 2048 bits.

Format

CSNDDSG			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0, 1, or 2
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	In/Output	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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0, 1, or 2 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Digital-signature-hash formatting method (one, optional)	
X9.31	Formats the hash according to the ANSI X9.31 standard and generates the digital signature.
PKCS-1.1	Calculates 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. See “PKCS #1 Hash Formats” on page 425.
ISO-9796	Formats the hash according to the ISO 9796-1 standard and generates the digital signature. This is the default. See “Formatting hashes and keys in public-key cryptography” on page 425.
PKCS-1.0	Calculates 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. See “PKCS #1 Hash Formats” on page 425.
ZERO-PAD	Places the supplied hash-value in the low-order bit positions of a bit-string of the same length as the modulus. Sets each non-hash-value bit position to 0. Ciphers the resulting bit-string to obtain the digital signature.
Hashing method specification When using **X9.31** formatting, specify one.	
SHA-1	Specifies that the hash is to be generated using the SHA-1 algorithm.
RPMD-160	Specifies that the hash is to be generated using the RIPEMD-160 algorithm.

Tips:
1. Use the MD5 or SHA-1 algorithms to create the hash for **PKCS-1.1** and **PKCS-1.0**.
2. You can use any hashing method to obtain the hash for **ISO-9796** and **ZERO-PAD**.
3. See “Formatting hashes and keys in public-key cryptography” on page 425 for a discussion of hash formatting methods.

PKA_private_key_identifier_length
The *PKA_private_key_identifier_length* parameter is a pointer to an integer

variable containing the number of bytes of data in the *PKA_private_key_identifier* variable. The maximum length is 2500 bytes.

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

hash_length
The *hash_length* parameter is a pointer to an integer variable containing the number of bytes of data in the hash variable.

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

Note:

- 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. This verb requires the input text to be a byte multiple up to the correct maximum length.
- 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-byte or 20-byte hash values, respectively:

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

- For **ZERO-PAD**, the information identified by the hash parameter must be less than or equal to the number of bytes required to contain the modulus of the RSA key. If the private key permits both signature and key-management usage, the hash length is restricted to not more than 36 bytes.
- See “Formatting hashes and keys in public-key cryptography” on page 425 for a discussion of hash formatting methods.

signature_field_length
The *signature_field_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *signature_field* variable. On output, if the size is sufficient, the variable contains the actual length of the digital signature returned by the verb. The maximum length is 256 bytes.

signature_bit_length
The *signature_bit_length* parameter is a pointer to an integer variable containing the number of bits of data of the digital signature returned in the *signature_field* variable.

signature_field
The *signature_field* parameter is a pointer to a string variable containing the stored digital signature. 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 active role.

With the use of the Override DSG Zero-Pad Length Restriction command (offset X'030C'), the hash-length restriction does not apply when using **ZERO-PAD** formatting.

Digital_Signature_Verify (CSNDDSV)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The Digital_Signature_Verify verb is used to verify a digital signature.

Provide the digital signature, the public key, the hash formatting method, and the hash of the data to be validated. The hash quantity might be created through use of the One_Way_Hash or the MDC_Generate verbs.

The hash formatting method is selected through keywords in the rule array. The supplied hash information is formatted and compared to the public-key ciphered digital signature.

If the digital signature is validated, the verb returns a return code of 0. If the digital signature is not validated, and there are no other problems, the verb returns a return code of 4 and reason code of 429 (decimal).

Restrictions

Not all CCA implementations support each formatting method.

Format

CSNDDSV			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0 or 1
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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Digital-signature-hash formatting method (one, optional)	
X9.31	Formats the hash according to the ANSI X9.31 standard and compare to the digital signature. See “Formatting hashes and keys in public-key cryptography” on page 425.
PKCS-1.1	Formats the hash as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 01 and compares to the digital signature. See “PKCS #1 Hash Formats” on page 425.
ISO-9796	Formats the hash according to the ISO 9796-1 standard and compare to the digital signature. This is the default. See “Formatting hashes and keys in public-key cryptography” on page 425.
PKCS-1.0	Formats the hash as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 00 and compare to the digital signature. See “PKCS #1 Hash Formats” on page 425.
ZERO-PAD	Specifies that the supplied hash value is to be placed in the low-order bit positions of a bit-string of the same length as the modulus with all non-hash-value bit positions set to zero. After ciphering the supplied digital signature, the result is compared to the hash-extended bit string.

Notes:

1. Use the MD5 or the SHA-1 hash algorithms to create the **PKCS-1.1** and **PKCS-1.0** formats.
2. You can use any hashing method for digital signatures for **ISO-9796** and **ZERO-PAD** formats.

PKA_public_key_identifier_length

The *PKA_public_key_identifier_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *PKA_public_key_identifier* variable. The maximum length is 2500 bytes.

PKA_public_key_identifier

The *PKA_public_key_identifier* parameter is a pointer to a string variable containing either a key label identifying a key-storage record or a registered public-key, or a key token.

hash_length

The *hash_length* parameter is a pointer to an integer variable containing the number of bytes of data in 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. This verb requires the input text to be a byte multiple up to the correct 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-byte 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 less 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 number of bytes of data in the *signature_field* variable.

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 active role.

MDC_Generate (CSNBMDG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

Use the MDC_Generate verb to create a 128-bit hash value on a data string whose integrity you intend to confirm. After using this verb to generate an MDC, you can compare the MDC to a known value or communicate the value to another entity so that they can compare the MDC hash value to one that they calculate.

The MDC_Generate verb enables you to perform the following tasks:
- Specify the two-encipherment or four-encipherment version of the algorithm
- Segment your text into a series of verb calls

You can also use the verb as a keyed hash algorithm. See the Related Information at the end of this verb description.

Specifying two or four encipherments: Four encipherments per algorithm round improve security; two encipherments per algorithm round improve performance. To specify the number of encipherments, use the **MDC-2**, **MDC-4**, **PADMDC-2**, or **PADMDC-4** keyword with the rule_array parameter. Two encipherments create results that differ from four encipherments; ensure that you use the same number of encipherments to verify the MDC.

For a description of the MDC calculations, see “Modification detection code calculation methods” on page 411.

Segmenting text: The MDC_Generate verb lets you segment text into a series of verb calls. If you can present all of the data to be hashed in a single invocation of the verb, use the rule array keyword **ONLY**. You can segment your text and present the segments with a series of verb calls. Use the rule array keywords **FIRST** and **LAST** for the first and last segments. If you use more than two segments, use the rule array keyword **MIDDLE** for the additional segments.

Between verb calls, unprocessed text data and intermediate information from the partial MDC calculation is stored in the *chaining_vector* variable and the MDC key in the MDC variable. During segmented processing, the application program must not change the data in either of these variables.

Restrictions

- When padding is requested (by specifying a process rule of **PADMDC-2** or **PADMDC-4** in the rule_array variable), a text length of zero is valid for any segment-control specified in the rule_array variable **FIRST**, **MIDDLE**, **LAST**, or **ONLY**). When **LAST** or **ONLY** is specified, the supplied text is padded with X'FF' bytes and a padding count in the last byte to bring the total text length to the next multiple of 8 that is greater than or equal to 16.
- When no padding is requested (by specifying a process rule of **MDC-2** or **MDC-4** in the rule_array variable), the total length of text provided (over a single or segmented calls) must be at least 16 bytes and a multiple of 8 bytes. For segmented calls, a text length of zero is valid on any of the calls.

Format

CSNBMDG			
return_code	Input	Integer	
reason_code	Input	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
text_length	Input	Integer	
text	Input	String	text_length bytes
rule_array_count	Input	Integer	0, 1, or 2
rule_array	Input	String array	rule_array_count * 8 bytes
chaining_vector	In/Output	String	18 bytes
MDC	In/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 9.

text_length
The *text_length* parameter is a pointer to an integer variable containing the length, in bytes, of text to process.

text
The *text* parameter is a pointer to a string variable containing the text for which the verb calculates the MDC value.

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 0, 1, or 2 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Segmenting and Key Control (one, optional)	
ONLY	Specifies that segmenting is not used and the default key is used. This is the default.
FIRST	Specifies the first segment of text, and use of the default key.
MIDDLE	Specifies an intermediate segment of text, or the first segment of text and use of a user-supplied key.
LAST	Specifies the last segment of text, or that segmenting is not used, and use of a user-supplied key.
Algorithm Mode (one, optional)	
PADMDC-2	Specifies two encipherments for each 8-byte block using PADMDC procedures.
PADMDC-4	Specifies four encipherments for each 8-byte block using PADMDC procedures.
MDC-2	Specifies two encipherments for each 8-byte block using MDC procedures. This is the default.

Keyword	Meaning
MDC-4	Specifies four encipherments for each 8-byte block using MDC procedures.

chaining_vector

The *chaining_vector* parameter is a pointer to an 18-byte string variable the security server uses as a work area to hold segmented data between verb invocations.

Tip: When segmenting text, the application program must not change the data in this string between verb calls to the MDC_Generate verb.

MDC

The *MDC* parameter is a pointer to a user-supplied MDC key or to a 16-byte string variable containing the MDC value. This value can be the key that the application program provides. This variable is also used to hold the intermediate MDC result when segmenting text.

Important: When segmenting text, the application program must not change the data in this string between verb calls to the MDC_Generate verb.

Required commands

The MDC_Generate verb requires the Generate MDC command (offset X'008A') to be enabled in the active role.

Related information

The MDC_Generate verb uses a default key when you specify **ONLY** or **FIRST** keywords. If you want to use the MDC as a keyed-hash algorithm, place the key into the MDC variable and ensure that the chaining_vector variable is set to null (18 bytes of X'00'). Then, for a single segment of text, use the **LAST** keyword. For multiple segments of text, begin with the **MIDDLE** keyword and then use additional calls specifying **MIDDLE** as required and, finally, **LAST**. As with the default key, you must not alter the value of the MDC or *chaining_vector* variables between calls.

One_Way_Hash (CSNBOWH)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The One_Way_Hash verb obtains a hash value from a text string using the MD5, SHA-1, or RIPEMD-160 hashing methods, which you specify 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. The keywords that you supply in the rule_array determine your intention.

For the SHA-1 hash algorithm, the verb hashes longer text strings using the coprocessor hardware, with shorter text strings hashed by software in the host computer. It is faster to process short text strings in the host computer, while it is faster to process long strings in the coprocessor.

Note: Hashing can also be performed using the MDC_Generate verb (CSNCMDG) for the (MDC-2, MDC-4,) PADMDC-2, and PADMDC-4 methods.

Restrictions

If **FIRST** or **MIDDLE** calls are made, the text size must be a multiple of the algorithm block size: 64 bytes.

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.

Format

CSNBOWH			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1 or 2
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	128 bytes
chaining_vector	In/Output	String	chaining_vector_length bytes
hash_length	Input	Integer	16 or 20 bytes
hash	In/Output	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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Hash method (one required)	
MD5	Specifies the use of the MD5 method
SHA-1	Specifies the use of the SHA-1 method
RPMD-160	Specifies the use of the RIPEMD-160 method
Segmenting control (one, 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 number of bytes of data in the text variable. The maximum length on i5/OS systems is 64 MB – 64 bytes, and on the other systems is 32 MB – 64 bytes.

Restriction: If **FIRST** or **MIDDLE** calls are made, the text size must be a multiple of the algorithm block-size.

text

The *text* parameter is a pointer to a string variable containing the data on which the hash value is computed.

chaining_vector_length

The *chaining_vector_length* parameter is a pointer to an integer variable containing the number of bytes of data in the chaining_vector variable. The value must be 128 for this verb.

chaining_vector

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

hash_length

The *hash_length* parameter is a pointer to an integer variable containing the number of bytes of data in the hash variable. This value 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 containing the hash value returned by the verb. With use of the **FIRST** or **MIDDLE** keywords, the hash variable receives intermediate results.

Required commands

The One-Way-Hash verb requires the One-Way-Hash, SHA-1 command (offset X'0107') to be enabled in the active role when calculating the hash for a text length greater than 8192 bytes.

Chapter 5. DES key management

This section describes verbs used to perform basic CCA DES key-management functions. Table 9 lists the verbs covered in this section. The following topics are presented:

- CCA DES key management
- Control vectors, key types, and key-usage restrictions
- Key tokens, key labels, and key identifiers
- Key-processing and key-storage verbs
- Security precautions

The following table lists the basic CCA DES key-management verbs. See "CCA DES key-management verbs" on page 144 for a detailed description of the verbs.

Table 9. Basic CCA DES key-management verbs

Verb	Page	Service	Entry point	Service location
Clear_Key_Import	145	Enciphers a clear key under the symmetric master-key, and updates or creates an internal key-token for a DATA key.	CSNBCKI	E
Control_Vector_Generate	146	Builds a control vector from keywords.	CSNBCVG	S
Control_Vector_Translate	148	Changes the control vector associated with a key in an external key-token.	CSNBCVT	E
Cryptographic_Variable_Encipher	151	Encrypts modest quantities of data using a unique key-class, CVARENC. The service is used to prepare the mask-array variable for the Control_Vector_Translate verb.	CSNBCVE	E
Data_Key_Export	153	Exports a DES data-key and creates an external key-token that contains a null control vector.	CSNBDKX	E
Data_Key_Import	155	Imports a DES data-key and creates an internal key-token for the key.	CSNBDKM	E
Diversified_Key_Generate	157	Generates a DES key based on supplied information and a key-generating key. The verb is often used in generating keys for use with smart cards.	CSNBDKG	E
Key_Encryption_Translate	158	Translates an encrypted DATA key with an all-zero control vector from ECB mode to CBC mode, or from CBC mode to ECB mode.	CSNBKET	E
Key_Export	167	Exports a DES key and creates an external key-token.	CSNBKEX	E

Table 9. Basic CCA DES key-management verbs (continued)

Verb	Page	Service	Entry point	Service location
Key_Generate	169	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	176	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	179	Combines clear key parts, enciphers the key, and performs updates.	CSNBKPI	E
Key_Test	183	Generates or verifies a verification pattern for keys and key parts.	CSNBKYT	E
Key_Test_Extended	187	Generates or verifies a verification pattern for keys and key parts on internal and external keys.	CSNBKYT	E
Key_Token_Build	191	Creates a DES key-token from supplied information.	CSNBKTB	S
Key_Token_Change	194	Reenciphers a DES key from the old symmetric master-key to the current symmetric master-key.	CSNBKTC	E
Key_Token_Parse	196	Parses a DES key-token and provides the contents as individual variables.	CSNBKTP	S
Key_Translate	200	Changes the encipherment of a key from one key-encrypting key to another key-encrypting key.	CSNBKTR	E
Multiple_Clear_Key_Import	202	Imports DES keys to form a double-length DES data-key.	CSNBCKM	E
PKA_Decrypt	204	Uses an RSA private-key to decrypt a symmetric key formatted in an RSA DSI PKDS #1 block type 2 structure and return the symmetric key in the clear.	CSNDPKD	E
PKA_Encrypt	206	Uses an RSA public-key to encrypt a clear symmetric-key in an RSA DSI PKCS #1 block type 2 structure and return the encrypted key. Using the **ZERO-PAD** option, you can encipher information including a hash to validate digital signatures such as ISO 9796-2.	CSNDPKE	E
PKA_Symmetric_Key_Export	209	Exports a symmetric key under an RSA public key.	CSNDSYX	E

Table 9. Basic CCA DES key-management verbs (continued)

Verb	Page	Service	Entry point	Service location
PKA_Symmetric_Key_Generate	212	Generates a new DES key and returns one copy multiply-enciphered under the symmetric master-key or a DES key-encrypting key and another copy enciphered under an RSA public key.	CSNDSYG	E
PKA_Symmetric_Key_Import	216	Imports a symmetric key under an RSA private key.	CSNDSYI	E
Prohibit_Export	220	Modifies an internal key so it can no longer be exported.	CSNBPEX	E
Prohibit_Export_Extended	221	Modifies an external key so it can no longer be exported once it has been imported.	CSNBPEXX	E
Random_Number_Generate	222	Generates a random number.	CSNBRNG	E
E=Cryptographic Engine, S=Security API software				

CCA DES key management

The DES algorithm operates on 64 data-bits at a time: eight 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 is 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 places and times, 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 using different roles. To activate a profile and its role to enable different hardware capabilities, users, or programs must supply identification and a password for verification. Using these 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 program 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 to separate the cryptographic keys into a set of key types and restrict the use of a key. A control vector is a logical extension of a key variant, which is a method of key separation that some other cryptographic systems use. 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 cannot be employed in the decipher verb, thereby preventing the use of a key-encrypting key to obtain a cleartext key.

Securely installing and verifying keys
Capabilities are provided to install 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.

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

Your application programs 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 might minimize the resulting damage should an adversary determine 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 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 first and the second parts of a master key (each 56-bit component) are required to be unique. For compatibility, the third part can be the same as the first part, thus creating an effective double-length master-key.

Key-encrypting keys, sometimes designated transport keys, are double-length DES keys composed of two halves, each half being a 56-bit DES key. The halves of a key-encrypting key can be the same value, in which case the key-encrypting key operates as though it were a single-length, 56-bit, DES key.

A key that is multiply-enciphered under the master key is an *operational key*. The key is operational because a cryptographic facility can use the master key to multiply-decipher it to obtain the 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 KEK whose control vector provides key-exporting authority.

Control vectors, key types, and key-usage restrictions

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.

CCA implementations 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 carried in the clear in the key token along with the encrypted key that it specifies.

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. Any change to the original control vector would result in later recovering an altered key-value. 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 processes are described in detail at “Understanding CCA key encryption and decryption processes” on page 402.

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.

PIN keys and 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 all of a single-length key, are multiply-enciphered using the exclusive-OR of the encrypting key and the control vector. The second half (right 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 and the process for encrypting a CCA DES key.

Checking a control vector before processing a cryptographic command

Before a CCA 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 (key-usage) 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 4 shows the flow of cryptographic command processing in a cryptographic facility.

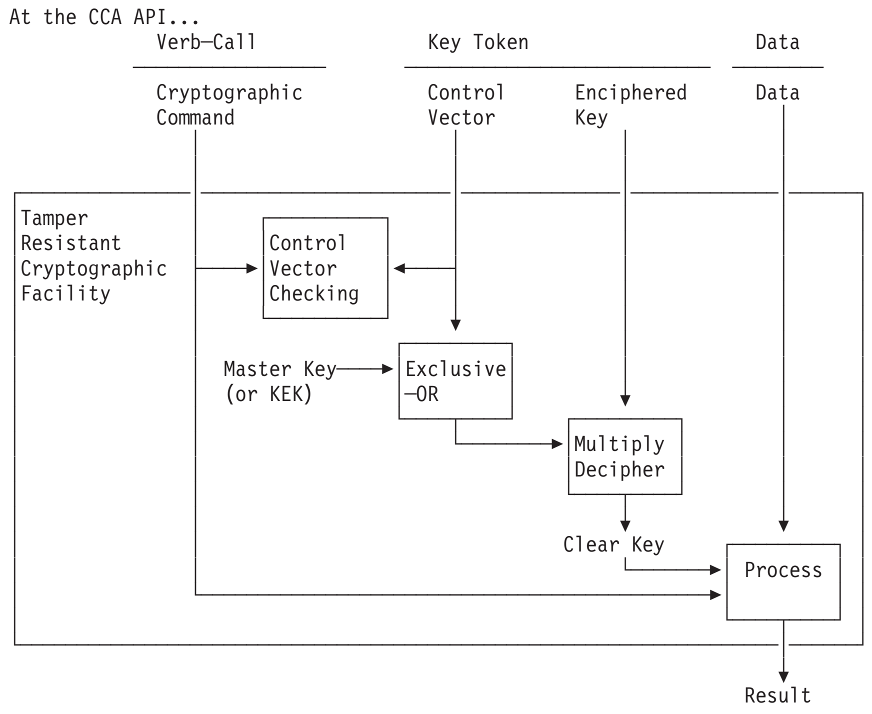


Figure 4. Flow of cryptographic command processing in a cryptographic facility

Key types

The CCA implementation in this product defines DES key-types as shown in Table 10 on page 130. 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:
  - CIPHER and DATA. A key with these key types can be used to both encipher and decipher data.

    Note: 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 for MAC. In this document, DATA means keys with control vector bits 8 to 15 valued to X'00'.
  - MAC. A key with this key type can be used to create a message-authentication code (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.*
  - ENCIPHER and DECIPHER. Used to only encrypt data versus only to decrypt data.
  - MAC and MACVER. Used in generating (and verifying) a MAC versus only verifying a MAC.

- PINGEN and PINVER. Used in generating (and verifying) a personal identification number (PIN) versus only verifying a PIN.
- OPINENC and IPINENC. Used to only encrypt a PIN block versus only to decrypt a PIN block.

Similarly, these unusual key types are paired for other opposite purposes:
- CVARENC and CVARXCVL
- CVARENC and CVARXCVR

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 exports a key.
- IMPORTER or IKEYXLAT. A key with this key type can decipher a key at a node that imports the key.

An EXPORTER key is paired with an IMPORTER or an IKEYXLAT key. An IMPORTER key is paired with an EXPORTER or an OKEYXLAT key. These key types permit the establishment of a unidirectional key-distribution channel which is important both to preserve the asymmetric capabilities possible with CCA-architecture 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 139.

Depending on the key type, a key can be single or double in length. 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 quantity. 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 therefore has the strength of a single-length key. See Table 10 on page 130.

Some verbs can create a default control-vector for a 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 and subtype, a control vector contains key-usage values that further restrict the use of a key. Most key types define a default set of key-usage restrictions in a control vector. See Table 75 on page 387. Key-usage restrictions can be varied by using keywords when constructing control-vector values using the Key_Token_Build verb or the Control_Vector_Generate verb, or by manually setting bits in the control vector.

Figure 5 on page 132 shows the key-type, key subtype, and key-usage keywords that can be combined in the Control_Vector_Generate verb and the Key_Token_Build verb to build a control vector. The left column lists the key types, the middle column lists the subtype keywords, and the right column lists the key-usage keywords that further define a control vector. Table 11 on page 133 describes the control-vector-usage keywords.

For information about the control vector bits, see Appendix C, “CCA control-vector definitions and key encryption.”

Table 10. Key types and verb usage

Key type	Usable with verbs
Cipher Class (Data Operation Keys) These keys are used to cipher text. In operational form and in external form, these keys are associated with a control vector.	
CIPHER	Encipher, Decipher
ENCIPHER	Encipher
DECIPHER	Decipher
MAC Class (Data Operation Keys) These keys are used to generate and verify a message-authentication code (MAC). 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. 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
DATAC	Encipher, Decipher
DATAM	MAC_Generate, MAC_Verify
DATAMV	MAC_Verify
Secure Messaging Class (Data Operation Keys) These keys are used to encrypt keys or PINs. They are double-length keys. In operational form and in external form, these keys are associated with a control vector.	
SECMSG	Diversified_Key_Generate **Note:** This key-type is added in release 2.30 in anticipation of additional verbs that employ the key type in a future release.
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 keys are associated with a control vector.	
EXPORTER	Data_Key_Export, Key_Export, Key_Generate, Key_Translate, Control_Vector_Translate
IMPORTER	Data_Key_Import, Key_Import, Key_Generate, Key_Translate, Control_Vector_Translate, Secure_Key_Import
IKEYXLAT, OKEYXLAT	Key_Translate
PIN Class These keys are used in the various financial-PIN processing commands. They are double-length keys. In operational form and in external form, these keys are associated with a control vector.	
PINGEN	Clear_PIN_Generate, Clear_PIN_Generate_Alternate, Encrypted_PIN_Generate, Encrypted_PIN_Generate_Alternate, Encrypted_PIN_Verify
PINVER	Encrypted_PIN_Verify
IPINENC	Clear_PIN_Generate_Alternate, Encrypted_PIN_Translate, Encrypted_PIN_Verify

Table 10. Key types and verb usage (continued)

Key type	Usable with verbs
OPINENC	Clear_PIN_Encrypt, Encrypted_PIN_Generate, Encrypted_PIN_Translate
Key-Generating-Key Class These keys are used to derive keys. They are double-length keys.	
KEYGENKY	Diversified_Key_Generate, Encrypted_PIN_Translate, Encrypted_PIN_Verify
DKYGENKY	Diversified_Key_Generate
Cryptographic Variable Class These keys are used in the special verbs that operate with cryptographic variables and are single-length keys. In operational form and in external form, these keys are associated with a control vector.	
CVARENC	Cryptographic_Variable_Encipher
CVARXCVL	Control_Vector_Translate
CVARXCVR	Control_Vector_Translate

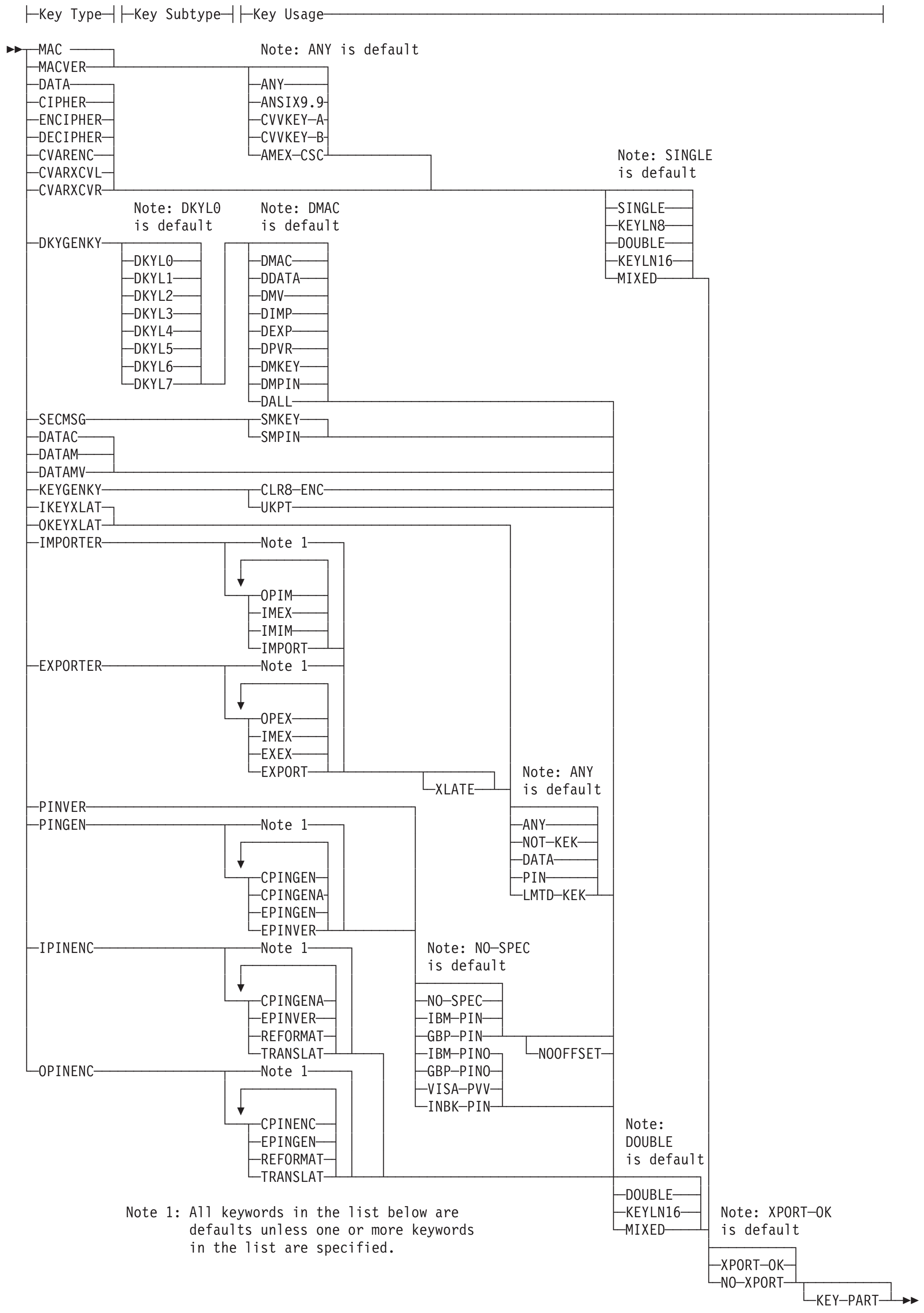


Figure 5. Control_Vector_Generate and Key_Token_Build CV keyword combinations

Table 11. Control vector key-subtype and key-usage keywords

Keyword	**Meaning**
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	IMPORTER keys that have a control vector with this attribute can be used to import a key in the Key_Import verb.
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	EXPORTER 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 keys that have a control vector with this attribute can be used in the Key_Translate verb.
ANY	Key-encrypting keys that have a control vector with this attribute can be used to transport any type of key.
NOT-KEK	Key-encrypting keys that have a control vector with this attribute cannot be used to transport key-encrypting keys.
DATA	Key-encrypting keys that have a control vector with this attribute can be used to transport keys with a key type of DATA, CIPHER, ENCIPHER, DECIPHER, MAC, and MACVER.
PIN	Key-encrypting keys that have a control vector with this attribute can be used to transport keys with a key type of PINVER, IPINENC, and OPINENC. **Note:** The PINGEN key cannot be transported by this type of KEK.
LMTD-KEK	Key-encrypting keys that have a control vector with this attribute can be used to exchange keys with key-encrypting keys that carry NOT-KEK, PIN, or DATA key-type ciphering restrictions.
Data operation keys	
SMKEY	Enable the encryption of keys in an EMV secure message.
SMPIN	Enable the encryption of PINs in an EMV secure message
PIN keys	
NO-SPEC	The control vector does not require a specific PIN-calculation method.
IBM-PIN	Select the IBM 3624 PIN-calculation method.
IBM-PINO	Select the IBM 3624 PIN-calculation method with offset processing.
GBP-PIN	Select the IBM German Bank Pool PIN-calculation method.
GBP-PINO	Select the IBM German Bank Pool PIN-calculation method with institution-PIN input or output.
VISA-PVV	Select the VISA-PVV PIN-calculation method.
INBK-PIN	Select the Interbank PIN-calculation method.

Table 11. Control vector key-subtype and key-usage keywords (continued)

Keyword	Meaning
NOOFFSET	Indicates that a PINGEN or PINVER key cannot participate in the generation or verification of a PIN when an offset or the VISA-PVV process is requested.
CPINGEN	The key can participate in the Clear_PIN_Generate verb.
CPINGENA	The key can participate in the Clear_PIN_Generate_Alternate verb.
EPINGEN	The key can participate in the Encrypted_PIN_Generate verb.
EPINVER	The key can participate in the Encrypted_PIN_Verify verb.
CPINENC	The key can participate in the Clear_PIN_Encrypt verb.
REFORMAT	The key can participate in the Encrypted_PIN_Translate verb in the Reformat mode.
TRANSLAT	The key can participate in the Encrypted_PIN_Translate verb in the Translate mode.
Key-generating keys	
CLR8-ENC	The key can be used to multiply encrypt eight bytes of clear data with a generating key.
DALL	The key can be used to generate keys with the following key types: DATA, DATAC, DATAM, DATAMV, DMKEY, DMPIN, EXPORTER, IKEYXLAT, IMPORTER, MAC, MACVER, OKEYXLAT, and PINVER.
DDATA	The key can be used to generate a single-length or double-length DATA or DATAC key.
DEXP	The key can be used to generate an EXPORTER or an OKEYXLAT key.
DIMP	The key can be used to generate an IMPORTER or an IKEYXLAT key.
DMAC	The key can be used to generate a MAC or DATAM key.
DMKEY	The key can be used to generate a SECMSG with SMKEY secure messaging key for encrypting keys.
DMPIN	The key can be used to generate a SECMSG with SMPIN secure messaging key for encrypting PINs.
DMV	The key can be used to generate a MACVER or DATAMV key.
DPVR	The key can be used to generate a PINVER key.
DKYL0	A DKYGENKY key with this subtype can be used to generate a key based on the key-usage bits.
DKYL1	A DKYGENKY key with this subtype can be used to generate a DKYGENKY key with a subtype of DKYL0.
DKYL2	A DKYGENKY key with this subtype can be used to generate a DKYGENKY key with a subtype of DKYL1.
DKYL3	A DKYGENKY key with this subtype can be used to generate a DKYGENKY key with a subtype of DKYL2.
DKYL4	A DKYGENKY key with this subtype can be used to generate a DKYGENKY key with a subtype of DKYL3.
DKYL5	A DKYGENKY key with this subtype can be used to generate a DKYGENKY key with a subtype of DKYL4.
DKYL6	A DKYGENKY key with this subtype can be used to generate a DKYGENKY key with a subtype of DKYL5.

Table 11. Control vector key-subtype and key-usage keywords (continued)

Keyword	Meaning
DKYL7	A DKYGENKY key with this subtype can be used to generate a DKYGENKY key with a subtype of DKYL6.
Key lengths	
MIXED	Indicates that the key can be either a replicated single-length key or a double-length key with two different, random 8-byte values.
SINGLE, **KEYLN8**	Specifies the key as a single-length key.
DOUBLE, **KEYLN16**	Specifies the key as a double-length key.
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 generally permits an application to provide either a key token or a key label, in which case the parameter description is designated a *key identifier*. Key tokens, key labels, and key 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 DES key-token is a 64-byte data structure that can contain the key and other information frequently needed with the key.

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

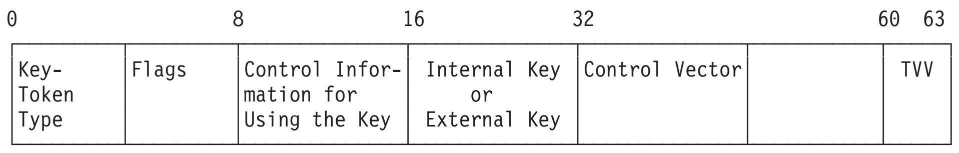

0
8
16
32
60
63
Key-Token Type
Flags
Control Information for Using the Key
Internal Key or External Key
Control Vector
TVV

Figure 6. Key-token contents

- Miscellaneous control information: token type (null, internal, or external), token version layout, and other 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 provides information about the permitted uses of the key.
- 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 or use the Key_Token_Parse verb to disassemble a key token. You can also use application code to assemble or disassemble a key token. You should keep in mind, however, that the contents and format of key tokens are version and implementation sensitive. Key-token formats are described in Appendix B, “Data structures,” on page 347.

The cryptographic system uses key labels and external, internal, and null key-tokens, as shown in Figure 7.

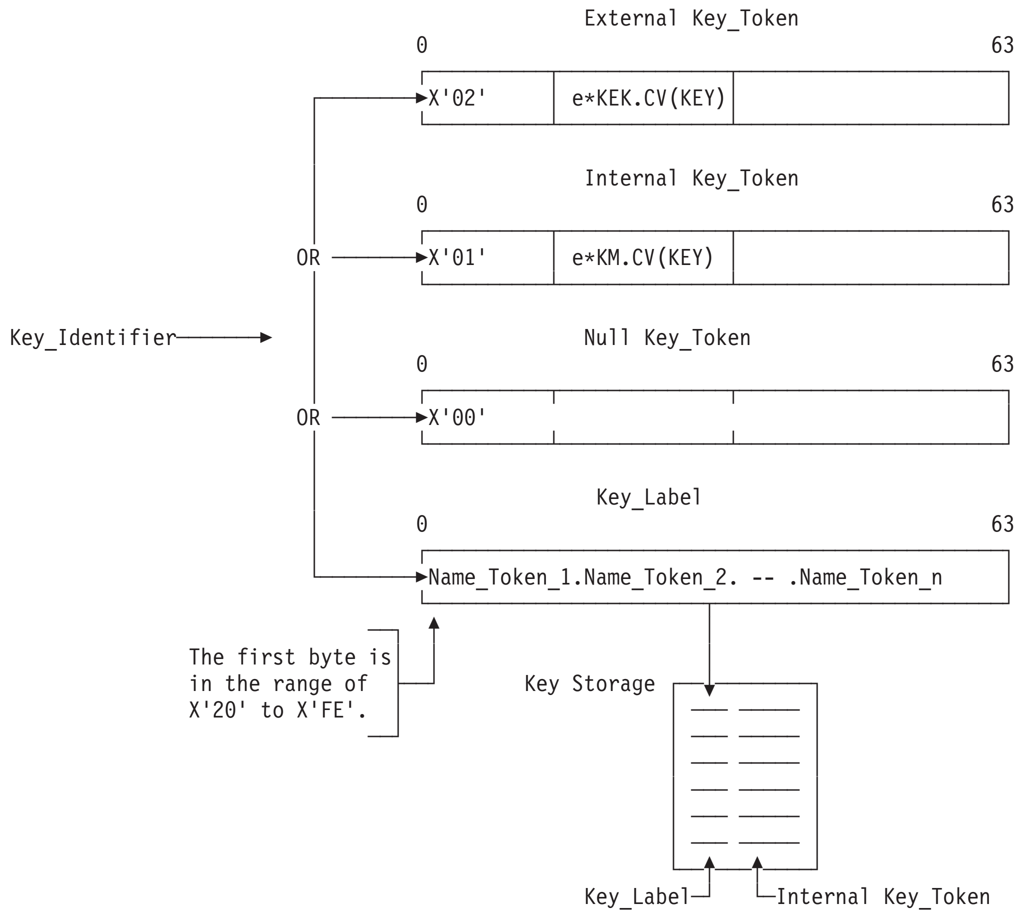


Figure 7. Use of key tokens and key labels

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 storage. An application program can obtain an external key-token by calling one of the following verbs:

- Control_Vector_Translate
- Data_Key_Export
- Key_Export
- Key_Generate
- Key_Token_Build
- Key_Translate

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 symmetric 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
- Diversified_Key_Generate
- Key_Generate
- Key_Import
- Key_Part_Import
- Key_Record_Read
- Key_Token_Build
- Prohibit_Export
- Secure_Key_Import
- Symmetric_Key_Import

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 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 242.

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.

Key-processing and key-storage verbs

Figure 8 on page 139 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 Multiple_Clear_Key_Import, Diversified_Key_Generate, Key_Generate, Key_Part_Import, Clear_Key_Import, and Random_Number_Generate verbs.

CCA subsystems do not reveal the clear value of enciphered keys, and do provide significant control over encrypted keys. Simple key-distribution is addressed by the Cryptographic Node Management (CNM) 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, Clear_Key_Import, Multiple_Clear_Key_Import, and Key_Test verbs allow you to install keys 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 Cryptographic Node Management utility to enter a key part from a keyboard or diskette.

The key parts are single or double in 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 is stored outside the cryptographic facility enciphered under the symmetric master-key. When all the key parts are accumulated, the key-part bit is turned off in the key's control vector.

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 role for the Load First (Master) Key Part command or the Load and Combine (Master) Key Part command.

You can use the Key_Test and Key_Test_Extended verbs to generate a verification pattern.

Note: Starting with Release 3.10, the Key_Test_Extended verb allows you to also operate on an external key.

The verification pattern can then be used to determine the equivalence of another key or a key part. An application program can use the Key_Test and Key_Test_Extended verbs to verify the contents of an enciphered key, or an enciphered key-part. The CNM utility also includes services to generate and use key and key-part verification patterns. With Release 3.10, the Key_Test and Key_Test_Extended verbs optionally allow you to adjust the parity bits of a DES key prior to computing or verifying the key-test pattern.

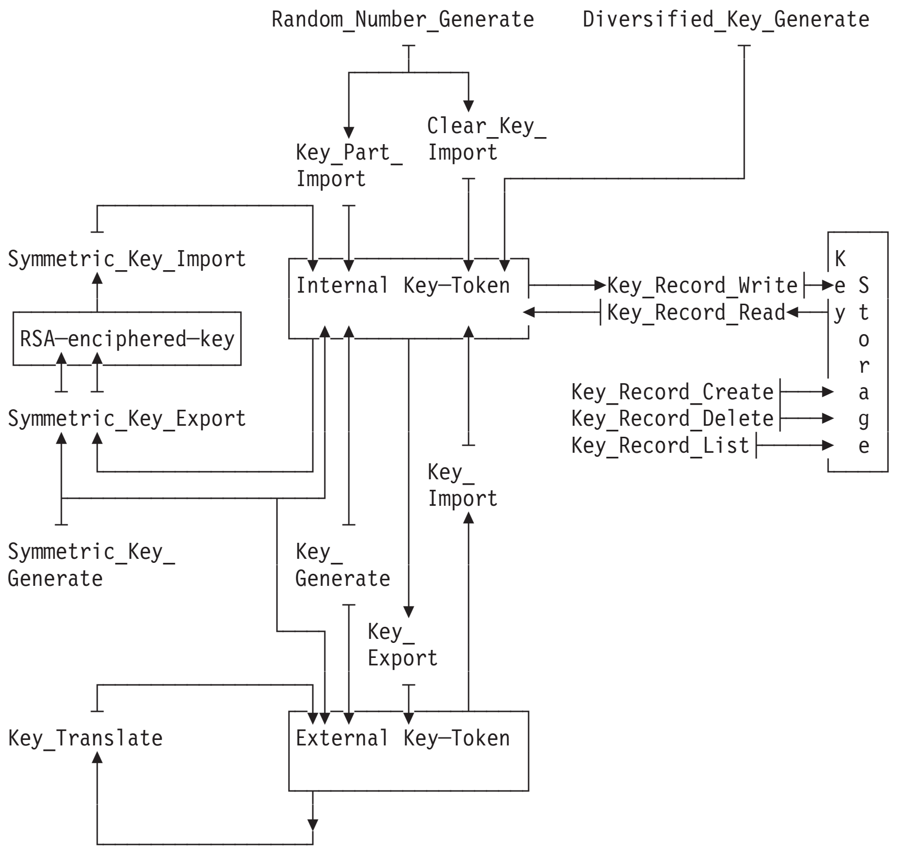


Figure 8. Key-processing verbs

Though you do not know the value of the key or the key part, you can test a key, a key part, or the contents of a master-key register to ensure it has a correct value. You can provide the verification information to the individual who loads the key parts 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 not 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.

Loading of key parts into the coprocessor with the Master_Key_Process and Key_Part_Import verbs or the CNM utility exposes the key parts to potential copying by unauthorized processes. If you are concerned by this exposure, you should randomly generate master keys within the coprocessor, or you should consider distribution of other keys using public key cryptographic techniques.

Generating keys

A CCA cryptographic facility can generate clear keys, key parts, and multiply-enciphered keys or pairs of keys. These keys are generated as follows:

- To generate a clear key, use the odd-parity mode of the Random_Number_Generate verb.
- To generate a key part, use the odd-parity mode of the Random_Number_Generate verb. for the first part, and use the even-parity mode for subsequent key parts. 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 multiply-enciphers 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.

Note: Keys can also be diversified from key-generating keys. See “Diversifying keys” on page 142 for more information.

Before generating a key, consider how the key is to be archived and recovered if unexpected events occur. Before using the Key_Generate verb, 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 a key token with your own updated control-vector that contains non-default restrictions.

  If you update a key token, first use the Control_Vector_Generate and Key_Token_Build verbs to create the control vector and the key token, then use the Key_Generate verb to generate the key.
- Where and when the key is to 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 symmetric 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 is to 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 symmetric 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 is to 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 is imported under the paired IMPORTER 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 is interchanged with a non-CCA system. Use the **DOUBLE** keyword for a double-length key. Because 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 values.

Exporting and importing keys, symmetric techniques

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. (You can also use an RSA asymmetric key as a transport key, see “Exporting and importing keys, asymmetric techniques” on page 142.)

A key that is enciphered under a key-encrypting key other than the symmetric 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 symmetric 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 9 on page 142 shows both the key-export and key-import operations. Data operation keys, PIN 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 385.

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. Users can use the Data_Key_Export verb and the Data_Key_Import verb to export and import DATA keys. These verbs do not import and export key-encrypting keys and PIN keys.

The key-encipherment processes are described in detail at “Understanding CCA key encryption and decryption processes” on page 402.

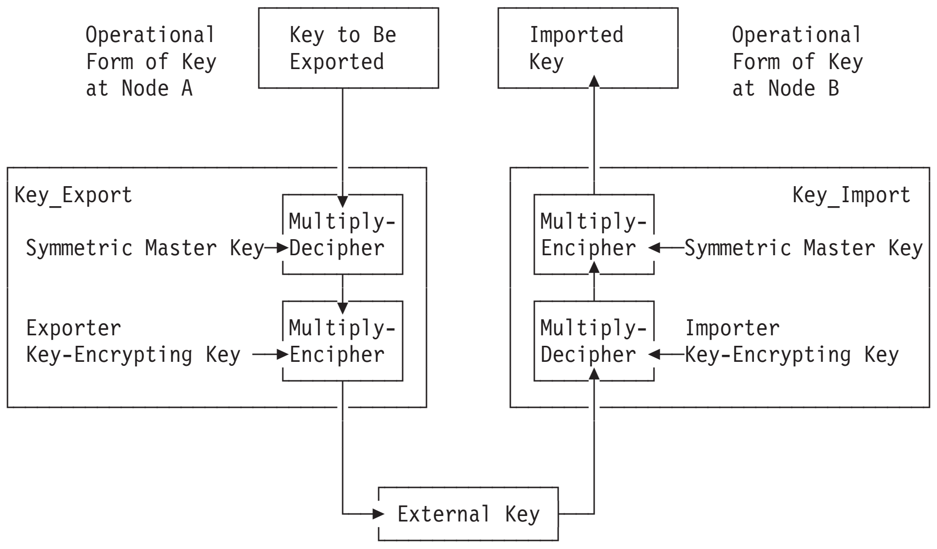


Figure 9. Key exporting and importing

Exporting and importing keys, asymmetric techniques

You can also distribute a DES key from one node to another node by “wrapping” (encrypting) the DES key in the public key of the receiver (IMPORTER). CCA provides two services for wrapping the DES key in the public key of the recipient:

- PKA_Symmetric_Key_Export
- PKA_Symmetric_Key_Generate

and you use the PKA_Symmetric_Key_Import verb to unwrap the transported key using the recipient’s matching private key.

Several techniques for formatting the key to be distributed are in common use and are supported by the verbs. The verbs support processing of default DATA keys. PKA_Symmetric_Key_Generate and PKA_Symmetric_Key_Import can also be used to exchange a DES key-encrypting-key.

DATA keys can be exchanged with CCA and non-CCA implementations using two methods defined in the RSA PKCS #1 v2.0 standard:

- RSAES-OAEP
- RSAES-PKCS-v1_5

Key-encrypting keys can be exchanged between CCA implementations using the “PKA92” formatting method. PKA92 is an OAEP formatting method.

The formatting methods are discussed in “Formatting hashes and keys in public-key cryptography” on page 425.

Diversifying keys

CCA supports several methods for *diversifying* a key using the Diversified_Key_Generate verb. Key-diversification is a technique often used in working with smart cards. In order to secure interactions with a population of cards, a “key-generating key” is used with some data unique to a card to derive (“diversify”) keys for use with that card. The data is often the card serial number or other quantity stored on the card. The data is often public, and therefore it is very important to handle the key-generating key with a high degree of security lest the interactions with the whole population of cards be placed in jeopardy.

In the current implementation, several methods of diversifying a key are supported: **CLR8-ENC**, **TDES-ENC**, **TDES-DEC**, **SESS-XOR**, **TDES-XOR**, and **TDESEMV2** and **TDESEMV4**. The first two methods triple-encrypt data using the *generating_key* to form the diversified key. The diversified key is then multiply-enciphered by the master key modified by the control vector for the output key. The **TDES-DEC** method is similar except that the data is triple-decrypted.

The **SESS-XOR** method provides a means for modifying an existing DATA, DATAC, MAC, DATAM, or MACVER, DATAMV single- or double-length key. The provided data is exclusive-ORed into the clear value of the key. This form of key diversification is specified by several of the credit card associations.

The **TDES-ENC** and **TDES-DEC** methods permit the production of either another key-generating key, or a final key. Control-vector bits 19 – 22 associated with the key-generating key specify the permissible type of final key. (See DKYGENKY in Figure 24 on page 389.) Control-vector bits 12 – 14 associated with the key-generating key specify if the diversified key is a final key or another in a series of key-generating keys. Bits 12 – 14 specify a counter that is decreased by one each time the Diversified_Key_Generate verb is used to produce another key-generating key. For example, if the key-generating key that you specify has this counter set to B'010', then you must specify the control vector for the *generated_key* with a DKYGENKY key type having the counter bits set to B'001' and specifying the same final key type in bits 19 – 22. Use of a *generating_key* with bits 12 – 14 set to B'000' results in the creation of the final key. Thus you can control both the number of diversifications required to reach a final key, and you can closely control the type of the final key.

The **TDESEMV2**, **TDESEMV4**, and **TDES-XOR** methods also derive a key by encrypting supplied data including a transaction counter value received from an EMV smart card. The processes are described in detail at “Visa and EMV-related smart card formats and processes” on page 437. Refer to “Working with Europay–MasterCard–Visa smart cards” on page 276 to understand the various verbs you can use to operate with EMV smart cards.

Storing keys in key storage

Only internal key-tokens 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, are described in Chapter 7, “Key-storage mechanisms.”

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

Security precautions

Be sure to refer to Appendix H, “Observations on secure operations,” on page 449.

In order to maintain a secure cryptographic environment, each cryptographic node must be audited on 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 both 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 (command offset X'00C3 should be avoided.
- The Key_Part_Import verb can be used to enter 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. Note, however, that the clear key-parts have passed through the general purpose computer. Consider accumulating the parts on different machines or using public-key cryptography in the key-distribution scheme.
- Be careful that the public key used in the PKA_Symmetric_Key_Generate and PKA_Symmetric_Key_Export verbs is associated with a legitimate receiver of the exported keys.

CCA DES key-management verbs

Clear_Key_Import (CSNBCKI)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The Clear_Key_Import verb enciphers a clear, single-length DES key under a symmetric 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.

See also the Multiple_Clear_Key_Import verb in "Multiple_Clear_Key_Import (CSNBCKM)" on page 202.

Restrictions

None

Format

CSNBCKI			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
clear_key	Input	String	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 9.

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 symmetric 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 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 preexisting 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 active role.

Control_Vector_Generate (CSNBCVG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Control_Vector_Generate verb builds a control vector from keywords specified by the key_type and rule_array parameters. For descriptions of the keywords and for valid combinations of these keywords, see Figure 5 on page 132, "Key types" on page 128, and "Key-usage restrictions" on page 129. You might achieve added security by using optional keywords, or in some cases required keywords, supplied in the rule-array variable.

Restrictions

None

Format

CSNBCVG			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_type	Input	String	8 bytes
rule_array_count	Input	Integer	
rule_array	Input	String array	rule_array_count * 8 bytes
reserved	Input	String	null pointer or XL8'00' variable
control_vector	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 9.

key_type

The *key_type* parameter is a pointer to a string variable containing a keyword for the key type. The keyword is 8 bytes in length, left-aligned, and padded on the right with space characters. Supply a keyword from the following list:

CIPHER	DATAMV	MAC
CVARENC	DECIPHER	MACVER
CVARPINE	DKYGENKY	OKEYXLAT
CVARXCVL	ENCIPHER	OPINENC
CVARXCVR	EXPORTER	PINGEN
DATA	IKEYXLAT	PINVER
DATAC	IMPORTER	KEYGENKY[1]
DATAM	IPINENC	SECMSG[2]

For definitions of these keywords, see "Control vectors, key types, and key-usage restrictions" on page 127.

1. **CLR8-ENC** must be coded in the rule array when the **KEYGENKY** key-type is coded.
2. **SMKEY** or **SMPIN** must be coded in the rule array when the **SECMSG** key-type is coded.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. For the valid combinations of keywords for the key type and the rule array, see Figure 5 on page 132. The rule_array keywords are shown below:

ANY	DKYL5	GBP-PINO	NO-XPORT
CLR8-ENC[3]	DKYL6	IBM-PIN	NOT-KEK
CPINENC	DKYL7	IBM-PINO	OPEX
CPINGEN	DMAC	IMEX	OPIM
CPINGENA	DMKEY	IMIM	PIN
DALL	DMPIN	IMPORT	REFORMAT
DATA	DMV	INBK-PIN	SINGLE
DDATA	DOUBLE	KEY-PART	SMKEY
DEXP	DPVR	KEYLN8	SMPIN
DIMP	EPINGEN	KEYLN16	TRANSLAT
DKYL0	EPINGENA	LMTD-KEK	UKPT
DKYL1	EPINVER	MIXED	VISA-PVV
DKYL2	EXEX	NOOFFSET	XLATE
DKYL3	EXPORT	NO-SPEC	XPORT-OK
DKYL4	GBP-PIN		

reserved

This *reserved* parameter is a pointer to a string variable. The parameter must either be a null pointer, or a pointer to a variable of eight bytes of X'00'.

control_vector

The *control_vector* parameter is a pointer to a string variable containing the control vector returned by the verb.

Required commands

None

3. **CLR8-ENC** must be coded when the **KEYGENKY** key-type is coded.

Control_Vector_Translate (CSNBCVT)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Control_Vector_Translate verb changes the control vector used to encipher an external key. See "Changing control vectors with the Control_Vector_Translate verb" on page 398 for additional information about this verb.

Restrictions

None

Format

CSNBCVT

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
KEK_key_identifier	Input	String	64 bytes
source_key_token	Input	String	64 bytes
array_key_left	Input	String	64 bytes
mask_array_left	Input	String	56 bytes
array_key_right	Input	String	64 bytes
mask_array_right	Input	String	56 bytes
rule_array_count	Input	Integer	0, 1, or 2
rule_array	Input	String array	rule_array_count * 8 bytes
target_key_token	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 9.

KEK_key_identifier
The *KEK_key_identifier* parameter is a pointer to a string variable containing an internal key-token or the key label of an internal key-token record containing the key-encrypting key. The control vector in the internal key-token must specify the key type IMPORTER, EXPORTER, IKEYXLAT, or OKEYXLAT.

source_key_token
The *source_key_token* parameter is a pointer to a string variable containing the external key-token with the key and control vector to be processed.

array_key_left
The *array_key_left* parameter is a pointer to a string variable containing an internal key-token or a key label of an internal key-token record that deciphers the left mask-array. The internal key-token must contain a control vector specifying a CVARXCVL key-type.

mask_array_left
The *mask_array_left* parameter is a pointer to a string variable containing the mask array enciphered under the left-array key.

array_key_right

The *array_key_right* parameter is a pointer to a string variable containing an internal key-token or the key label of an internal key-token record that deciphers the right mask-array. The internal key-token must contain a control vector specifying a CVARXCVR key-type.

mask_array_right

The *mask_array_right* parameter is a pointer to a string variable containing the mask array enciphered under the right-array key.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 0, 1, or 2 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Table 12. Control_Vector_Translate rule_array keywords

Keyword	Meaning
Parity adjustment (one, optional)	
ADJUST	Ensures that all target-key bytes have odd parity. This is the default.
NOADJUST	Prevents the parity of the target key from being altered.
Key portion (one, optional)	
LEFT	Causes an 8-byte source key, or the left half of a 16-byte source key, to be processed with the result placed into *both* halves of the target key. This is the default.
RIGHT	Causes the right half of a 16-byte source key to be processed with the result placed into only the right half of the target key. The left half of the target key is unchanged.
BOTH	Causes both halves of a 16-byte source key to be processed with the result placed into corresponding halves of the target key. When you use the **BOTH** keyword, the mask array must be able to validate the translation of both halves.
SINGLE	Causes the left half of the source key to be processed with the result placed into only the left half of the target. The right half of the target key is unchanged.

target_key_token

The *target_key_token* parameter is a pointer to a string variable containing an external key-token with the new control-vector. This key token contains the key halves with the new control-vector.

Required commands

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

Cryptographic_Variable_Encipher (CSNBCVE)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Cryptographic_Variable_Encipher verb uses a CVARENC key to encrypt plaintext to produce ciphertext using the Cipher Block Chaining (CBC) method. The plaintext must be a multiple of 8 bytes in length.

Specify the following to encrypt plaintext:

- An internal key-token or a key label of an internal key-token record that contains the key to be used to encrypt the plaintext with the *c-variable_encrypting_key_identifier* parameter. The control vector in the key token must specify the CVARENC key-type.
- The length of the plaintext, which is the same as the length of the returned ciphertext, with the *text_length* parameter. The plaintext must be a multiple of 8 bytes in length.
- The plaintext with the *plaintext* parameter.
- The initialization vector with the *initialization_vector* parameter.
- A variable for the returned ciphertext with the *ciphertext* parameter. The length of this field is the length that you specified with the text_length variable.

The verb does the following:

- Uses the CVARENC key and the initialization value with the CBC method to encrypt the plaintext.
- Returns the encrypted plaintext in the variable pointed to by the *ciphertext* parameter.

Restrictions

- The text length must be a multiple of eight bytes.
- The minimum length of text that the security server can process is 8 bytes and the maximum is 256 bytes.

Format

CSNBCVE			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
c-variable_encrypting_key_identifier	Input	String	64 bytes
text_length	Input	Integer	
plaintext	Input	String	text_length bytes
initialization_vector	Input	String	8 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 9.

c-variable_encrypting_key_identifier
The *c-variable_encrypting_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 internal key-token must contain a control vector that specifies a CVARENC key-type.

text_length
The *text_length* parameter is a pointer to an integer variable containing the length of the plaintext variable and the ciphertext variable.

plaintext
The *plaintext* parameter is a pointer to is a string variable containing the plaintext to be encrypted.

initialization_vector
The *initialization_vector* parameter is a pointer to a string variable containing the 8-byte initialization vector the verb uses in encrypting the plaintext.

ciphertext
The *ciphertext* parameter is a pointer to a string variable containing the ciphertext returned by the verb.

Required commands

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

Data_Key_Export (CSNBDKX)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Data_Key_Export verb exports a single-length or double-length internal DATA-key. The verb can export the key from an internal key-token in key storage or application storage. This verb, which is authorized with a different control point than used with the Key_Export verb, allows you to limit the export operations to DATA keys as compared to the capabilities of the more general verb.

The verb overwrites the 64-byte target-key-token variable 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 is enciphered without any control vector (that is, an all-zero control vector), otherwise the source-key control vector is also used with the target key.

A key with a default, double-length DATA control-vector is exported into a version X'01' external key-token. Otherwise, keys are exported into version X'00' key tokens.

Restrictions

Starting with Release 2.41, unless you enable the Unrestrict Data Key Export command (offset X'0277'), having replicated key-halves is not permitted to export a key having unequal key-halves. Key parity bits are ignored.

Format

CSNBDKX			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

source_key_identifier
The *source_key_identifier* parameter is a pointer to a string variable containing the internal key-token or the key label of the internal key-token to be exported. Only a DATA key can be exported.

exporter_key_identifier
The *exporter_key_identifier* parameter is a pointer to a string variable containing the (EXPORTER) transport key-token or the key label of the (EXPORTER) transport key-token used to encipher the target key.

target_key_token

The *target_key_token* parameter is a pointer to a string variable containing the reencrypted source-key token. Any existing information in this variable is overwritten.

Required commands

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

By also specifying the Unrestrict Data Key Export command (offset X'0277'), you can permit a less secure mode of operation that enables an equal key-halves EXPORTER key-encrypting-key to export a key having unequal key-halves (key parity bits are ignored).

Data_Key_Import (CSNBDKM)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The Data_Key_Import verb imports an encrypted, source DES single-length or double-length DATA key and creates or updates a target internal key-token with the master-key-enciphered source key. The verb can import the key into an internal key-token in application storage or in key storage. This verb, which is authorized with a different control point than used with the Key_Import verb, allows you to limit the import operations to DATA keys as compared to the capabilities of the more general verb.

Specify the following:

source_key_token:
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. A double-length key that should result in a default DATA control vector must be specified in a version X'01' external key-token. Otherwise, both single-length and double-length keys are presented in a version X'00' key token.

Alternatively, you can provide the encrypted DATA-key at offset 16 in an otherwise all X'00' key-token. The verb processes this token format as a DATA key encrypted by the IMPORTER key and a null (all zero) control vector.

importer_key_identifier:
An IMPORTER key-encrypting-key under which the source key is deciphered.

target_key_identifier:
An internal or null key-token. The internal key-token can be located in application storage or in key storage.

The verb builds the internal key-token as follows:

- Creates a default control-vector for a DATA key-type in the internal key-token, provided the control vector in the external key-token is zero. If the control vector is not zero, the verb copies the control vector from the external key-token into the internal 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 symmetric 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 parity of the source key.

Restrictions

Starting with Release 2.41, unless you enable the Unrestrict Data Key Import command (offset X'027C'), an IMPORTER transport key having replicated key-halves is not permitted to import a key having unequal key-halves. Key parity bits are ignored.

Format

CSNBDKM

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length 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 9.

source_key_token
The *source_key_token* parameter is a pointer to a string variable containing the external key-token to be imported. Only a DATA key can be imported.

importer_key_identifier
The *importer_key_identifier* parameter is a pointer to a string variable containing the (IMPORTER) transport key or the key label of the (IMPORTER) transport key used to decipher the source key.

target_key_identifier
The *target_key_identifier* parameter is a pointer to a 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

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

By also specifying the Unrestrict Data Key Import command (offset X'027C'), you can permit a less secure mode of operation that enables an equal key-halves IMPORTER key-encrypting-key to import a key having unequal key-halves (key parity bits are ignored).

Diversified_Key_Generate (CSNBDKG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Diversified_Key_Generate verb generates a key based on a function of a key-generating key, the process rule, 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.

This verb is especially useful for creating "diversified keys" for operating with finance industry smart cards. Be sure to review "Diversifying keys" on page 142.

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 of the key-generating key determines the type of target key that is generated and, except for the **SESS-XOR** process, restricts the use of this key to the key-diversification process.
- The data and its length used in the diversification 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.
  - For the **CLR8-ENC**, **TDESEMV2**, **TDESEMV4**, and **TDES-XOR** processes, a null token might not be specified
  - For the **TDES-ENC** or **TDES-DEC** processes, a null token might be specified
  - For the **SESS-XOR** process, a null token must be specified

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 appropriate key-type and the specified 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
- Recovers the data-encrypting key and determines that the control vector is appropriate for the specified process
- Decrypts the data as can be required by the specified process
- Generates the key appropriate to the specified process
- Does not adjust the parity of the derived key
- Returns the diversified key, multiply-enciphered by the master key modified by the control vector

Restrictions

The **TDES-XOR** rule-array keyword is available starting with Release 2.50. The **TDESEMV2** and **TDESEMV4** rule-array keywords are available starting with Release 2.51.

Format

CSNBDKG

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
generating_key_identifier	In/Output	String	64 bytes
data_length	Input	Integer	
data	Input	String	data_length bytes
data_decrypting_key_identifier	In/Output	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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Process rule (required)	
CLR8-ENC	Specifies that eight bytes of clear (not encrypted) data shall be triple-DES encrypted with the generating key to create a generated key. The encryption process is like that shown in Figure 28 on page 404 for a single-length key with a control vector valued to binary zero. The key selected by the *generating_key_identifier* must specify a KEYGENKY key-type also with control vector bit 19 set to one. The key identified by the *data_decrypting_key_identifier* must identify a null key-token. The key token identified by the *generated_key_identifier* variable must contain a control vector that specifies a single-length key of one of these types: DATA, CIPHER, ENCIPHER, DECIPHER, MAC, or MACVER.

Keyword	Meaning
TDES-ENC	Specifies that 8 bytes or 16 bytes of clear (not encrypted) data shall be triple-DES *en*crypted with the generating key to create the generated key. If the *generated_key_identifier* variable specifies a single-length key, then 8 bytes of clear data are triple-DES encrypted. If the *generated_key_identifier* variable specifies a double-length key, then 16 bytes of clear data are triple-DES encrypted in ECB mode. The key selected by the *generating_key_identifier* must specify a DKYGENKY key-type that has the appropriate control vector usage bits (bits 19 – 22) set for the desired generated key. Control vector bits 12 – 14 binary encode the key-derivation sequence level (DKYL7 down to DKYL0, see DKYGENKY in Figure 24 on page 389). The final key is derived when bits 12 – 14 are B'000'. The verb verifies the incremental relationship between the value in *generated_key_identifier* control vector and the *generating_key_identifier* control vector. Or in the case when the *generated_key_identifier* is a null token, the appropriate counter value is placed into the output key-token. The *data_decrypting_key_identifier* must identify a null key-token. A key token identified by the *generated_key_identifier* variable that is not a null key-token must contain a control vector that specifies a single-length or double-length key having a key type consistent with the specification in bits 19 – 22 of the generating key.

Keyword	Meaning
TDES-DEC	Specifies that 8 or 16 bytes of clear (not encrypted) data shall be triple-DES *de*crypted with the generating key to create the generated key. If the *generated_key_identifier* variable specifies a single-length key, then 8 bytes of clear data are triple-DES decrypted. If the *generated_key_identifier* variable specifies a double-length key, then 16 bytes of clear data are triple-DES decrypted in ECB mode. The key selected by the *generating_key_identifier* must specify a DKYGENKY key-type that has the appropriate control vector usage bits (bits 19 – 22) set for the desired generated key. Control vector bits 12 – 14 binary encode the key-derivation sequence level (DKYL7 down to DKYL0, see DKYGENKY in Figure 24 on page 389). The final key is derived when bits 12 – 14 are B'000'. The verb verifies the incremental relationship between the value in *generated_key_identifier* control vector and the *generating_key_identifier* control vector. Or in the case when the *generated_key_identifier* is a null-token, the appropriate counter value is placed into the output key-token. The *data_decrypting_key_identifier* must identify a null key-token. A key token identified by the *generated_key_identifier* variable that is not a null key-token must contain a control vector that specifies a single-length or double-length key having a key type consistent with the specification in bits 19 – 22 of the generating-key.

Keyword	Meaning
TDESEMV2, **TDESEMV4**	**Note:** These options are available starting with Release 2.51. Specifies that 10 bytes, 18 bytes, 26 bytes, or 34 bytes of clear data shall be processed to form an EMV card-unique key and then a session key as specified in the *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2*, Annex A1.3. See “Visa and EMV-related smart card formats and processes” on page 437 for additional details. The supplied *data* variable must contain the concatenation of: • 8 or 16 bytes of data to diversify the issuer-master-key. • 2 bytes containing the Application Transaction Counter (ATC) received from the smart card. Place the counter value in a string construct with the high-order counter bit first in the string. • Optionally, a 16-byte Initial Value used in obtaining the session key from the card-unique key. The key selected by the *generating_key_identifier* parameter must specify a DKYGENKY key-type at level-0 (bits 12 – 14 B'000') and indicate permission to create one of several key types in bits 19 – 22: • B'0001' DDATA, to generate a DATA key • B'0010' DMAC, to generate a MAC key • B'0011' DMV, to generate a MACVER key • B'1000' DMKEY, to generate a SECMSG SMKEY (used in secure messaging, key encryption, see the Secure_Messaging_for_Keys verb) • B'1001' DMPIN, to generate a SECMSG SMPIN (used in secure messaging, PIN encryption, see the Secure_Messaging_for_PINs verb). The *data_decrypting_key_identifier* must identify a null key-token. A key token or key-token record identified by the *generated_key_identifier* parameter that is not a null key-token. The token must contain a control vector that specifies a key type conforming to that specified in control-vector bits 19 – 22 for the key-generating key. The control vector must specify a double-length key.

Keyword	Meaning
TDES-XOR	**Note:** This option is available starting with Release 2.50. Specifies that 10 bytes or 18 bytes of clear (not encrypted) data shall be processed as described in “Visa and EMV-related smart card formats and processes” on page 437 to create the generated key. The data variable contains either 8 bytes or 16 bytes of data to be triple-encrypted to which you append a 2-byte Application Transaction Counter value (previously received from the smart card). Place the counter value in a string construct with the high-order counter bit first in the string. The key selected by the *generating_key_identifier* parameter must specify a DKYGENKY key-type at level-0 (bits 12 – 14 B'000') and indicate permission to create one of several key types in bits 19 – 22: • B'0001' DDATA, to generate a DATA key • B'0010' DMAC, to generate a MAC key • B'0011' DMV, to generate a MACVER key • B'1000' DMKEY, to generate a SECMSG SMKEY (used in secure messaging, key encryption, see the Secure_Messaging_for_Keys verb) • B'1001' DMPIN, to generate a SECMSG SMPIN (used in secure messaging, PIN encryption, see the Secure_Messaging_for_PINs verb) The *data_decrypting_key_identifier* must identify a null key-token. A key token or key-token record identified by the *generated_key_identifier* parameter that is not a null key-token. The token must contain a control vector that specifies a key type conforming to that specified in control-vector bits 19 – 22 for the key-generating key. The control vector must specify a double-length key.
SESS-XOR	Specifies the Visa** method for session-key generation, namely that 8 bytes or 16 bytes of data shall be exclusive-ORed with the clear value of the session key contained in the key token specified by the *generating_key_identifier* parameter. If the *generating_key_identifier* parameter specifies a single-length key, then 8 bytes of data are exclusive-ORed. If the *generating_key_identifier* parameter specifies a double-length key, then 16 bytes of data are exclusive-ORed. The key token specified by the *generating_key_identifier* parameter must be of key type DATA, DATAC, MAC, DATAM, MACVER, or DATAMV. The key identified by the *data_decrypting_key_identifier* must identify a null key-token. On input, the token identified by the *generated_key_identifier* parameter must identify a null key-token. The control vector contained in the output key token identified by the *generated_key_identifier* parameter is the same as the control vector contained in the key token specified by the *generating_key_identifier* parameter.

generating_key_identifier
The *generating_key_identifier* parameter is a pointer to a string variable containing the key-generating-key key-token or key label of a key-token record.

data_length
The *data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the data variable.

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 rule specified in the rule array. Currently this variable must contain clear data.

data_decrypting_key_identifier
The *data_decrypting_key_identifier* parameter is a pointer to a string variable containing the data decrypting key-token or key label of a key-token record. The specified process dictates the class of key. If the process rule does not support encrypted data, point to a null key-token. Currently this variable must contain a 64-byte null token.

generated_key_identifier
The *generated_key_identifier* parameter is a pointer to a string variable containing the target internal key-token or the key label of the target key-token record. Specify either an internal token or a skeleton token containing the desired control vector of the generated key.

- For the **CLR8-ENC**, **TDESEMV2**, **TDESEMV4**, and **TDES-XOR** processes, a null token might not be specified
- For the **TDES-ENC** or **TDES-DEC** processes, a null token might be specified
- For the **SESS-XOR** process, a null token must be specified.

The generated key is encrypted and returned in the specified token. The control vector in the specified internal token must be suitable for the specified process rule.

Required commands

The Diversified_Key_Generate verb requires the following commands to be enabled in the active role based on the keyword specified for the process rule:

Process rule	Command offset	Command
CLR8-ENC	X'0040'	Generate Diversified Key (CLR8-ENC)
SESS-XOR	X'0043'	Generate Diversified Key (SESS-XOR)
TDES-DEC	X'0042'	Generate Diversified Key (TDES-DEC)
TDES-ENC	X'0041'	Generate Diversified Key (TDES-ENC)
TDES-XOR	X'0045'	Generate Diversified Key (TDES-XOR)
TDESEMV2, TDESEMV4	X'0046'	Generate Diversified Key (TDESEMVn)

When a key-generating key of key type DKYGENKY is specified with control vector bits (19 – 22) of B’1111’, the Generate Diversified Key (DALL with DKYGENKY Key Type) command (offset X'0290') must also be enabled in the active role.

When using the **TDES-ENC** or **TDES-DEC** modes, you can specifically enable generation of a single-length key or a double-length key with equal key-halves (an effective single-length key) by enabling the Enable DKG Single Length Keys and Equal Halves for TDES-ENC, TDES-DEC command (offset X'0044').

Key_Encryption_Translate (CSNBKET)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758		2.54	
IBM 4764		3.23	

The Key_Encryption_Translate verb is used to change the method of key encryption. An input key can be a double-length external CCA DATA key, or a double-length CBC-encrypted key. The return key is encrypted using CBC encryption or CCA (ECB) encryption. The CCA DATA key must be double-length and have an all-zero control vector. The CBC-encrypted key is treated as a 16-byte string encrypted with an all-zero initialization vector.

Specify the following:

1. The translation reencryption operation using a rule-array keyword:
   - **CBCTOECB** changes a CBC key-encryption to a CCA (ECB) encryption.
   - **ECBTOCBC** changes a CCA (ECB) key-encryption to CBC encryption.
2. The key-encrypting key:
   - When performing CBCTOECB translation, specify an IMPORTER key.
   - When performing ECBTOCBC translation, specify an EXPORTER key.
3. Using the *key_in* parameter, identify a 64-byte external CCA DATA key-token, or a 16-byte CBC-encrypted key. Set the *key_in_length* variable to the length of the *key_in* variable.
4. Using the *key_out* parameter, identify a 64-bye external CCA DATA key-token with an all-zero control vector, or a 16-byte string. Set the *key_out_length* variable to the length of the *key_out* variable.

The verb performs the following tasks:

- Recovers the key-encrypting key and ensures that the type is consistent with the requested ECBTOCBC or CBCTOECB translation.
- Decrypts the supplied key_in key using the key-encrypting key, and then encrypts the result using the key-encrypting key.
- For CBCTOECB translation, updates the *key_out* variable with the data key in an external token with an all-zero control vector.
- For ECBTOCBC translation, returns the key in a 16-byte string.

Restrictions

None

Format

CSNBKET

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
kek_key_identifier_length	Input	Integer	64
kek_key_identifier	In/Output	String	kek_key_identifier_length bytes

key_in_length	Input	Integer	16 or 64
key_in	Input	String	key_in_length bytes
key_out_length	In/Output	Integer	16 or 64
key_out	Output	String	key_out_length bytes

Parameters

For the definitions of *return_code*, *reason_code*, *exit_data_length*, and *exit_data*, see "Parameters common to all verbs" on page 9.

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

rule_array
The rule_array parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

The rule_array keywords are shown in the following table:

Keyword	Meaning
Key translation method (one required)	
CBCTOECB	Specifies decryption of a 16-byte string and CCA key key-encryption of the resulting clear-key value as an external CCA DATA key.
ECBTOCBC	Specifies decryption of a CCA DATA key and the CBC encryption of the resulting clear key.

kek_identifier_length
The kek_identifier_length parameter is a pointer to an integer variable containing a value of 64, the length of a CCA DES key token.

kek_identifier
The kek_identifier parameter is a pointer to a string variable containing the key-encrypting key key-token, or key label of a key-token record.

key_in_length
The key_in_length parameter points to an integer variable valued to 16 for the CBCTOECD translation, or valued to 64 for the ECBTOCBC translation.

key_in
The key_in parameter points to a string variable containing a CCA external key-token, or a 16–byte CBC-encrypted key.

key_out_length
The key_out_length parameter points to an integer variable. On input, set the variable to:

- At least 16 for the ECBTOCBC translation
- At least 64 for the CBCTOECB translation

Upon successful completion, the verb sets the variable to the length of the returned *key_out* variable.

key_out
The key_out parameter points to a string variable. The verb returns the encrypted key in a CCA external DATA key-token with an all-zero control vector, or in a 16-byte string.

Required commands

The Key-Encryption-Translate verb requires the following commands to be enabled in the active role, based on the keyword specified in the rule array.

- Translate Key from CBC to ECB (offset X'030D') with the **CBCTOECB** keyword.
- Translate Key from ECB to CBC (offset X'030E') with the **ECBTOCBC** keyword.

Key_Export (CSNBKEX)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Key_Export verb exports a source DES internal-key into a target external key-token. Existing information in the target key-token is overwritten. The target key is enciphered by the EXPORTER-key exclusive-ORed with the control vector of the target key.

Specify the following:

key_type
A keyword for the key type. Use of the **TOKEN** keyword is the preferred coding style. For compatibility with older systems, however, you can explicitly name a key type, in which case the key type must match the key in the control vector of the source key-identifier.

source_key_identifier
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.

exporter_key_identifier
An EXPORTER key-encrypting-key under which the target key is enciphered.

target_key_token
A 64-byte variable to hold the target key-token.

The verb builds the external key-token:

- 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 for single-length or double-length keys, in which case the target control vector is set to 0.
- 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 variable identified by the *target_key_token* parameter, ignoring any preexisting data.

Restrictions

Starting with Release 2.41, unless you enable the Unrestrict Reencipher from Master Key command (offset X'0276'), an EXPORTER key-encrypting-key having equal key-halves is not permitted to export a key having unequal key-halves. Key parity bits are ignored.

Format

CSNBKEX			
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

key_type

The *key_type* parameter is a pointer to a string variable containing a keyword that specifies the key type of the source key-token. The keyword is 8 bytes in length, and must be left-aligned and padded on the right with space characters. The key_type keywords are shown below:

CIPHER	IMPORTER	PINGEN
DATA	IPINENC	PINVER
DECIPHER	MAC	TOKEN
ENCIPHER	MACVER	DATAC (S/390 only)
EXPORTER	OKEYXLAT	DATAM (S/390 only)
IKEYXLAT	OPINENC	DATAMV (S/390 only)

source_key_identifier

| The *source_key_identifier* parameter is a pointer to a string variable containing
| the DES internal source key-token or key label of a key-token record.

exporter_key_identifier

The *exporter_key_identifier* parameter is a pointer to a string variable containing the EXPORTER key-encrypting-key token or key label of a key-token record.

target_key_token

| The *target_key_token* parameter is a pointer to a string variable containing the
| DES external target key-token.

Required commands

The Key_Export verb requires the Reencipher from Master Key command (offset X'0013') to be enabled in the active role.

By also specifying the Unrestrict Reencipher from Master Key command (offset X'0276'), you can permit a less secure mode of operation that enables an equal key-halves EXPORTER key-encrypting-key to export a key having unequal key-halves (key parity bits are ignored).

Key_Generate (CSNBKGN)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 preexisting control-vector from the key token. Use of the **TOKEN** keyword allows you to associate other than default control vectors with the generated keys. Use of the **TOKEN** keyword is the preferred coding style.

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, 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 application storage or to 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

None

Format

CSNBKGN

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

key_form

The *key_form* parameter is a pointer to a string variable containing the keyword that defines whether one or two copies of the key is generated, and the type of key-encrypting key used to encipher the key. The keyword is 4 characters in length and must be left-aligned and padded on the right with space characters.

- 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.

Specify one of the following keywords for the key_form variable:

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 is available to the local node.

key_length

The *key_length* parameter is a pointer to an 8-byte string variable, left-aligned, and padded on the right with space characters, that contains the length of the new key or keys. 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. The key_length variable must contain one of the following:

SINGLE or **KEYLN8**
For a single-length key. See "Key-length specification" on page 174.

SINGLE-R
For a double-length key with equal-valued halves ("single replicated")

DOUBLE or **KEYLN16**
For a double-length key. See "Key-length specification" on page 174.

Note: Some CCA implementations might support the keyword **DOUBLE-O** to enable generation of double-length keys with key-halves guaranteed to be unique. The associated key-form control vector bits (bits 40-42) B'110' are described in "Key-form bits, 'fff' and 'FFF'" on page 391. This implementation does not support the **DOUBLE-O** keyword, but this implementation does support generation of guaranteed unique-key-halves if you supply a key token with a control vector having form-field bits of B'110'. Support of form-field B'110' is not available in all CCA implementations.

The key halves are different except when the same 56-bit key would be generated twice in succession — a minuscule possibility.

8 spaces
When you provide a control vector, or when you wish the verb to select the key length based on the key type, provide eight space characters to direct the verb to select the key length.

key_type_1 and key_type_2

The *key_type_1* and *key_type_2* parameters are pointers to 8-byte string variables, each containing a keyword that specifies the key type for each new key being generated. To specify the key type via the control vector in the preexisting key-token, use the **TOKEN** keyword. Alternatively, you can specify the key type using keywords shown in Table 13 on page 173 and Table 14 on page 173. This is useful when you want to create default-value key-tokens and control-vectors.

- Table 13 on page 173 lists the keywords allowed when generating a single key copy (*key_form* OP, IM, or EX). *Key_type_2* should contain a string of eight space characters.
- Table 14 on page 173 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 of a key-token record for the key used to encipher the IM-form and EX-form keys. If an OP-form key is requested, the associated KEK identifier must 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 a key-token record 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 must 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 the Key_Form OP column in Table 13 on page 173, the Key_Generate verb requires the Generate Key command (offset X'008E') to be enabled in the active role.
- If you specify the key-form and key-type combinations shown with an X in the Key_Form IM column in Table 13 on page 173, the Key_Generate verb requires the Generate Key Set command (offset X'008C') to be enabled in the active role. The verb applies the restrictive rules of the IMEX column in Table 14 on page 173 to the generation of the IM form key.
- If you specify the key-form and key-type combinations shown with an X in the Key_Form EX column in Table 13 on page 173, the Key_Generate verb requires the Generate Key Set command (offset X'008C') to be enabled in the active role. The verb applies the restrictive rules of the EXEX column in Table 14 on page 173 to the generation of the EX form key.
- If you specify the key-form and key-type combinations shown with an X in Table 14 on page 173, the Key_Generate verb requires the Generate Key Set command (offset X'008C') to be enabled in the active role.
- If you specify the key-form and key-type combinations shown with an E in Table 14 on page 173, the Key_Generate verb requires the Generate Key Set Extended command (offset X'00D7') to be enabled in the active role.
- If you specify the **SINGLE-R** key-length keyword, the Key_Generate verb also requires the Replicate Key command (offset X'00DB') to be enabled in the active role.

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. (See “Understanding CCA key encryption and decryption processes” on page 402.)

There are two methods for specifying the type of keys to be generated:
- Specify key-type keywords from Table 13 on page 173 or Table 14 on page 173
- Use the **TOKEN** keyword and encode the key type and other information in the control vector you provide in the generated_key_identifier_n key-token variables

Use of the key-type keywords generates default control vector values. See Table 75 on page 387. One or two keywords are examined based on the *key_form* variable. Table 13 on page 173 shows the key-type keywords you can use to generate a single key copy with default control-vectors. Table 14 on page 173 shows the key types you can use to 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) command is required as those keys require special handling.

You can generate a single-length key with any control vector value[4]. when you specify **SINGLE** and **OP**. In this case, the verb uses the Generate Key command (X'008E')

If you encode the key type in a control vector supplied in a key token (and use the **TOKEN** key-type keyword), remember that non-default control vector values for the key type can be employed.

Certain key-type keywords have an asterisk (*) indicating that these *keywords* are not recognized by the verb as key type specifications. Nevertheless, those key types are supported when supplied as control vector values.

Table 13. 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
DATAC * DATAM * DATAMV * KEYGENKY * DKYGENKY * SECMSG *	X	X	X

Notes:

1. The key types marked with an * must be requested through the specification of a proper control vector in a key token and the use of the **TOKEN** keyword.
2. Additional key types can be generated as operational keys when you supply key form as **OP**, key type as **TOKEN**, key length as eight space characters, and provide the desired control vector in the key token specified by the *generated_key_identifier_1* parameter.

Table 14. 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**
DATA MAC MAC MACVER DATAC * DATAM * DATAM * CIPHER CIPHER CIPHER DECIPHER DECIPHER ENCIPHER ENCIPHER KEYGENKY * DKYGENKY *	DATA MAC MACVER MAC DATAC * DATAM * DATAMV * CIPHER DECIPHER ENCIPHER CIPHER ENCIPHER CIPHER DECIPHER KEYGENKY * DKYGENKY *	X	X	X	X

4. The command-level architecture permits many CV values and value-pairs to be generated so long as they adhere to rules defined in that architecture. It is beyond the scope of this publication to explain all permissible combinations. Only those with defined usage are shown in the tables.

Table 14. Key_type and key_form keywords for a key pair (continued)

key_type_1	key_type_2	key_form OPOP, OPIM, IMIM	key_form OPEX	key_form EXEX	key_form IMEX
EXPORTER IMPORTER EXPORTER IKEYXLAT IKEYXLAT IMPORTER OKEYXLAT OKEYXLAT PINGEN PINVER	IMPORTER EXPORTER IKEYXLAT EXPORTER OKEYXLAT OKEYXLAT IMPORTER IKEYXLAT PINVER PINGEN		X	X	X
OPINENC IPINENC	IPINENC OPINENC	E	X	X	X
OPINENC	OPINENC	X			
CVARDEC * CVARENC * CVARENC * CVARENC * CVARXCVL * CVARXCVR * CVARDEC * CVARPINE *	CVARENC * CVARDEC * CVARXCVL * CVARXCVR * CVARENC * CVARENC * CVARPINE * CVARDEC *		E		E
Note: The key types marked with an * must be requested through the specification of a proper control-vector in a key token and the use of the **TOKEN** keyword.					

Key-length specification: The *key_length* parameter points to a variable containing a keyword or eight space characters which specifies the length of a key, either 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**, **DOUBLE**, or eight space characters. The **SINGLE-R** keyword 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.

Table 15 shows the valid key lengths for each key type. An 'X' indicates that a key length is permitted for a key type and a 'D' indicates the default key-length the verb uses when you supply eight space characters with the *key_length* parameter.

Table 15. Key lengths by key type

Key type	SINGLE, KEYLN8	SINGLE-R	DOUBLE, KEYLN16
MAC MACVER	X, D X, D	X X	X X
DATA	X, D	X	X
DATAC * DATAM * DATAMV *	X X X	X X X	X X X
EXPORTER IMPORTER		X X	X, D X, D
IKEYXLAT OKEYXLAT		X X	X, D X, D
CIPHER DECIPHER ENCIPHER	X, D X, D X, D	X X X	X X X

Table 15. Key lengths by key type (continued)

Key type	**SINGLE, KEYLN8**	**SINGLE-R**	**DOUBLE, KEYLN16**
DKYGENKY IPINENC OPINENC PINGEN PINVER		X X X X X	X, D X, D X, D X, D X, D
CVARDEC * CVARENC * CVARPINE * CVARXCVL * CVARXCVR *	X X X X X	X X X X X	X X X X X
KEYGENKY *	X	X	X, D
SECMSG *		X	X, D
Note: The key types marked with an * must be requested through the specification of a proper control-vector in a key token and the use of the **TOKEN** keyword.			

Key_Import (CSNBKIM)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 symmetric master-key.

Specify the following:

key_type
A keyword for the key type. Use of the **TOKEN** keyword is the preferred coding style. 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.

source_key_token
An external key-token or an encrypted external 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. (The first byte of a null key-token is X'00'.)

importer_key_identifier
An IMPORTER key-encrypting-key under which the target key is deciphered.

target_key_identifier
An internal or null key-token, or the key label of an internal or null key-token record 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, nonconflicting 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 other than **TOKEN** 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 in decrypting the source key.
  - The control vector in the source key-token is used in encrypting the source key under the master key. A source key having the default external DATA control vector (8 or 16 bytes of X'00') results in a target key with the default internal DATA control vector.

  The key token is returned to the application or key storage with the imported key.

Restrictions

Starting with Release 2.41, unless you enable the Unrestrict Reencipher to Master Key command (offset X'027B'), an IMPORTER key-encrypting-key having equal key-halves is not permitted to import a key having unequal key-halves. Key parity bits are ignored.

Format

CSNBKIM

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

key_type

The *key_type* parameter is a pointer to a string variable containing an 8-byte keyword, left-aligned, and padded on the right with space characters, that specifies the key type of the key to be imported. In general, you should use the **TOKEN** keyword.

CIPHER	IKEYXLAT	OKEYXLAT
DATA	IMPORTER	OPINENC
DECIPHER	IPINENC	PINGEN
ENCIPHER	MAC	PINVER
EXPORTER	MACVER	TOKEN

source_key_token

| The *source_key_token* parameter is a pointer to a string variable containing the external source DES 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 string variable containing the key-token or key label for the IMPORTER (transport) key-encrypting-key.

target_key_identifier

| The *target_key_identifier* parameter is a pointer to a string variable containing
| the internal target DES key-token or key label of a key-token record.

Required commands

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

By also enabling the Unrestrict Reencipher To Master Key command (offset X'027B'), you can permit a less secure mode of operation that enables an equal key-halves IMPORTER key-encrypting-key to import a key having unequal key-halves (key parity bits are ignored).

Key_Part_Import (CSNBKPI)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The Key_Part_Import verb is used to accumulate parts of a key and store the result as an encrypted partial key or as the final key. Individual key-parts are exclusive-ORed together to form the accumulated key.

On each call to Key_Part_Import (except **COMPLETE**, see below), specify 8 bytes or 16 bytes of clear key-information based on the length of the key that you are accumulating. Align an 8-byte clear key in the high-order bytes (leftmost bytes) of a 16-byte field. Also specify an internal key-token in which the key information is accumulated. The key token must include a control vector. The control vector defines the length of the key, 8 or 16 bytes (single length or double length). The control vector must have the KEY-PART bit set on. The verb returns the accumulated key information as a master-key-encrypted value in the updated key-token.

You can use the Key_Token_Build verb to create the internal key-token into which the first key-part is imported.

On each call to Key_Part_Import, also specify a rule-array keyword to define the verb action: **FIRST**, **MIDDLE**, **LAST**, **ADD-PART**, or **COMPLETE**.

- With the **FIRST** keyword, the verb ignores any key information present in the input key-token. Each byte of the 8- or 16-byte key-part should have the low-order bit set such that the byte has an *odd* number of one-bits, otherwise assuming no other problems, the verb returns reason code 2. Use of the **FIRST** keyword requires that the Load First Key Part command be enabled in the access-control system.
- With the **MIDDLE** keyword, the verb exclusive-ORs the clear key-part with the (internally decrypted) key value from the input key-token. Each byte of the 8- or 16-byte key-part should have the low-order bit set such that the byte has an *even* number of one-bits. If any byte in the updated key has an even number of one bits, and there are no other problems, the verb returns reason code 2. Use of the **MIDDLE** keyword requires that the Combine Key Parts command be enabled in the access-control system. The key-part bit remains on in the control vector of the updated key token returned from the verb.
- With the **LAST** keyword, the verb exclusive-ORs the clear key-part with the (internally decrypted) key value in the input key-token. Each byte of the 8- or 16-byte key-part should have the low-order bit set such that the byte has an *even* number of one-bits. If any byte in the updated key has an even number of one bits, and there are no other problems, the verb returns reason code 2. This use of the **LAST** keyword requires that the Combine Key Parts command be enabled in the access-control system. The key-part bit is set off in the control vector of the updated key token returned from the verb.
- With the **ADD-PART** keyword, the verb exclusive-ORs the clear key-part with the (internally decrypted) key value in the input key-token. Each byte of the 8- or 16-byte key-part should have the low-order bit set such that the byte has an *even* number of one-bits. If any byte in the updated key has an even number of one bits, and there are no other problems, the verb returns reason code 2. Use of the **ADD-PART** keyword requires that the Add Key Part command be enabled in

the access-control system. The key-part bit remains on in the control vector of the updated key token returned from the verb.

- With the **COMPLETE** keyword, the key-part bit is set off in the control vector of the updated key token returned from the verb. Use of the **COMPLETE** keyword requires that the Complete Key Part command be enabled in the access-control system. The 16-byte key_part variable must be declared but is ignored by the coprocessor.

Notes:

1. If your input creates a key value with one or more bytes with an even number of one bits, that is an out-of-parity key, and the verb returns a reason-code value of 2. Many verbs check the parity of keys and, if the key does not have odd parity in each key-byte, might return a warning or might terminate without performing the requested operation. In general, out-of-parity DATA keys are tolerated.
2. You can enforce a dual-control, split-knowledge security policy by employing the **FIRST**, **ADD-PART**, and **COMPLETE** keywords. See "Required commands" on page 181. New applications should employ the **ADD-PART** and **COMPLETE** keywords in lieu of the **MIDDLE** and **LAST** keywords in order to ensure a separation of responsibilities between someone who can add key-part information and someone who can declare that appropriate information has been accumulated in a key. Consider using the Key_Test verb to ensure a correct key-value has been accumulated prior to using the **COMPLETE** option to mark the key as fully operational.

Restrictions

A "replicated key-halves" key (both cleartext halves of a double-length key are equal) performs like a single-length DES key and is therefore weaker than a double-length key with unequal halves. Key parity bits are ignored.

When the Unrestrict Combine Key Parts command (offset X'027A') is turned off in the active role, and when the key information decrypted from the key token is a double-length key and has other than all-zero key bits (parity bits are ignored), the halves of the key decrypted from the source key-token and the halves of the updated key are inspected. The updated key is only returned if either the halves of the source and the updated key are both equal or both unequal. When the equality of the key-halves of the resulting accumulated key represents a change from the equality of the source-key halves, the verb terminates with return code 8 and reason code 2062.

Format

CSNBKPI

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Table 16. Key_Part_Import rule_array keywords

Keyword	Meaning
Key part (one required)	
FIRST	Specifies that an initial key-part is provided. The verb returns this key-part encrypted by the master key in the key token which you supplied.
ADD-PART	Specifies that additional key-part information is provided. The verb exclusive-ORs the key part into the key information held encrypted in the key token.
COMPLETE	Specifies that the key-part bit shall be turned off in the control vector of the key rendering the key fully operational. No key_part information is added to the key with this keyword.
MIDDLE	Specifies that an intermediate key-part, which is neither the first key-part nor the last key-part, is provided. The verb exclusive-ORs the key part into the key information held encrypted in the key token.The command control point for this keyword is the same as that for the **LAST** keyword and different from that for the **ADD-PART** keyword.
LAST	Specifies that the last key-part is provided. The verb exclusive-ORs the key part into the key information held encrypted in the key token. The key-part bit is turned off in the control vector.

key_part
The *key_part* parameter is a pointer to a string variable containing a key part to be entered. The key part might be either 8 or 16 bytes in length. For 8-byte keys, place the key part in the high-order bytes of the 16-byte key-part field. The information in this variable must be defined but are ignored by the coprocessor when you use the **COMPLETE** rule-array keyword.

key_identifier
The *key_identifier* parameter is a pointer to a 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 active role:

- Load First Key Part (offset X'001B') with the **FIRST** keyword.

- Combine Key Parts (offset X'001C') with the **MIDDLE** and **LAST** keywords.
- Add Key Part (offset X'0278') with the **ADD-PART** keyword.
- Complete Key Part (offset X'0279') with the **COMPLETE** keyword.

The Key_Part_Import verb enforces the key-halves restriction documented above when the Unrestrict Combine Key Parts command (offset X'027A') is disabled in the active role. Enabling this command results in less secure operation and is not recommended.

Key_Test (CSNBKYT)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

You use the Key_Test verb to verify the value of a key or key-part. Several verification algorithms are supported. The verb supports testing of clear keys, enciphered keys, master keys, and key-parts. The verification pattern and the verification processes do not reveal the value of an encrypted key, other than equivalency of two key values.

See also the Key_Test_Extended verb for operating on external keys.

The verb operates in either a **GENERATE** or **VERIFY** mode that you specify with a rule-array keyword. You also specify the type of key or key-part.

If you test one of the master keys (keywords **KEY-KM**, **KEY-NKM**, or **KEY-OKM**) you might specify which class of master key to test, either symmetric or asymmetric, using the **SYM-MK** and the **ASYM-MK** rule-array keywords. If you do not select a master-key class, the verb requires that both selected asymmetric and symmetric master-keys have the same value.

There are three verification methods that apply. See “Master key verification algorithms” on page 409. For historical reasons, the verification information is passed in two 8-byte variables, *random_number* and *verification_pattern*. For simplicity, these variables can be two 8-byte elements of a 16-byte array and processed by your application as a single quantity. Both parameters must be coded when calling the API.

- When the verb generates a verification pattern, it returns information in the random number and verification pattern variables.
- When the verb tests a verification pattern, it uses information supplied in the random number and verification pattern variables. Supply the verification data and random number from a previous procedure call to the Key_Test verb. The verb returns the verification results in the form of a return code. If verification fails, the verb returns a return code of 4 and reason code of 1.

For certain types of keys, you can specify an alternative key-test algorithm using a rule-array keyword. The algorithms are explained in “Cryptographic key verification techniques” on page 409.

- Except for master keys, you can specify the **ENC-ZERO** algorithm. The verification information is provided in the four high-order bytes of the verification pattern variable.
- For master keys, you can specify the **MDC-4** algorithm.

Specify the type of key or key-part with a rule-array keyword: master key, clear or enciphered, and so forth.

The optional **ADJUST** keyword adjusts the low-order bit in each key byte so that the byte incorporates an odd number of one-bits. Use of this keyword assures that the verification pattern is computed without regard to the “parity” of the tested key or key-part. You can use the optional, default **NOADJUST** keyword to preserve the supplied “parity” bits. Any parity adjustment is only used for key-verification purposes. The original value of the key or key-part is preserved.

Restrictions

Release 3.20 and later support the **ADJUST** and **NOADJUST** keywords.

Format

CSNBKYT

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	2, 3, 4, or 5
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 9.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 2, 3, 4, or 5 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Process rule (one required)	
GENERATE	Generates a verification pattern.
VERIFY	Verifies a verification pattern.
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.
Master-key selector (one, optional)	
SYM-MK	Specifies use of the symmetric master-key registers.
ASYM-MK	Specifies use of the asymmetric master-key registers.
Parity adjustment (one, optional)	
ADJUST	Adjust the low-order bit in each key byte used in the key-test computation so that the byte contains an odd number of one bits.
NOADJUST	Do not alter the key-byte values. This is the default.
Verification-process rule (one, optional)	
ENC-ZERO	Specifies use of the “encrypt zeros” method. Use only with **KEY-CLR**, **KEY-CLRD**, **KEY-ENC**, or **KEY-ENCD** keywords.
MDC-4	Specifies use of the MDC-4 master-key-verification method. Use only with **KEY-NKM**, **KEY-KM**, or **KEY-OKM** keywords.

key_identifier

The *key_identifier* parameter is a pointer to a 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 cannot be a clear key.

random_number

The *random_number* parameter is a pointer to a string variable containing a number the verb might use in the verification process. When you specify the **GENERATE** keyword, the verb returns the random number. When you specify the **VERIFY** keyword, you must supply the number. With the **ENC-ZERO** method, the random_number variable is not used but must be specified.

verification_pattern

The *verification_pattern* parameter is a pointer to a string variable containing the binary verification pattern. When you specify the **GENERATE** keyword, the verb returns the verification pattern. When you specify the **VERIFY** keyword, you must supply the verification pattern.

With the **ENC-ZERO** method, the verification data occupies the high-order four bytes while the low-order four bytes are unspecified (the data is passed between your application and the cryptographic engine but is otherwise unused). See “Cryptographic key verification techniques” on page 409.

Required commands

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

Key_Test_Extended (CSNBKYTX)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4764		3.20	
		3.23	

You use the Key_Test_Extended verb to verify the value of an external key, internal key, or key part. The Key_Test_Extended verb differs from the Key_Test verb:
- It also operates on external keys
- It does not support **KEY-CLR** or **KEY-CLRD** keywords supported in the Key_Test verb

Several verification algorithms are supported. The verb supports testing of clear keys, enciphered keys, master keys, and key parts. The verification pattern and the verification processes do not reveal the value of an encrypted key, other than equivalency of two key values.

The verb operates in either a **GENERATE** or **VERIFY** mode that you specify with a rule-array keyword. You also specify the type of key or key part.

If you test one of the master keys (keywords **KEY-KM**, **KEY-NKM**, or **KEY-OKM**), you might specify which class of master key to test, either symmetric or asymmetric, using the **SYM-MK** and the **ASYM-MK** rule-array keywords. If you do not select a master-key class, the verb requires that both selected asymmetric and symmetric master-keys have the same value. There are three verification methods that apply. See “Master key verification algorithms” on page 409.

For historical reasons, the verification information is passed in two 8-byte variables, *random_number* and *verification_pattern*. For simplicity, these variables can be two 8-byte elements of a 16-byte array and processed by your application as a single quantity. Both parameters must be coded when calling the API.
- When the verb generates a verification pattern, it returns information in the random number and verification pattern variables.
- When the verb tests a verification pattern, it uses information supplied in the random number and verification pattern variables. Supply the verification data and random number from a previous procedure call to the Key_Test_Extended verb. The verb returns the verification results in the form of a return code. If verification fails, the verb returns a return code of 4 and reason code of 1.

For certain types of keys, you can specify an alternative key-test algorithm using a rule-array keyword. The algorithms are explained in “Cryptographic key verification techniques” on page 409.
- Except for master keys, you can specify the **ENC-ZERO** algorithm. The verification information is provided in the four high-order bytes of the verification pattern variable.
- For master keys, you can specify the **MDC-4** algorithm.

Specify the type of key or key part with a rule-array keyword: master key, clear or enciphered, and so forth.

The optional **ADJUST** keyword adjusts the low-order bit in each key byte so that the byte incorporates an odd number of one-bits. Use of this keyword assures that the verification pattern is computed without regard to the parity of the tested key or key part. You can use the optional, default **NOADJUST** keyword to preserve the supplied parity bits. Any parity adjustment is only used for key-verification purposes. The original value of the key or key part is preserved.

Restrictions

This verb is first supported in Release 3.20.

Format

CSNBKYTX

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	2, 3, 4, or 5
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 9.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 2, 3, or 4 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Process rule (one required)	
GENERATE	Generates a verification pattern.
VERIFY	Verifies a verification pattern.
Key or key-part rule (one required)	
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.
Master-key selector (one, optional)	
SYM-MK	Specifies use of the symmetric master-key registers.
ASYM-MK	Specifies use of the asymmetric master-key registers.
Parity adjustment (one, optional)	
ADJUST	Adjust the low-order bit in each key byte used in the key-test computation so that the byte contains an odd number of one bits.
NOADJUST	Do not alter the key-byte values. This is the default.
Verification-process rule (one, optional)	
ENC-ZERO	Specifies use of the “encrypt zeros” method. Use only with **KEY-CLR**, **KEY-CLRD**, **KEY-ENC**, or **KEY-ENCD** keywords.
MDC-4	Specifies use of the MDC-4 master-key-verification method. Use only with **KEY-NKM**, **KEY-KM**, or **KEY-OKM** keywords.

key_identifier

The *key_identifier* parameter is a pointer to a 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 cannot be a clear key.

random_number

The *random_number* parameter is a pointer to a string variable containing a number the verb might use in the verification process. When you specify the **GENERATE** keyword, the verb returns the random number. When you specify the **VERIFY** keyword, you must supply the number. With the **ENC-ZERO** method, the random_number variable is not used but must be specified.

verification_pattern

The *verification_pattern* parameter is a pointer to a string variable containing the binary verification pattern. When you specify the **GENERATE** keyword, the verb returns the verification pattern. When you specify the **VERIFY** keyword, you must supply the verification pattern.

With the **ENC-ZERO** method, the verification data occupies the high-order four bytes while the low-order four bytes are unspecified (the data is passed between your application and the cryptographic engine but is otherwise unused). See “Cryptographic key verification techniques” on page 409.

Required commands

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

Key_Token_Build (CSNBKTB)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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. See Figure 5 on page 132.

The Key_Token_Build verb does not perform cryptographic services on any key value. You cannot use this verb to change a key or to change the control vector related to a key.

Restrictions

Note: Version 1 code used a smaller master key verification pattern. Beginning with Version 2, the verb interface is changed to accept an 8-byte verification pattern identified by the *master_key_verification_pattern* parameter.

Format

CSNBKTB			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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
reserved_1*	Input	void *	Integer valued to 0
reserved_2	Input	Integer	null pointer or 0
reserved_3	Input	String	null pointer or XL8'00'
control_vector	Input	String	16 bytes
reserved_4	Input	String	null pointer or XL8'00'
reserved_5	Input	Integer	null pointer or 0
reserved_6	Input	String	null pointer or 8-space variable
master_key_verification_pattern	Input	String	8 bytes

* Previous implementations used the *reserved_1* parameter to point to a four-byte integer or string that represented the master key verification pattern. Beginning with Version 2, the CCA Support Program requires this parameter to point to a four-byte value equal to binary zero.

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 9.

key_token

The *key_token* parameter is a pointer to a string variable containing the assembled key-token.

Note: This variable cannot contain a key label.

key_type

The *key_type* parameter is a pointer to a string variable containing a keyword that defines the key type. The keyword is 8 bytes in length and must be left-aligned and padded on the right with space characters. Valid key_type keywords are shown below:

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

For information about key types, see Appendix C, “CCA control-vector definitions and key encryption,” on page 385.

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 variable.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Table 17. Key_Token_Build rule_array keywords

Keyword	Meaning
Token type (one required)	
INTERNAL	Specifies an internal key-token.
EXTERNAL	Specifies an external key-token.
Key status (one, 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 (one, optional) **Note:** If you specify the **USE-CV** keyword in the *key_type* parameter, use the **CV** keyword here.	
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.
Control-vector keywords (one or more, optional).	
	See Figure 5 on page 132 for the key-usage keywords that can be specified for a given key type.

key_value
The *key_value* parameter is a pointer to a string variable containing the encrypted key-value incorporated into the encrypted-key portion of the key token if you use the **KEY** rule_array keyword. Single-length keys must be left-aligned in the variable and padded on the right (low-order) with eight bytes of X'00'.

control_vector
The *control_vector* parameter is a pointer to a string variable. If you use the **CV** rule-array keyword, the variable is copied to the control-vector field of the key token.

master_key_verification_pattern
The *master_key_verification_pattern* parameter is a pointer to a string variable. The value is inserted into the key token when you specify both the **KEY** and **INTERNAL** keywords in the rule array.

Required commands

None

Key_Token_Change (CSNBKTC)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

Use the Key_Token_Change verb to reencipher a DES key from encryption under the old master-key to encryption under the current master-key and to update the keys in internal DES key-tokens.

Notes:

1. An application system is responsible for keeping all of its keys in a usable form. When the master key is changed, the IBM 4764 and IBM 4758 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 keys.
2. Previous implementations of IBM CCA products had additional capabilities with this verb such as deleting key records and key tokens in key storage. Also, use of a wild card (*) was supported in those implementations.

Restrictions

None

Format

CSNBKTC			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Table 18. Key_Token_Change rule_array keywords

Keyword	Meaning
RTCMK	Reenciphers a DES key to the current master-key in an internal key-token in application storage or in key storage If the supplied key is already enciphered under the current master-key the verb returns a positive response (return code 0, reason code 0). If the supplied key is enciphered under the old master-key, the key is updated to encipherment by the current master-key and the verb returns a positive response (return code 0, reason code 0). Other cases return some form of abnormal response.

key_Identifier
The *key_identifier* parameter is a pointer to a string variable containing the DES internal key-token or the key label of an internal key-token record in key storage.

Required commands

If you specify RTCMK keyword, the Key_Token_Change verb requires the Reencipher to Current Master Key command (offset X'0090') to be enabled in the active role.

Key_Token_Parse (CSNBKTP)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Key_Token_Parse verb disassembles a key token into separate pieces of information. The verb can disassemble an external key-token or an internal key-token in application storage.

Use the *key_token* parameter to specify the key token to disassemble.

The verb returns some of the key-token information in a set of variables identified by individual parameters and the remaining key-token information as keywords in the rule array.

Control vector information is returned in keywords found in the rule array when the verb can fully parse the control vector. Supported keywords are shown in Figure 5 on page 132. Otherwise, the verb returns return code 4, reason code 2039.

The Key_Token_Parse verb performs no cryptographic services.

Restrictions

None

Format

CSNBKTP			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_token	Input	String	64 bytes
key_type	Output	String	8 bytes
rule_array_count	In/Output	Integer	
rule_array	Output	String array	rule_array_count * 8 bytes
key_value	Output	String	16 bytes
MKVP	Output	Integer	(only for a version X'03' internal-token)
reserved_2	Output	Integer	
reserved_3	Output	String	8 bytes
control_vector	Output	String	16 bytes
reserved_4	Output	String	8 bytes
reserved_5	Output	Integer	
reserved_6	Output	String	8 bytes
master_key_verification_pattern	Output	String	8 bytes (Only for a version X'00' internal token)

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 9.

key_token

The *key_token* parameter is a pointer to a string variable in application storage containing an external or internal key-token to be disassembled.

Note: You cannot use a key label for a key-token record in key storage. The key token must be in application storage.

key_type

The *key_type* parameter is a pointer to a string variable containing a keyword defining the key type. The keyword is 8 bytes in length and must be left-aligned and padded on the right with space characters. Valid key_type keywords are shown below:

CIPHER	DATAC	EXPORTER	MACVER
CVARDEC	DATAM	IKEYXLAT	OKEYXLAT
CVARENC	DATAMV	KEYGENKY	OPINENC
CVARPINE	DECIPHER	IMPORTER	PINGEN
CVARXCVL	DKYGENKY	IPINENC	PINVER
CVARXCVR	ENCIPHER	MAC	SECMSG
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 variable. This value must be a minimum of 3 and should be at least 20 for this verb.

On input, specify the maximum number of usable array elements that are allocated. On output, the verb sets the value to the number of keywords returned to the application.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords that expresses the contents of the key token. The keywords are 8 bytes in length and are left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Table 19. Key_Token_Parse rule_array keywords

Keyword	**Meaning**
Token type (one returned)	
INTERNAL	Specifies an internal key-token.
EXTERNAL	Specifies an external key-token.
Key status (one returned)	
KEY	Indicates the key token contains a key. The key_value variable contains the key.
NO-KEY	Indicates the key token does not contain a key.
Control-vector (CV) status (one returned)	
CV	The key token specifies that a control vector is present. The verb sets the control vector variable with the value of the control vector found in the key token.
NO-CV	The key token does not specify the presence of a control vector. The verb sets the control vector variable with the value of the control vector variable found in the key token.
Control-vector keywords	
	See Figure 5 on page 132 for the key-usage keywords that can result with a given key type.

key_value
The *key_value* parameter is a pointer to a string variable. If the verb returns the **KEY** keyword in the rule array, the key-value variable contains the 16-byte enciphered key.

MKVP
The *MKVP* parameter is a pointer to an integer variable. The verb writes zero into the variable except when parsing a version X'03' internal key-token.

reserved_2/5
The *reserved_2* and *reserved_5* parameters are either null pointers or pointers to integer variables. If the parameter is not a null pointer, the verb writes zero into the reserved variable.

reserved_3/4
The *reserved_3* and *reserved_4* parameters are either null pointers or pointers to string variables. If the parameter is not a null pointer, the verb writes eight bytes of X'00' into the reserved variable.

reserved_6
The *reserved_6* parameter is either a null pointer or a pointer to a string variable. If the parameter is not a null pointer, the verb writes eight space characters into the reserved variable.

control_vector
The *control_vector* parameter is a pointer to a string variable in application storage. If the verb returns the **NO-CV** keyword in the rule array, the key token did not contain a control-vector value and the control vector variable is filled with 16 space characters.

master_key_verification_pattern
The *master_key_verification_pattern* parameter is a pointer to a string variable in application storage. For version 0 key-tokens that contain a key, the 8-byte

master key version number will be copied to the variable. Otherwise the variable is filled with eight space characters.

Required commands

None

Key_Translate (CSNBKTR)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 reenciphered.
- 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 variable 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	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

input_key_token
The *input_key_token* parameter is a pointer to a string variable containing an external key-token. The external key-token contains the key to be reenciphered (translated).

input_KEK_key_identifier
The *input_KEK_key_identifier* parameter is a pointer to a 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 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 string variable containing an external key-token. The external key-token contains the reenciphered key.

Required commands

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

Multiple_Clear_Key_Import (CSNBCKM)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Multiple_Clear_Key_Import verb multiply-enciphers a clear, single-length or double-length DES DATA key under a symmetric master-key.

You can use this verb to create an internal key-token from a null key token. In this case, the control vector is set to the value of a single-length or double-length default control-vector. Or, you can update an existing internal DATA key-token with the enciphered value of the clear key.

You can specify a key label of an existing record in key storage.

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

CSNBCKM

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0 or 1
rule_array	Input	String array	rule_array_count * 8 bytes
clear_key_length	Input	Integer	8 or 16
clear_key	Input	String	clear_key_length bytes
key_identifier	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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 0 or 1 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Algorithm (optional)	
DES	The key should be enciphered under the master key as a DES key. This is the default.

clear_key_length
The *clear_key_length* parameter is a pointer to an integer variable containing the number of bytes of data in the clear_key variable.

clear_key
The *clear_key* parameter is a pointer to a string variable containing the single-length (8-byte) or double-length (16-byte) plaintext DES-key to be imported.

key_identifier
The *key_identifier* parameter is a pointer to a string variable containing a null key-token, or an internal key-token, or the key label of an internal key-token record in key storage. A key token is returned to the application, or to key storage if the label of a valid key-storage record was specified.

Required commands

| The Multiple_Clear_Key_Import verb requires the Encipher Under Master Key
| command (offset X'00C3') to be enabled in the active role.

PKA_Decrypt (CSNDPKD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Decrypt verb decrypts (unwraps) input data using an RSA private-key. The decrypted data is examined to ensure it meets RSA DSI PKCS #1 block type 2 format specifications. See "PKCS #1 Hash Formats" on page 425.

Restrictions

1. A key-usage flag bit (see offset 050 in the private-key section) must be on to permit use of the private key in the decryption of a symmetric key.
2. The RSA private-key modulus size (key size) is limited by the Function Control Vector to accommodate potential governmental export and import regulations. The verb enforces this restriction.

Format

CSNDPKD			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
source_encrypted_key_length	Input	Integer	
source_encrypted_key	Input	String	source_encrypted_key_length bytes
data_structure_length	Input	Integer	
data_structure	In/Output	String	data_structure_length bytes
private_key_identifier_length	Input	Integer	
private_key_identifier	Input	String	private_key_identifier_length bytes
clear_target_key_length	In/Output	Integer	
clear_target_key	Output	String	clear_target_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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Recovery method (required)	
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02 documentation. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.

source_encrypted_key_length
The *source_encrypted_key_length* parameter is a pointer to an integer variable containing the number of bytes of data in the source_encrypted_key variable. The maximum size allowed is 256 bytes.

source_encrypted_key
The *source_encrypted_key* parameter is a pointer to a string variable containing the input key to be decrypted.

data_structure_length
The *data_structure_length* parameter is a pointer to an integer variable containing the number of bytes of data in the data_structure variable. This value must be 0.

data_structure
The *data_structure* parameter is a pointer to a string variable. This variable is currently ignored.

private_key_identifier_length
The *private_key_identifier_length* parameter is a pointer to an integer variable containing the number of bytes of data in the private_key_identifier variable. The maximum size allowed is 2500 bytes.

private_key_identifier
The *private_key_identifier* parameter is a pointer to a string variable containing the RSA private-key token, or the label of an RSA private-key token in key storage, used to decrypt the source key.

clear_target_key_length
The *clear_target_key_length* parameter is a pointer to an integer variable containing the number of bytes of data in the clear_target_key variable. On input, this variable specifies the maximum permissible length of the result. On output, this verb updates the variable to indicate the length of the returned key. The maximum size allowed is 256 bytes.

clear_target_key
The *clear_target_key* parameter is a pointer to a string variable containing the decrypted (clear) key returned by this verb.

Required commands

| The PKA_Decrypt verb requires the PKA Decipher Key Data command (offset
| X'011F') to be enabled in the active role.

PKA_Encrypt (CSNDPKE)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Encrypt verb encrypts (wraps) input data using an RSA public key. The data that you encrypt might include:

- For keys, the encrypted data can be formatted according to RSA DSI PKCS #1 block type 2 format specifications. See “PKCS #1 Hash Formats” on page 425.
- Other data, such as a digital signature, can be RSA-ciphered using the public key and the **ZERO-PAD** option. The data that you provide is padded on the left with zero bits to the modulus length of the public key. When validating a digital signature using the **ZERO-PAD** option, you are responsible for formatting of the hash and any other required information.

Restrictions

The RSA public-key modulus size (key size) is limited by the Function Control Vector to accommodate governmental export and import regulations.

A message can be encrypted provided that it is smaller than the public key modulus.

The **ZERO-PAD** rule-array keyword is only available starting with Release 2.50.

Format

CSNDPKE			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
clear_source_data_length	Input	Integer	
clear_source_data	Input	String	clear_source_data_length bytes
data_structure_length	In/Output	Integer	
data_structure	Input	String	data_structure_length bytes
public_key_identifier_length	Input	Integer	
public_key_identifier	Input	String	public_key_identifier_length bytes
target_data_length	In/Output	Integer	
target_data	Output	String	target_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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	**Meaning**
Format method (one required)	
PKCS-1.2	Specifies the method found in RSA DSI PKCS #1 block type 02 documentation. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.
ZERO-PAD	Places the supplied data in the low-order bit positions of a bit string of the same length as the modulus. As required, high-order bits are set to zero. Ciphers the resulting bit-string with the public key.

clear_source_data_length

The *clear_source_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the clear_source_data variable. When using the **PKCS-1.2** keyword, the maximum size allowed is 245 bytes with a 2048-bit public key. When using the **ZERO-PAD** keyword, the maximum size allowed is 256 bytes with a 2048-bit public key.

clear_source_data

The *clear_source_data* parameter is a pointer to a string variable containing the input data to be encrypted.

data_structure_length

The *data_structure_length* parameter is a pointer to an integer variable containing the number of bytes of data in the data_structure variable. This value must be 0.

data_structure

The *data_structure* parameter is a pointer to a string variable. This variable is currently ignored.

public_key_identifier_length

The *public_key_identifier_length* parameter is a pointer to an integer variable containing the number of bytes of data in the public_key_identifier variable. The maximum size allowed is 2500 bytes.

public_key_identifier

The *public_key_identifier* parameter is a pointer to a string variable containing the RSA public-key token, or the label of an RSA public-key token in key storage, used to encrypt the source data.

target_data_length

The *target_data_length* parameter is a pointer to an integer variable containing the number of bytes of data in the target_data variable. On input, this variable specifies the maximum permissible length of the result. On output, this verb updates the variable to indicate the length of the returned data. The maximum size allowed is 256 bytes. The data length is the same as the size of the public-key modulus.

target_data

The *target_data* parameter is a pointer to a string variable containing the encrypted data returned by the verb. The returned encrypted target-data is the same length as the public-key modulus.

Required commands

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

PKA_Symmetric_Key_Export (CSNDSYX)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Symmetric_Key_Export verb enciphers a symmetric DES or CDMF default DATA-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 DATA control-vector must have the default value for a single-length or a double-length key as listed in Table 75 on page 387.

Choose a key-formatting method through a rule array keyword specification. The formatted key is then enciphered (wrapped) using the supplied public key. Formatting options:

PKCSOAEP
The PKCSOAEP keyword specifies to format a single-length or double-length DATA key (or CDMF key) according to the method described in the RSA DSI PKCS#1-v2.0 documentation for RSAES-OAEP. See “PKCS #1 Hash Formats” on page 425.

PKCS-1.2
The PKCS-1.2 keyword specifies to format a single-length or double-length DATA key (or CDMF key) according to the method described in the RSA DSI PKCS #1 documentation for block type 2. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format. See “PKCS #1 Hash Formats” on page 425.

ZERO-PAD
The ZERO-PAD keyword specifies to format a single-length or double-length DATA key (or CDMF key) by padding the key value to the left with bits valued to zero.

Restrictions

The RSA public-key modulus size (key size) is limited by the Function Control Vector to accommodate potential governmental export and import regulations.

You can only export a default DATA-key with this verb.

Format

CSNDSYX

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
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 9.

rule_array_count

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

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Key-formatting method (one required)	
PKCSOAEP	Specifies that a DES (or CDMF) DATA-key can be exported using the formatting method found in RSA DSI PKCS#1-v2.0 RSAES-OAEP documentation.
PKCS-1.2	Specifies that a DES (or CDMF) DATA-key can be exported using the formatting method following the rules defined in the RSA Laboratories PKCS#1 v2.0 RSAES-PKCS1-v1_5 specification.
ZERO-PAD	Specifies that a DES (or CDMF) DATA-key can be exported with the key value padded on the left with bits valued to zero.

source_key_identifier_length

The *source_key_identifier_length* parameter is a pointer to an integer variable containing the number of bytes of data in the source_key_identifier variable. The maximum size allowed 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 number of bytes of data in the RSA_public_key_token variable. The maximum size allowed 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 is to import the exported key.

RSA_enciphered_key_length
The *RSA_enciphered_key_length* parameter is a pointer to an integer variable containing the number of bytes of data in the RSA_enciphered_key variable. On output, the variable is updated with the actual length of the RSA_enciphered_key variable. The maximum size allowed is 2500 bytes.

RSA_enciphered_key
The *RSA_enciphered_key* parameter is a pointer to a string variable containing the exported RSA-enciphered key returned by the verb.

Required commands

The PKA_Symmetric_Key_Export verb requires these commands to be enabled in the active role for exporting various key types:

- Symmetric Key Export PKCS-1.2/OAEP (offset X'0105') for DATA keys using the **PKCSOAEP** and **PKCS-1.2** methods
- ZERO-PAD Symmetric Key Export (offset X'023E') for DATA keys using the **ZERO-PAD** method.

PKA_Symmetric_Key_Generate (CSNDSYG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Symmetric_Key_Generate verb generates a random DES-key and enciphers the key value. The key value is enciphered under an RSA public-key for distribution to a remote node (that has the associated private key). The key value is also multiply-enciphered under either the symmetric master-key or a DES key-encrypting-key.

Rule-array keywords define how the RSA-enciphered key shall be enciphered, the length of the generated key, and the type of DES key used to encipher the local copy of the key.

There are three classes of rule-array keywords:

1. Required keywords to select the formatting method used to expand and secure the generated key that is encrypted (wrapped) by the public key. Three of the methods deal with DATA keys and the other two are used with key-encrypting keys.
2. Optional key-length keywords to control the length of the generated key.
3. When generating DATA keys, optional keywords to select the key used to encrypt (wrap) the *local_enciphered_key*.

Key encryption (wrapping) methods:

- DATA keys, either single-length or double-length, can be generated with the default DATA control-vector as defined in Table 75 on page 387. One copy of the key, the *local_enciphered_key*, is returned encrypted by the symmetric master key or by an IMPORTER or EXPORTER key-encrypting-key. If you do not specify a null key-token, you must supply either the single-length or double-length default control vector in a key token.

  The public key is used to wrap another copy of the generated key and returned in the *RSA_enciphered_key_token*. On input you must specify a null key-token. You choose how the generated key shall be formatted prior to RSA encryption using one of these keywords:

  PKCSOAEP
  The key is formatted into an “encrypted message” following the rules defined in the RSA Laboratories PKCS#1 v2.0 RSAES-OAEP specification. See “PKCS #1 Hash Formats” on page 425.

  PKCS-1.2
  The key is formatted into an “encrypted message” following the rules defined in the RSA Laboratories PKCS#1 v2.0 RSAES-PKCS1-v1_5 specification. See “PKCS #1 Hash Formats” on page 425.

  ZERO-PAD
  The generated key value is extended with zero bits to the left.
- Key-encrypting keys, either effective single-length or true double-length, are generated with the details dependent on the keyword you use to control the key formatting technique.

PKA92

With this keyword, the verb generates a key-encrypting key and returns two copies of the key. You must specify a pair of complementary control vectors that conform to the rules for an OPEX case as defined for the Key_Generate verb. The control vector for one key copy must be from the EXPORTER class while the control vector for the other key-copy must be from the IMPORTER class.

The verb enciphers one key copy using the *RSA_public_key* and the key encipherment technique defined in “PKA92 key format and encryption process” on page 405. The control vector for this key is taken from an internal (operational) DES key token that must be present on input in the *RSA_enciphered_key_token* variable.

The control vector for the local key is taken from a DES key token that must be present on input in the *local_enciphered_key_identifier* variable or in the key token identified by the key label in that variable.

Note: A node-identification (EID) value must have been established prior to use of the **PKA92** keyword. Use the Cryptographic_Facility_Control verb to set the EID.

NL-EPP-5

With this keyword, the verb generates a key-encrypting key and returns two copies of the key. The verb enciphers one key copy using the key encipherment technique defined by certain OEM equipment. See “Encrypting a key-encrypting key in the NL-EPP-5 format” on page 407. On input, the *RSA_enciphered_key_token* variable must contain a DES internal key token that contains a control vector for an IMPORTER key-encrypting-key.

The control vector for the local key is taken from a DES key token that must be present on input in the *local_enciphered_key_identifier* variable or in the key token identified by the key label in that variable.

Restrictions

The permissible key-length of the RSA public key is limited by the value specified in the function control vector for RSA encipherment of keys.

Format

CSNDSYG			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1, 2, or 3
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
local_enciphered_key_ identifier_length	In/Output	Integer	
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

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 9.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 1, 2, or 3 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
Key-formatting method (one required)	
PKCSOAEP	Specifies the PKCS#1-V2.0 OAEP method of key encipherment for DATA keys.
PKCS-1.2	Specifies the PKCS #1, block type 2 method of key encipherment for DATA keys. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format.
ZERO-PAD	Specifies the pad-with-zero-bits-to-the-left method of key encipherment for DATA keys.
PKA92	Specifies the PKA92 method of key encipherment for key-encrypting keys.
NL-EPP-5	Specifies the NL-EPP-5 process of key encipherment for key-encrypting keys. See “Encrypting a key-encrypting key in the NL-EPP-5 format” on page 407.
Key length (optional use with **PKA92** or **NL-EPP-5**)	
SINGLE-R	For key-encrypting keys, specifies that a generated key-encrypting key is to have equal left and right halves and thus perform as a single-length key. Otherwise, the two key-halves are independent random values.
Key length (optional use with **PKCSOAEP**, **PKCS-1.2**, and **ZERO-PAD**)	
SINGLE, **KEYLN8**	Specifies that an exported DATA key should be single length. This the default.
DOUBLE, **KEYLN16**	Specifies that an exported DATA key should be double length.
DES encipherment (optional use with **PKCSOAEP**, **PKCS-1.2**, and **ZERO-PAD**)	
OP	Enciphers one key copy with the symmetric master-key. This is the default.
IM	Enciphers one key copy using the IMPORTER key-encrypting-key specified with the *key_encrypting_key_identifier* parameter.
EX	Enciphers one key copy using the EXPORTER key-encrypting-key specified with the *key_encrypting_key_identifier* parameter.

key_encrypting_key_identifier

The *key_encrypting_key_identifier* parameter is a pointer to a string variable containing the key token or the key label of a key token in key storage with the key-encrypting key used to encipher one generated-key copy for DES-based key distribution.

RSA_public_key_identifier_length
The *RSA_public_key_identifier_length* parameter is a pointer to an integer variable containing the number of bytes of data in the RSA_public_key_identifier variable. The maximum size allowed 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 imports the exported key.

local_enciphered_key_identifier_length
The *local_enciphered_key_identifier_length* parameter is a pointer to an integer variable containing the number of bytes of data in the local_enciphered_key_identifier variable. The maximum size allowed 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 a key token. The control vector for the local key is taken from the identified key token. On output, the generated key is inserted into the identified key token.

On input, you must specify a token type consistent with your choice of local-key encryption. If you specify **IM** or **EX**, you must specify an external key-token. Otherwise, specify an internal key-token or a null key-token.

When **PKCSOAEP**, **PKCS-1.2**, or **ZERO-PAD** is specified, a null key-token can be specified. In this case, a DATA key is returned. For an internal key (**OP**), a default DATA control-vector is returned in the key token. For an external key (**IM** or **EX**), the control vector is set to null.

RSA_enciphered_key_token_length
The *RSA_enciphered_key_token_length* parameter is a pointer to an integer variable containing the number of bytes of data in the RSA_enciphered_key_token variable. On output, the variable is updated with the actual length of the RSA_enciphered_key_token variable. The maximum size allowed is 2500 bytes.

RSA_enciphered_key_token
The *RSA_enciphered_key_token* parameter is a pointer to a string variable containing the generated RSA-enciphered key returned by the verb. If you specify **PKCS-1.2** or **ZERO-PAD**, on input you should specify a null key token. If you specify **PKA92** or **NL-EPP-5**, on input specify an internal (operational) DES key-token.

Required commands

The PKA_Symmetric_Key_Generate verb requires these commands to be enabled in the active role depending on the key-formatting method:

- Symmetric Key Generate PKCS-1.2/OAEP (offset X'023F') for DATA keys using the **PKCSOAEP** and **PKCS-1.2** methods
- Symmetric Key Generate ZERO-PAD (offset X'023C') for DATA keys using the **ZERO-PAD** method
- Symmetric Key Generate (offset X'010D')
- NL-EPP-5 Symmetric Key Generate (offset X'010E')

PKA_Symmetric_Key_Import (CSNDSYI)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 DES-key using the master key and a control vector.

You specify the operational importing RSA private-key, the RSA-enciphered DES key to be imported, and a rule-array keyword to define the key-formatting method.

Several methods for recovering keys are available. You select a method using of a rule-array keyword:

For processing single-length or double-length DATA keys, use one of the these three methods. The control vector in any non-NULL key token identified by the *target_key_identifier* parameter must specify the default value for a DATA control-vector corresponding to the key length found in the decrypted information. See Table 75 on page 387.

PKCSOAEP
: The PKCSOAEP keyword specifies that after decrypting the RSA_enciphered_key variable, the format is checked for conformance with RSA DSI PKCS#1-v2.0 RSAES-OAEP specifications for a single-length or double-length key. See “PKCS #1 Hash Formats” on page 425.

PKCS-1.2
: The PKCS-1.2 keyword specifies that after decrypting the RSA_enciphered_key variable, the format is checked for conformance with RSA DSI PKCS #1 block type 2 specifications for a single-length or double-length key. In the RSA PKCS #1 v2.0 standard, RSA terminology describes this as the RSAES-PKCS1-v1_5 format. See “PKCS #1 Hash Formats” on page 425.

ZERO-PAD
: The ZERO-PAD keyword specifies that after decrypting the RSA_enciphered_key variable, the format is checked to ensure that all bytes to the left of either a single-length or a double-length key are zero bits.

For key-encrypting keys:

PKA92
: Key-encrypting keys and their control vectors are deciphered using the method employed in the TSS3 PKA92 implementation. See “PKA92 key format and encryption process” on page 405.

  A node-identification (EID) value must be established prior to use of this verb. Under the PKA92 scheme, the EID values at the exporting and importing nodes must be different. Use the Cryptographic_Facility_Control verb to set the EID.

Note: This implementation imports IPINENC, OPINENC, PINGEN, and PINVER key types when formatted according to the PKA92 scheme. However, the implementation does not provide a means for enciphering these key types in PKA92 format. This extension to CCA is considered nonstandard, and might not be present in other CCA implementations such as the implementation on IBM eServer zSeries (S/390).

Restrictions

1. Private key key-usage controls can prevent use of specific private keys in this verb. See "PKA_Key_Generate (CSNDPKG)" on page 87. A key-usage flag bit (see offset 050 in the private-key section) must be on to permit use of the private key in the decryption of a symmetric key.
2. The RSA private-key modulus size (key size) is limited by the Function Control Vector to accommodate potential governmental export and import regulations.
3. Under PKA92, the EID enciphered with a key-encrypting key cannot be the same as the EID of the importing cryptographic engine.
4. Other IBM implementations of this verb might 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	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
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
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

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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown below:

Keyword	Meaning
RSA key-encipherment method (one required)	
PKCSOAEP	Specifies the method found in RSA DSI PKCS#1-v2.0 RSAES-OAEP documentation.
PKCS-1.2	Specifies the method found in RSA DSI PKCS#1-v2.0 RSAES-PKCS1-v1_5 specification.
ZERO-PAD	Specifies that a DES (or CDMF) DATA-key can be imported with the key value padded from the left with bits valued to zero.
PKA92	Specifies the PKA92 method of key encipherment for key-encrypting keys.

RSA_enciphered_key_length
The *RSA_enciphered_key_length* parameter is a pointer to an integer containing the number of bytes of data in the RSA_enciphered_key variable. The maximum size allowed 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 number of bytes of data in the RSA_private_key_identifier variable. The maximum size allowed 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 number of bytes of data in the target_key_identifier variable. On output, the value is updated with the actual length of the target_key_identifier variable returned by the verb. The maximum size allowed 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 active role for importing various key types:

- Symmetric Key Import PKCS-1.2/OAEP (offset X'0106') for DATA keys using the **PKCSOAEP** and **PKCS-1.2** methods
- Symmetric Key Import ZERO-PAD (offset X'023D') for DATA keys using the **ZERO-PAD** methods

- Symmetric Key Import (offset X'0235') when importing key-generating keys using the **PKA92** method
- Symmetric Key Import with PIN Keys (offset X'0236') when importing PINGEN, PINVER, IPINENC, or OPINENC keys using the **PKA92** method

Prohibit_Export (CSNBPEX)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Prohibit_Export verb modifies an operational key that can be exported so that it can no longer be exported. (See also the Prohibit_Export_Extended verb for operating on an external key.)

The verb does the following:

- Multiply deciphers the key under a key formed by the exclusive-OR of the master key and the control vector.
- Turns off the export bit in the control vector.
- Multiply enciphers the key under a key formed by the exclusive-OR of the master key and the control vector. The key and the modified control vector are stored in the key token.

Restrictions

None

Format

Prohibit_Export			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length 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 9.

key_identifier

The *key_identifier* parameter is a pointer to a string variable containing the internal key-token, or the key label of an internal key-token record in key storage.

Required commands

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

Prohibit_Export_Extended (CSNBPEXX)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4764		3.20	
		3.23	

The Prohibit_Export_Extended verb modifies an external key so that it can no longer be exported. (See “Prohibit_Export_Extended (CSNBPEXX)” for details about operating on an internal key.)

The verb performs the following functions:
- Multiply deciphers the source key under a key formed by the exclusive-OR of the source key’s control vector and the specified key-encrypting key (KEK).
- Turns off the export (allowed) bit in the source key’s control vector.
- Multiply enciphers the key under a key formed by the exclusive-OR of the KEK key and the source key’s modified control vector. The encrypted key and the modified control vector are stored in the source-key key token.

Restrictions
This verb is first available in Release 3.20.

Format

CSNBPEXX			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
source_key_token	In/Output	String	64 bytes
KEK_key_identifier	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 9.

source_key_token
The *source_key_token* parameter is a pointer to a string variable containing an external key-token.

KEK_key_identifier
The *KEK_key_identifier* parameter is a pointer to a string variable containing an internal key-encrypting-key key-token, or the key label of an internal key-encrypting-key key-token record in key storage.

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

Random_Number_Generate (CSNBRNG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 56 bits with the low-order bit in each of the 8 bytes adjusted for even or for odd parity, or 64 bits without parity adjustment. The verb returns the random number in an 8-byte binary variable.

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	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

form

The *form* parameter is a pointer to a string variable containing a keyword to select the characteristic of the random number. The keyword is 8 bytes in length and must be left-aligned and padded on the right with space characters. The keywords are shown below:

Table 20. Key_Token_Build form keywords

Keyword	**Meaning**
Generation type (one required)	
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 a string variable containing the random number returned by the verb.

Required commands

The Random_Number_Generate verb requires the Generate Key command (offset X’008E’) to be enabled in the active role.

Chapter 6. Data confidentiality and data integrity

This section 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). It contains sections on:

- Encryption and message authentication codes
- Data confidentiality and data integrity verbs

The following table lists the data confidentiality and data integrity verbs. See "Data confidentiality and data integrity verbs" on page 228 for a description of these verbs.

Table 21. Data confidentiality and data integrity verbs

Verb	Page	Service	Entry point	Service location
Decipher	229	Deciphers data	CSNBDEC	E
Encipher	232	Enciphers data	CSNBENC	E
MAC_Generate	235	Generates a message authentication code (MAC)	CSNBMGN	E
MAC_Verify	238	Verifies a MAC	CSNBMVR	E
E=Cryptographic Engine				

Encryption and message authentication codes

This section explains how to use the services that are described to ensure the confidentiality of data through encryption, and to ensure the integrity of data using message authentication codes (MACs).

Note: See Chapter 4, "Hashing and digital signatures," on page 107 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 convert ciphertext back to plaintext. These services use the DES data encryption algorithm. DES operates on blocks of 64 bits (8 bytes). Based on the length of the DES key that you specify, the Encipher and Decipher verbs perform either basic, single DES or triple DES [5]. See "Single-DES and Triple-DES encryption algorithms for general data" on page 413 for more information.

If you know that your data is always in multiples of 8 bytes, you can request the use of the cipher block chaining (CBC) mode of encryption. In this mode, 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 412.

5. CCA implementations always encipher DES keys and PIN blocks with triple DES.

If some portion of the ciphertext is altered, the CBC decryption of that block and the subsequent block does not recover the original plaintext. Other blocks of plaintext are 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, must adopt a strategy for the last block of data. The Encipher and Decipher verbs can also be used with the ANSI X9.23 mode of encryption. In X9.23 encryption, at least 1 byte of data and up to 8 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. The ANSI X9.23 process ensures that the enciphered data is always a multiple of 8 bytes as required for CBC encryption. 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, called 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 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 0 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 or CIPHER-class (control vector bits 8 to 15 equal to X'00' or X'03', respectively). In addition to the class of key defined in CV bits 8 to 14, CV bit 18 must also be on to encipher data while CV bit 19 must also be on to decipher data. See Appendix C, “CCA control-vector definitions and key encryption.” DATA keys can participate in both enciphering and using MAC, while CIPHER-class keys only perform in ciphering operations.

If an invocation of the Encipher or the Decipher verb includes an 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 the data must be a multiple of 8 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 8 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

This section includes the MAC verbs. For information on using hashing or digital signatures to ensure the integrity of data, see Chapter 4, “Hashing and digital signatures.”

The MAC_Generate and the MAC_Verify verbs can authenticate message generation and verification consistent with ANSI standard X9.9, ISO DP 8731, Part I, (ISO/IEC 9797-1, Algorithm 1) and ANSI X9.19 Optional Procedure 1 (ISO/IEC 9797-1, Algorithm 3). These methods together use both single-length and double-length keys. If the specified key is double length, the ANSI X9.19 algorithm is performed; otherwise, ANSI X9.9 is performed. See Appendix C, “CCA control-vector definitions and key encryption.”

The verbs can also be used with the message padding technique employed with EMV smart card messages. The verbs perform EMV-required padding when you supply a rule-array keyword **EMVMAC** or **EMVMACD** consistent with the specified single-length or double-length key.

Both the DATA class and the MAC or MACVER key types can be used. Control vector bit 20 must be on for keys used in the MAC_Generate verb. Control vector bit 21 must be on for keys used in the MAC_Verify verb.

You can employ MAC values with 4-byte, 6-byte, or 8-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 4 bytes (the default method)
- 8 hexadecimal characters, invoked using the **HEX-8** keyword
- 8 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 “MAC_Generate (CSNBMGN)” on page 235 and “MAC_Verify (CSNBMVR)” on page 238.

MAC and segmented data

Using MAC services procedure, you can divide a string of data into parts, and generate or verify a MAC in a series of procedure calls to the appropriate verb. This can be useful when it is no possible to bring the entire string into memory. For example, you might want to use MAC for the entire contents of a data set which is tens or hundreds of megabytes in length. The length of the data in each procedure-call is restricted only by the operating environment and the particular verb.

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)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Decipher verb uses the Data Encryption Standard (DES) or the Commercial Data Masking Facility (CDMF) algorithm and a cipher key to decipher data called ciphertext. This verb results in data called plaintext.

You can enhance performance if you align the start of the plaintext and ciphertext variables on a 4-byte boundary.

Both single-DES and triple-DES are performed based on the length of the key. DATA, CIPHER, and DECIPHER key types can be used. For additional information about the ciphering verbs, see “Ensuring data confidentiality” on page 225.

Restrictions

The starting address of plaintext **cannot** begin within the ciphertext variable.

The *text_length* variable is restricted to a maximum value of 32 MB – 8 bytes, and to 64 MB – 8 bytes in the i5/OS environment.

The installed function control vector regulates the maximum data ciphering capability to CDMF, single DES, or triple DES.

Format

CSNBDEC			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_identifier	Input	String	64 bytes
text_length	In/Output	Integer	
ciphertext	Input	String	text_length bytes
initialization_vector	Input	String	8 bytes
rule_array_count	Input	Integer	0, 1, 2, or 3
rule_array	Input	String array	rule_array_count * 8 bytes
chaining_vector	In/Output	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 9.

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.

text_length
The *text_length* parameter is a pointer to an integer variable. On input, the

text_length variable contains the number of bytes of data in the ciphertext variable. On output, the *text_length* variable contains the number of bytes of data in the plaintext variable.

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 a string variable containing the initialization vector that 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* variable. This value must be 0, 1, 2, or 3 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

For a coprocessor that uses both DES and CDMF, you can choose the encryption process.

The rule_array keywords are shown in the following table:

Keyword	**Meaning**
Deciphering method (one, optional)	
CBC	Specifies cipher block chaining. The data must be a multiple of 8 bytes. This is the default.
X9.23	Specifies cipher block chaining with 1 – 8 bytes of padding. This is compatible with the requirements in ANSI Standard X9.23.
ICV (one, optional)	
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.
Decryption process (one, optional)	
DES	Specifies use of the DES ciphering algorithm. If an adapter cannot use DES general data-decipherment, the verb is rejected. This is the default on an adapter that uses both DES and CDMF algorithms.
CDMF	Specifies use of the CDMF ciphering algorithm.

chaining_vector
The *chaining_vector* parameter is a pointer to a string variable containing the segmented data between calls by the security server. The output chaining vector is contained in bytes zero through seven.

Important: Application programs must not alter the contents of this variable between related **INITIAL** and **CONTINUE** calls.

plaintext

The *plaintext* parameter is a pointer to a string variable containing the plaintext returned by the verb. The starting address of plaintext variable **cannot** begin within the ciphertext variable. The verb updates the *text_length* variable to the length of the plaintext when it returns. The length is different when padding is removed.

Required commands

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

Encipher (CSNBENC)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Encipher verb uses the DES algorithm and a secret key to encipher data. This verb returns data called ciphertext.

The returned ciphertext can be as many as 8 bytes longer than the plaintext due to padding. Ensure the ciphertext variable is large enough to receive the returned data.

Performance can be enhanced by aligning the start of the plaintext and ciphertext variables on 4-byte boundaries.

DATA, CIPHER, and ENCIPHER key-types can be used. Both single DES and triple DES are performed based on the length of the key. For additional information about the ciphering verbs, see “Ensuring data confidentiality” on page 225.

Restrictions

The *text_length* variable is restricted to a maximum value of 32 MB – 8 bytes and to 64 MB – 8 bytes in the i5/OS environment.

The installed function control vector regulates the maximum data ciphering capability to single-DES or triple-DES.

Format

CSNBENC			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_identifier	In/Output	String	64 bytes
text_length	In/Output	Integer	
plaintext	Input	String	text_length bytes
initialization_vector	Input	String	8 bytes
rule_array_count	Input	Integer	0, 1, 2, or 3
rule_array	Input	String array	rule_array_count * 8 bytes
pad_character	Input	Integer	
chaining_vector	In/Output	String	18 bytes
ciphertext	Output	String	updated 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 9.

key_identifier
The *key_identifier* parameter is a pointer to a 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. On input, the *text_length* variable contains the number of bytes of data in the cleartext variable. On output, the *text_length* variable contains the number of bytes of data in the ciphertext variable.

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 a string variable containing the initialization vector that 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* variable. The value must be 0, 1, 2, or 3 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Ciphering method (one, optional)	
CBC	Specifies cipher-block chaining. The data must be a multiple of 8 bytes. This is the default.
X9.23	Specifies cipher-block chaining with 1 – 8 bytes of padding. This is compatible with the requirements in ANSI Standard X9.23.
ICV (one, optional)	
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.
Encryption process (one, optional)	
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 variable containing a value used as a padding character. The value must be 0 – 255. When you use the **X9.23** count byte and padding bytes as required.

chaining_vector

The *chaining_vector* parameter is a pointer to a string variable containing a work area that the security server uses to carry segmented data between procedure calls.

| **Important:** Application programs must not alter the contents of this variable
| between related **INITIAL** and **CONTINUE** calls.

ciphertext

The *ciphertext* parameter is a pointer to a string variable containing the enciphered text returned by the verb. The starting address of the ciphertext variable **cannot** begin within the plaintext variable. The returned ciphertext might be up to 8 bytes longer than the plaintext because of padding. The verb updates the *text_length* variable to the length of the ciphertext when it returns. The length is different when padding is added.

Required commands

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

MAC_Generate (CSNBMGN)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The MAC_Generate verb generates a message authentication code (MAC) for a text string that you supply. For additional information about using the MAC generation and verification verbs, see "Ensuring data integrity" on page 227.

You can enhance performance by aligning the start of the text variable on a 4-byte boundary.

You specify the message authentication code process through the choice of a rule-array keyword. There are defaults based on your use of a single-length or double-length key.

X9.1-1 ANSI X9.9-1 procedure, by default when you supply a single-length key. This is the same as ISO/IEC 9797-1, Algorithm 1.

X9.19OPT
ANSI X9.19 Optional Procedure, by default when you supply a double-length key. This is the same as ISO/IEC 9797-1, Algorithm 3.

EMVMAC and **EMVMACD**
EMV authentication processes. See the *EMV 4.0 Book 2, Annex A.1.2* for information about this form of MAC generation. The verb extends the text you supply with X'80' and the minimum number (0...7) bytes of X'00' for the extended message to be a multiple of 8 bytes in length. The MAC is computed based on ISO/IEC 9797-1, Algorithm 1 or 3 depending on key length. When specifying a single-length key, use **EMVMAC**. When specifying a double-length key, use **EMVMACD**.

Note: The EMV specification permits the MAC to be 4, 5, ..., 8 bytes in length. The MAC_Verify verb only uses MAC lengths of 4, 6, and 8 bytes.

You can specify any of these key types: DATA, DATAM, or MAC.

Restrictions

The *text_length* variable must be at least 8 bytes, and less than 32 MB – 8 bytes, or less than 64 MB – 8 bytes in the i5/OS environment.

Format

CSNBMGN			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	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	0, 1, 2, or 3
rule_array	Input	String array	rule_array_count * 8 bytes
chaining_vector	In/Output	String	18 bytes
MAC	Output	String	4, 6, 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 9.

key_identifier
The *key_identifier* parameter is a pointer to a string variable containing an internal key token or the key label of an internal key-token record in key storage. Use either MAC, DATA, or DATAM key-types. Keys can be either single length or double length.

text_length
The *text_length* parameter is a pointer to an integer variable containing the number of data bytes in the text variable.

text
The *text* parameter is a pointer to a string variable containing the text that 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* variable. The value must be 0, 1, 2, or 3 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
MAC ciphering-method (one, optional)	
EMVMAC	Specifies the EMV-related message-padding and calculation method. You must also specify a single-length key.
EMVMACD	Specifies the EMV-related message-padding and calculation method. You must also specify a double-length key.
X9.9-1	Specifies the ANSI X9.9-1 and X9.19 basic procedure. This is the default for a single-length key.

Keyword	Meaning
X9.19OPT	Specifies the ANSI X9.19 optional procedure. This is the default for a double-length key.
Segmenting control (one, optional)	
ONLY	Specifies that the application program does not use segmenting. This is the default.
FIRST	Specifies that this is the first segment of data from the application program.
MIDDLE	Specifies that this is an intermediate segment of data from the application program.
LAST	Specifies that this is the last segment of data from the application program.
MAC length and presentation (one, optional)	
MACLEN4	Specifies a 4-byte MAC. This is the default.
MACLEN6	Specifies a 6-byte MAC.
MACLEN8	Specifies an 8-byte MAC.
HEX-8	Specifies a 4-byte MAC and presents it as 8 hexadecimal characters.
HEX-9	Specifies a 4-byte MAC and presents it as two groups of 4 hexadecimal characters separated by a space character.

chaining_vector

The *chaining_vector* parameter is a pointer to a string variable containing a work area the security server uses to carry segmented data between procedure calls.

| **Important:** Application programs must not alter the contents of this variable
| between related **FIRST**, **MIDDLE**, and **LAST** calls.

MAC

The *MAC* parameter is a pointer to a string variable containing the resulting MAC returned by the verb. The value is left-aligned in the variable. Allocate a variable 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 active role.

MAC_Verify (CSNBMVR)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The MAC_Verify verb verifies a message authentication code (MAC) for a text string that you supply. For additional information about using the MAC generation and verification verbs, see “Ensuring data integrity” on page 227.

You can enhance performance by aligning the start of the text variable on a 4-byte boundary.

You specify the message authentication code process through the choice of a rule-array keyword. There are defaults based on your use of a single-length or double-length key.

X9.1-1 ANSI X9.9-1 procedure, by default when you supply a single-length key. This is the same as ISO/IEC 9797-1, Algorithm 1.

X9.19OPT
ANSI X9.19 Optional Procedure, by default when you supply a double-length key. This is the same as ISO/IEC 9797-1, Algorithm 3.

EMVMAC and **EMVMACD**
EMV authentication procedure.

Note: See the *EMV 4.0 Book 2, Annex A.1.2, for information about this form of MAC verification.*

The verb extends the text you supply with X'80' and the minimum number (0...7) bytes of X'00' for the extended message to become a multiple of 8 bytes in length. The MAC is computed based on ISO/IEC 9797-1, Algorithm 1 or 3 depending on key length. When specifying a single-length key, use **EMVMAC**. When specifying a double-length key, use **EMVMACD**.

Note: The EMV specification permits the MAC to be 4, 5, ..., 8 bytes in length. This verb only supports MAC lengths of 4, 6, and 8 bytes.

You can specify any of these key types: DATA, DATAM, MAC, or MACVER.

Restrictions

The *text_length* variable must be at least eight bytes, and less than 32 MB – 8 bytes, or less than 64 MB – 8 bytes in the i5/OS environment.

Support for **EMVMAC** and **EMVMACD** begins with Release 2.51.

Format

CSNBMVR			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	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	0, 1, 2, or 3
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 9.

key_identifier
The *key_identifier* parameter is a pointer to a string variable containing an internal key token or the key label of an internal key-token record in key storage. Use either MAC, MACVER, DATA, DATAM, or DATAMV key types. Keys can be either single length or double length.

text_length
The *text_length* parameter is a pointer to an integer variable containing the number of bytes of data in the text variable.

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* variable. The value must be 0, 1, 2, or 3 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
MAC ciphering-method (one, optional)	
EMVMAC	Specifies the EMV-related message-padding and calculation method. You must also specify use of a single-length key.
EMVMACD	Specifies the EMV-related message-padding and calculation method. You must also specify use of a double-length key.

Keyword	Meaning
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 that the application program does not use segmenting. This is the default.
FIRST	Specifies that this is the first segment of data from the application program.
MIDDLE	Specifies that this is an intermediate segment of data from the application program.
LAST	Specifies that this is the last segment of data from the application program.
MAC length and presentation (one, optional)	
MACLEN4	Specifies a 4-byte MAC. This is the default.
MACLEN6	Specifies a 6-byte MAC.
MACLEN8	Specifies an 8-byte MAC.
HEX-8	Specifies a 4-byte MAC and accepts it as 8 hexadecimal characters.
HEX-9	Specifies a 4-byte MAC and accepts it as two groups of 4 hexadecimal characters separated by a space character.

chaining_vector

The *chaining_vector* parameter is a pointer to a string variable containing a work area the security server uses to carry segmented data between procedure calls.

| **Important:** Application programs must not alter the contents of this variable
| between related **FIRST**, **MIDDLE**, and **LAST** calls.

MAC

The *MAC* parameter is a pointer to a string variable containing the trial MAC. Ensure that this parameter is a pointer to a 9-byte string variable, because 9 bytes are always sent to the security server. The MAC value must be left-aligned in the variable. The verb verifies the MAC if you specify the **ONLY** or **LAST** keyword for the segmenting control.

Required commands

The MAC_Verify verb requires the Verify MAC command (offset X'0011') to be enabled in the active role.

Chapter 7. Key-storage mechanisms

This section describes how you can use key-storage mechanisms and the associated verbs for creating, writing, reading, listing, and deleting records in key storage. The following table lists the verbs in this section. See "Key-storage verbs" on page 243 for detailed information on each verb.

Table 22. Key-storage-record services

Verb	Page	Service	Entry point	Service location
DES_Key_Record_Create	244	Creates a key record in DES key-storage.	CSNBKRC	S
DES_Key_Record_Delete	245	Deletes a key record or deletes the key token from a key record in DES key-storage.	CSNBKRD	S
DES_Key_Record_List	247	Lists the key names of the key records in DES key-storage.	CSNBKRL	S
DES_Key_Record_Read	249	Reads a key token from DES key-storage.	CSNBKRR	S
DES_Key_Record_Write	250	Writes a key token into DES key-storage.	CSNBKRW	S
PKA_Key_Record_Create	251	Creates a record in the public-key key-storage.	CSNDKRC	S
PKA_Key_Record_Delete	253	Deletes a record or deletes the key token from a record in public-key key-storage.	CSNDKRD	S
PKA_Key_Record_List	255	Lists the key names of the records in public-key key-storage.	CSNDKRL	S
PKA_Key_Record_Read	257	Reads a key token from public-key key-storage.	CSNDKRR	S
PKA_Key_Record_Write	259	Writes a key token in public-key key-storage.	CSNDKRW	S
Retained_Key_Delete	261	Deletes a key retained within the cryptographic engine.	CSNDRKD	E
Retained_Key_List	262	Lists the public and private RSA keys retained within the cryptographic engine.	CSNDRKL	E
Service location: E=Cryptographic Engine, S=Security API software				

Key labels and key-storage management

Use the verbs described in this section to manage key storage. The CCA software manages key storage as an indexed repository of key records. Access key storage using a key label.

There are several independent key-storage systems that can be used 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 normally be retained within the coprocessor.

Key storage must be initialized before any records are created. Before a key token can be stored in key storage, a key-storage record must be created using the Key_Record_Create verb.

Use the Key_Record_Delete verb to delete a key token from a key record, or to entirely delete the key record from key storage.

Use the 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, represented by an asterisk (*), 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-aligned in the 64-byte string variable used as input to the verb. Some verbs use a key label while others use a key identifier. Calls that use a key identifier accept either a key token or a key label.

A key-label character string has the following properties:

- If the first character is within the range X'20' through X'FE', the input is treated as a key label, even if it is otherwise not valid. Inputs beginning with a byte valued in the range X'00' through X'1F' are considered to be some form of key token. A first byte valued to X'FF' is not valid.
- The first character of the key label cannot be numeric (0 – 9).
- The label is ended by a space character on the right (in ASCII it is X'20', and in EBCDIC it is X'40'). The remainder of the 64-byte field is padded with space characters.
- Construct a label with one to seven name-tokens, each separated by a period (.). The key label must not end with a period.
- A name-token consists of 1 – 8 characters in the character set A – Z, 0 – 9, and 3 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.

  Notes:

  – Some CCA implementations accept the characters a – z and fold these to their uppercase equivalents, A – Z. Only use the uppercase alphabetic characters.

  – Some implementations internally transform the EBCDIC encoding of a key label to an ASCII string. Also, the label might be put in tokenized form by dropping the periods and formatting each name token into 8-byte groups, padded on the right with space characters.

Some verbs accept a key label containing a wild card represented by an asterisk (*). (X'2A' in ASCII; X'5C' in EBCDIC). When a verb permits the use of a wild card, the wild card can appear as the first character, as the last character, or as the only character in a name token. Any of the name tokens can contain a wild card.

Examples of valid key labels include the following:

```
A
ABCD.2.3.4.5555
ABCDEFGH
BANKSYS.XXXXX.43*.*PDQ
```

Examples of key labels that are not valid 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)
```

Key-storage verbs

DES_Key_Record_Create (CSNBKRC)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The DES_Key_Record_Create verb adds a key record with a null key-token to DES key-storage. It is identified by the key label specified using the *key_label* parameter.

After creating a DES key record, you can use any of the following verbs to add or update a key token in the key record:
- Clear_Key_Import
- DES_Key_Record_Write
- Data_Key_Import
- Key_Generate
- Key_Import
- Key_Part_Import
- Multiple_Clear_Key_Import
- PKA_Symmetric_Key_Import

To delete a DES key record, use the DES_Key_Record_Delete verb.

Restrictions

None

Format

CSNBKRC			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

key_label
The *key_label* parameter is a pointer to a string variable containing the key label of the DES 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 active role.

DES_Key_Record_Delete (CSNBKRD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The DES_Key_Record_Delete verb can do either of the following tasks:
- Replaces the token in a key record with a null key-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 parameter. To identify multiple records, use a wild card (*) in the key label.

Restrictions

None

Format

CSNBKRD			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Task (one required)	
TOKEN-DL	Deletes a key token from a key record in DES key-storage.
LABEL-DL	Deletes an entire key record, including the key label, from DES 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 DES_Key_Record_Delete verb requires the Compute Verification Pattern command (offset X'001D') to be enabled in the active role.

DES_Key_Record_List (CSNBKRL)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The DES_Key_Record_List verb creates a key-record-list data set containing information about specified key records in key storage. Information listed includes whether record validation is correct, the type of key, and the date and time the record was created and last updated.

Specify the key records to be listed using the key-label variable. To identify multiple key records, use the wild card (*) in the key label.

Note: To list all the labels in key storage, specify the key_label parameter with *, *.*, *.*.*, and so forth, up to a maximum of seven name tokens (*.*.*.*.*.*.*).

The verb creates the key-record-list data set and returns the name of the data set 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 368.

AIX users should refer to the *CCA Support Program Installation Manual*, Section 3, “AIX installation instructions” for information concerning the location of the key-record-list directory.

Restrictions

None

Format

CSNBKRL			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

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, 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 number of bytes of data returned by the verb in the *data_set_name* variable. The maximum returned value is 64 bytes.

data_set_name

The *data_set_name* parameter is a pointer to a 64-byte string variable containing the name of the data set returned by the verb. The data set contains the key-record information.

The verb returns the data set name as a fully qualified file specification (for example, *C:\PROGRAM FILES\IBM\4764\KEYS\DESLIST\KYRLTnnn.LST* on a Windows system), where *nnn* is the numeric portion of the name. This value increases by one every time you use this verb. When this value reaches 999, it resets to 001.

Note: When the verb stores a key-record-list data set, it overlays any older data set with the same name.

security_server_name

The *security_server_name* parameter is a pointer to a string variable. The information in this variable is not currently used, but the variable must be declared.

Required commands

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

DES_Key_Record_Read (CSNBKRR)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The DES_Key_Record_Read verb copies a key token from DES key-storage to application storage. The returned key-token can be null.

Restrictions

None

Format

CSNBKRR			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 9.

key_label
The *key_label* parameter is a pointer to a string variable containing the key label of the record to be read from DES key storage.

key_token
The *key_token* parameter is a pointer to a string variable containing the key token read from DES key-storage.

Required commands

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

DES_Key_Record_Write (CSNBKRW)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The DES_Key_Record_Write verb copies an internal DES key-token from application storage into DES key-storage.

Note: 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	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
key_token	Input	String	64 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 9.

key_token
The *key_token* parameter is a pointer to a string variable containing the internal key-token to be written into DES key storage.

key_label
The *key_label* parameter is a pointer to a string variable containing the key label that identifies the record in DES 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 active role.

PKA_Key_Record_Create (CSNDKRC)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Key_Record_Create verb adds a key record to PKA key storage. The new key record can be a null key token or a valid PKA internal or external key token. It is identified by the key label specified with the key_label parameter.

After creating a PKA 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
- PKA_Key_Record_Write

To delete a PKA key record, you must use the PKA_Key_Record_Delete verb.

Restrictions
None

Format

CSNDKRC

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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	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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. This verb does not use keywords.

key_label
The *key_label* parameter is a pointer to a string variable containing the key label of the PKA key-record to be created.

key_token_length
The *key_token_length* parameter is a pointer to an integer variable containing

the number of bytes of data in the *key_token* variable. If the value of the key_token_length variable is zero, a record with a null PKA key-token is created.

key_token
The *key_token* parameter is a pointer to a string variable containing the key token being written to PKA key-storage.

Required commands

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

PKA_Key_Record_Delete (CSNDKRD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The PKA_Key_Record_Delete verb can do either of the following tasks:
- Replaces the token in a key record with a null key-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* parameter. To identify multiple records, use a wild card (*) in the key label.

Restrictions

None

Format

CSNDKRD			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0 or 1
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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 or 1 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Task (one, optional)	
TOKEN-DL	Deletes a key token from a key record in PKA key storage. This is the default.
LABEL-DL	Deletes an entire key record, including the key label, from PKA 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 PKA key-storage. Use a wild card (*) in the *key_label* variable 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 active role.

PKA_Key_Record_List (CSNDKRL)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Key_Record_List verb creates a key-record-list data set containing information about specified key records in PKA key-storage. Information includes whether record validation is correct, the type of key, and the dates and times when the record was created and last updated.

Specify the key records to be listed using the key_label variable. To identify multiple key records, use the wild card (*) in a key label.

Note: To list all the labels in key storage, specify the key_label parameter with *, *.*, *.*.*, and so forth, up to a maximum of seven name tokens (*.*.*.*.*.*.*).

The verb creates the list 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 368.

AIX users should refer to the *CCA Support Program Installation Manual*, Section 3, AIX installation instructions for information concerning the location of the key record list directory.

Restrictions

None

Format

CSNDKRL

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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	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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. This verb does not use keywords.

key_label
The *key_label* parameter is a pointer to a string variable containing a key record in PKA key-storage. You can use a wild card (*) to identify multiple records in key storage.

data_set_name_length
The *data_set_name_length* parameter is a pointer to an integer variable containing the number of bytes of data returned in the *data_set_name* variable. The maximum returned value is 64 bytes.

data_set_name
The *data_set_name* parameter is a pointer to a 64-byte string variable containing the name of the data set returned by the verb. The data set contains the key-record information.

The verb returns the data set name as a fully qualified file specification (for example, *C:\PROGRAM FILES\IBM\4764\KEYS\PKALIST\KYRLTnnn.LST* on a Windows system), 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 with the same name.

security_server_name
The *security_server_name* parameter is a pointer to a string variable. The information in this variable is not currently used, but the variable must be declared.

Required commands

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

PKA_Key_Record_Read (CSNDKRR)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The PKA_Key_Record_Read verb copies a key token from PKA key storage to application storage.

The returned key-token can be null. In this event, the *key_length* variable contains a value of 8 and the key-token variable contains 8 bytes of X'00' beginning at offset 0 (see “Null key token” on page 348).

Restrictions

None

Format

CSNDKRR			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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	64 bytes
key_token_length	In/Output	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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. This verb does not use keywords.

key_label
The *key_label* parameter is a pointer to a string variable containing the key label of the record to be read from PKA key storage.

key_token_length
The *key_token_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *key_token* variable. The maximum size is 2500 bytes.

key_token
The *key_token* parameter is a pointer to a string variable containing the key token read from PKA key-storage. This variable must be large enough to hold

the PKA key token being read. On successful completion, the *key_token_length* variable contains the actual length of the token being returned.

Required commands

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

PKA_Key_Record_Write (CSNDKRW)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The PKA_Key_Record_Write verb copies a PKA key token from application storage into PKA key storage.

The verb can perform the following processing options:
- Write the new key-token only if the old token was null
- Write the new key-token regardless of content of the old token

Note: 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	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0 or 1
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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 or 1 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Processing option (one, optional)	
CHECK	Specifies that the record is written only if a record of the same label in PKA key-storage contains a null key token. This is the default.
OVERLAY	Specifies that the record is overwritten regardless of the current content of the record in PKA key storage.

key_label

The *key_label* parameter is a pointer to a string variable containing the key label that identifies the key record in PKA 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 number of bytes of data in the *key_token* variable. The maximum size is 2500 bytes.

key_token

The *key_token* parameter is a pointer to a string variable containing the PKA key token to be written into PKA key storage.

Required commands

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

Retained_Key_Delete (CSNDRKD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Retained_Key_Delete verb deletes a PKA key that has been retained within the coprocessor.

You can retain both public and private keys within the coprocessor using verbs such as PKA_Key_Generate and PKA_Public_Key_Register. A list of retained keys can be obtained using the Retained_Key_List verb.

Restrictions

None

Format

CSNDRKD

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 or null pointer
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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 for this verb.

rule_array
The *rule_array* parameter should be a null address pointer.

key_label
The *key_label* parameter points to a 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 active role.

Retained_Key_List (CSNDRKL)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Retained_Key_List verb lists the key labels of those PKA keys that have been retained within the coprocessor. You filter the set of key labels returned to your application using the *key label mask input* variable.

Specify the keys to be listed using the *key_label_mask* variable. To identify multiple keys, use a wild card (*) in a mask. Only labels with matching characters to those in the mask up to the first "*" is returned. To list all retained key labels, specify a mask of an *, followed by 63 space characters. For example, if the coprocessor has retained key-labels a.a, a.a1, a.b.c.d, and z.a, and you specify the mask a.*, the verb returns a.a, a.a1 and a.b.c.d. If you specify a mask of a.a*, the verb returns a.a and a.a1.

You can retain both public and private keys within the coprocessor using verbs such as PKA_Key_Generate and PKA_Public_Key_Register. You can delete retained keys using the Retained_Key_Delete verb.

Restrictions

None

Format

CSNDRKL			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	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 or null pointer
key_label_mask	Input	String	64 bytes or 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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 for this verb.

rule_array
The *rule_array* parameter should be a null address pointer.

key_label_mask
The *key_label_mask* parameter points to a 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.

retained_keys_count
The *retained_keys_count* parameter points to an integer variable to receive the total 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.

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 Names command (offset X'0230') to be enabled in the active role.

Chapter 8. Financial services support

Several classes of verbs are described in this section:

- Verbs that process finance industry PINs. Information common to these verbs is described in the next section.
- Verbs that can change the acceptable PIN on a smart card based on Visa and EMV design concepts.
- SET-related verbs that 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.
- Transaction validation verbs that compute and validate codes for MasterCard, Visa, and American Express.

Table 23 lists the verbs described in this section. See "Financial services support verbs" on page 277 for a detailed description of each verb.

Table 23. Financial services support verbs

Verb	Page	Service	Entry point	Service location
Clear_PIN_Encrypt	278	Formats a PIN into a PIN block and produces 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 produced in encrypted PIN blocks.	CSNBCPE	E
Clear_PIN_Generate	281	Generates a clear PIN, or a PIN offset.	CSNBPGN	E
Clear_PIN_Generate_Alternate	284	Extracts a customer-selected PIN or institution-assigned PIN from an encrypted PIN-block and generates a PIN offset.	CSNBCPA	E
CVV_Generate	289	Generates a card-verification value according to the Visa CVV and MasterCard CVC rules for track 2.	CSNBCSG	E
CVV_Verify	292	Verifies a card-verification value according to the Visa CVV and MasterCard CVC rules for track 2.	CSNBCSV	E
Encrypted_PIN_Generate	295	Generates a PIN from an account number and other information and returns the result in an encrypted PIN-block.	CSNBEPG	E
Encrypted_PIN_Translate	299	Operates in two modes: • Translate mode reencrypts a PIN block under a different key. • Reformat mode can do the following: – Reformats a PIN from one PIN-block format into another PIN-block format – Changes selected non-PIN digits in a PIN block – Reencrypts a PIN block	CSNBPTR	E
Encrypted_PIN_Verify	304	Extracts and verifies a PIN by using the specified PIN-calculation method.	CSNBPVR	E
PIN_Change/Unblock	310	Calculates a PIN for a smart card based on keys and data you supply according to Visa and EMV specifications.	CSNBPCU	E

Table 23. Financial services support verbs (continued)

Verb	Page	Service	Entry point	Service location
Secure_Messaging_for_Keys	317	Securely incorporates a key into a text block that is then encrypted (for use with EMV smart cards).	CSNBSKY	E
Secure_Messaging_for_PINs	320	Securely incorporates a PIN block into a text block that is then encrypted (for use with EMV smart cards).	CSNBSPN	E
SET_Block_Compose	324	Creates a SET-protocol RSA-OAEP block that DES encrypts.	CSNDSBC	E
SET_Block_Decompose	327	Decomposes the RSA-OAEP block that DES decrypts.	CSNDSBD	E
Transaction_Validation	331	Generates and verifies American Express Card Security Codes (CSC).	CSNBTRV	E
E=Cryptographic Engine				

Processing financial PINs

This section describes how the financial personal identification number (PIN) verbs allow you to process financial PINs.

Note: In this section, automated teller machine (ATM) can also mean a point-of-sale device, an enhanced teller terminal, or a programmable workstation.

A financial PIN is used to authorize personal financial transactions for a customer who uses an automated teller machine or point-of-sale device. 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. With the financial PIN verbs you can specify PINs that range from 4 – 16 digits in length. (A financial PIN is usually 4 digits in length.)

The financial PIN verbs form a complete set of verbs that you can use in various combinations to process financial PINs. The verb relationships and primary inputs and outputs are depicted in Figure 10 on page 268. You use these verbs for the following tasks:

- Provide security for the PINs by using 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.
  - Reencrypting a PIN-block under another key with optional, integral changing of the PIN-block format
- Use multiple PIN-calculation methods
- Use multiple PIN-block formats and PIN-extraction methods
- Use ANSI X9.24 derived unique-key-per-transaction (UKPT) PIN-block encryption
- 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 trial PIN into an ATM to identify himself. The ATM does the following:

1. Obtains account information and other information from the magnetic stripe on the card.
2. Formats the trial PIN into a PIN block and encrypts the PIN block.
3. Sends the information from the card, the encrypted PIN block, and other data in a message to a host program for verification. In some applications, a program in the machine can use a verb to verify a clear PIN locally.

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, recalculates the account-number-based PIN, adjusts this value with any *offset*, compares the resulting value to the trial PIN, and returns the result 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 database and that is used in the PIN-verification process.

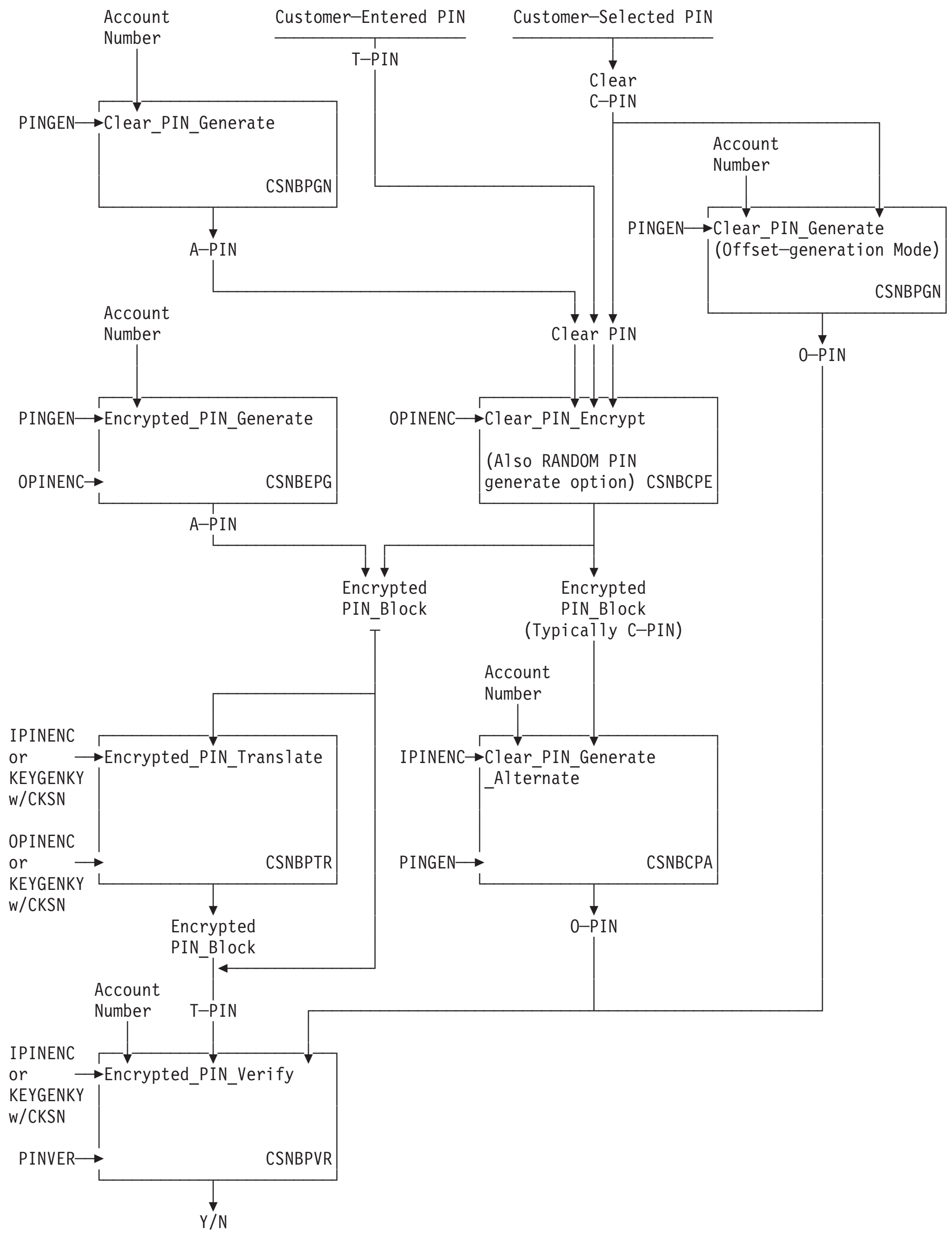


Figure 10. Financial PIN verbs

PIN-verb summary

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.

The *Clear_PIN_Generate* verb 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 427. Certain calculation methods also accept a C-PIN value and return an O-PIN calculated from the coprocessor-generated A-PIN value.

The *Encrypted_PIN_Generate* verb 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 accepts a clear 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 wants to distribute initial PIN values to customers.

The *Clear_PIN_Generate_Alternate* verb accepts an encrypted PIN block that contains 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 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 must 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 is indicated by the return code (and reason code) values. Return code 0 indicates the PIN is validated while return code 4 indicates that the trial PIN failed validation.

The *Encrypted_PIN_Translate* verb 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 reencryption. 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 calculation methods and a variety of PIN-block formats with the various PIN-processing verbs. Table 24 provides a summary of the supported combinations.

Table 24. PIN verb, PIN-calculation method, and PIN-block-format support summary

Verb / Calculation method, PIN block	Entry point	UKPT	IBM-PIN	IBM-PINO	GBP-PIN	INBK-PIN	NL-PIN-1	Visa-PVV	3624	ISO-0 ISO-1	ISO-2
Clear_PIN_Encrypt	CSNBCPE								X	X	X
Clear_PIN_Generate	CSNBPGN		X	X							
Clear_PIN_Generate_Alternate	CSNBCPA			X			X	X	X	X	
Encrypted_PIN_Generate	CSNBEPG		X		X	X			X	X	X
Encrypted_PIN_Translate	CSNBPTR	X							X	X	X
Encrypted_PIN_Verify	CSNBPVR	X	X	X	X	X		X	X	X	X

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 431.

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.

The following verbs receive clear PINs from your application program or return clear PINs to your program. None of the other PIN verbs reveals 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 that are 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 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 PINs. You can also further restrict those verbs in which a PINGEN key performs by selectively turning off bits in the default PINGEN control vector. See Appendix C, “CCA control-vector definitions and key encryption,” on page 385.

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.

Point-of-sale terminals that accept a customer’s PIN often use the UKPT mechanism specified in ANSI X9.24 to ensure that tampering with the device does not reveal keys used to encrypt previous PIN encryptions. With the Encrypted_PIN_Translate and Encrypted_PIN_Verify verbs, you can process PIN blocks encrypted according to ANSI X9.24. In these cases you supply the base key and a current-key serial number (CKSN). The verb derives the appropriate key to decrypt or encrypt the PIN block when a single-DEA method is specified. If a triple-DEA method is specified, it employs the algorithm specified in ANS X9.52-1997.

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 use a key with a PINGEN key type if control-vector 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. The PINVER key-type cannot 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.

KEYGENKY (UKPT base key-generating key) key type
The Encrypted_PIN_Translate and Encrypted_PIN_Verify verbs can derive a unique key from the KEYGENKY derivation key and current-key-serial-number to decrypt or encrypt a PIN block.

Supporting multiple PIN-calculation methods

With the PIN verbs you can use 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 relation to an account number. The methods are described in “PIN-calculation methods” on page 427. The PIN verbs can use the following PIN-calculation methods, which you specify with a keyword in the *rule_array* variable for a verb:

PIN-calculation method	Keyword
IBM German Bank Pool PIN	**GBP-PIN**
IBM 3624 PIN	**IBM-PIN**
IBM 3624 PIN Offset	**IBM-PINO**
Interbank PIN	**INBK-PIN**
Netherlands PIN-1	**NL-PIN-1**
Visa PIN-Validation Value (PVV)	**Visa-PVV**

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-aligned, 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, and GBP-PIN: When using the IBM-PIN, the IBM-PINO, the NL-PIN-1, and GBP-PIN methods, the data array contains elements for a decimalization table and validation data. For some verbs, it also includes a clear PIN or an offset to a clear PIN.

- **decimalization_table**

  The first element in the data array for a PIN-calculation method points to the decimalization table, which is 16 characters. The characters 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, do not include a decimal digit that is also used as a pad digit in the decimalization table. For information about using a pad digit, see "PIN profile" on page 273.
- **validation_data**

  The second element in the data array for a PIN-calculation method supplies 1 – 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. While normally the validation data consists of numeric-decimal characters, the Clear_PIN_Generate_Alternate, Encrypted_PIN_Generate, and Encrypted_PIN_Verify verbs have been updated to accept any hexadecimal character (0 - 9, A - F).
- **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 this should be 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" on page 276.
  - A one-digit key index selector value from 1 – 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 storage. The information in these elements is 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” on page 276.
  – A constant, 6.
  – A one-digit key index selector value from 1 – 6.
  – Three numeric characters of validation data.
- **reserved**

  The second and third elements in the data array for the Interbank calculation method are reserved. These elements point to 16-byte variables in application storage. The information in these elements is ignored, but the elements must be declared.

Supporting multiple PIN-block formats and PIN-extraction methods

The PIN verbs use multiple PIN-block formats, which you specify in a PIN_profile variable. The PIN-block formats are described in “PIN-block formats” on page 431. 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), or a sequence number.

When deriving the unique key according to the ANSI X9.24 UKPT process, the verbs also require you to supply the current key serial number (CKSN). The CKSN is supplied as an extension of the PIN profile.

PIN profile

A PIN-profile variable consists of three elements and an optional extension, the CKSN. The basic elements identify the PIN-block format, the level of format control, and any pad digit. Generally, you can code the basic 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-aligned or right-aligned and padded with space characters. Depending on the verb and the PIN-block format, all three elements might not be used. However, all three elements (that is, all 24 bytes) must be declared.

PIN-block format: The PIN-block format is the first element in a PIN-profile variable. You specify the format using one of these keywords, left aligned:

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**
EMV-PIN-change	**VISAPCU1** **VISAPCU2**

Format control enforcement: The format-control level is the second element in a PIN profile. For the IBM 4758 and IBM 4764, 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 discussed in Table 25 on page 274. The Pad Digit for PIN Formatting column indicates the values that the verb uses when it creates a PIN block. The Pad Digit for 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 - 9 and A - F. The pad digit must be uppercase, right-aligned in the 8-byte element, with 7 preceding space characters. When a pad digit is not required, specify 8 space characters.

Note: For the IBM 3621 PIN-block format, the pad digit should be a non-decimal character (from C'A' - 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, 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 272.

Table 25. Pad-digit specification by PIN-block format

PIN-block format	Pad digit for PIN formatting	Pad digit for PIN extraction
3624	0 through F	0 through F
ISO-0	F	The pad-digit specification is ignored.
ISO-1	The pad-digit specification is ignored.	The pad-digit specification is ignored.
ISO-2	The pad-digit specification is ignored.	The pad-digit specification is ignored.
EMV-PIN-change	The pad-digit specification is ignored.	The pad-digit specification is ignored.

Current key serial number: When a PIN block is encrypted with a derived, unique key, the PIN profile variable is extended by 24 bytes. The CKSN is left aligned within the extension and padded by 4bytes of X'00'.

The CKSN is the concatenation of a terminal identifier and a sequence number which, together, define a unique terminal (within the set of terminals associated with a given base key) and the sequence number of the transaction originated by that terminal. Each time the terminal completes a transaction, it increments the sequence number and modifies the transaction-encryption key retained within the terminal. The key-modification process is a one-way function so that tampering with the terminal does not reveal previously used keys. For details of this process, see "Unique-key-per-transaction calculation methods" on page 434.

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 can use multiple PIN-extraction methods. The valid PIN-extraction method depends 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* variable for the verb.

Table 26 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.

Table 26. PIN-extraction method keywords by PIN-block format

PIN-block format	PIN-extraction method keywords (used in the rule array)
3624	**HEXDIGIT** **PADDIGIT** **PADEXIST** **PINLEN04** – **PINLEN16**
ISO-0	**PINBLOCK**
ISO-1	**PINBLOCK**
ISO-2	**PINBLOCK**

The PIN-extraction method keywords operate in the following way:

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 is to use the pad value in the PIN profile to identify the end of the PIN.

HEXDIGIT
This keyword specifies that the verb is to 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 is to use the length specified in the keyword, where *xx* can range from 4 – 16 digits, to identify the PIN.

PADEXIST
This keyword specifies that the verb is to use the character in the sixth position of the PIN block as the value of the pad value.

Personal account number

A personal account number (PAN) identifies an individual and associates that individual to an account at the financial institution. The PAN consists of the following data:
- Issuer identification number
- Customer account number
- One check digit

For the ISO-0 or Visa-4 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 from 0 - 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.

Generating and verifying Visa and MasterCard card-verification values

Both Visa and MasterCard employ the same process for creating a credit-card verification value. Visa calls this a card-verification value (CVV). MasterCard calls this a card validation code (CVC). IBM uses the term CVV generically.

The CVV value can be encoded in track 2 or might be printed, but not embossed, on a credit card. (Look for a 3, 4, or 5 character code that might be printed in reverse italics in the card's signature panel following the card number on the back of a credit card.) This code is sometimes requested in a card-not-present transaction.

CCA can generate and verify a CVV using the CVV_Generate (CSNBCSG) and CVV_Verify (CSNBCSV) verbs.

The process used to generate a CVV is described in "CVV and CVC Method" on page 436.

Working with Europay–MasterCard–Visa smart cards

There are several verbs and verb capabilities you can use in secure communications with EMV smart cards. The processing capabilities are consistent with the specifications provided in these documents:
- *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2*
- *Design Visa Integrated Circuit Card Specification Manual*

EMV smart cards include the following processing capabilities:
- The Diversified_Key_Generate verb (see "Diversified_Key_Generate (CSNBDKG)" on page 157) with rule-array options **TDES-XOR**, **TDESEMV2**, and **TDESEMV4** enable you to derive a key used to cipher and authenticate messages, and more particularly message parts, for exchange with an EMV smart card. You use the derived key with verbs such as Encipher, Decipher, MAC_Generate, MAC_Verify, Secure_Messaging_for_Keys, and

Secure_Messaging_for_PINs. These message parts can be combined with message parts created using the Secure_Messaging_for_Keys and Secure_Messaging_for_PINs verbs.

- The Secure_Messaging_for_Keys verb (see "Secure_Messaging_for_Keys (CSNBSKY)" on page 317) enables you to securely incorporate a key into a message part (generally the value portion of a TLV component of a secure message for a card). Similarly, the Secure_Messaging_for_PINs verb (see "Secure_Messaging_for_PINs (CSNBSPN)" on page 320) enables secure incorporation of a PIN block into a message part.
- The PIN_Change/Unblock verb (see "PIN_Change/Unblock (CSNBPCU)" on page 310) enables you to encrypt a new PIN to send to a new EMV card, or to update the PIN value on an initialized EMV card. This verb generates the required session key.
- The **ZERO-PAD** option of the PKA_Encrypt verb (see "PKA_Encrypt (CSNDPKE)" on page 206) enables you to validate a digital signature created according to ISO 9796-2 standard by encrypting information you format, including a hash value of the message to be validated. You compare the resulting enciphered data to the digital signature accompanying the message to be validated.
- The MAC_Generate and MAC_Verify verbs post-pad a X'80'...X'00' string to a message as required for authenticating messages exchanged with EMV smart cards.

Financial services support verbs

Clear_PIN_Encrypt (CSNBCPE)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 431):
- 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 data:
- 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 273.
- 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. Specify a value of 99999 in the integer variable.

The verb performs the following tasks:
- Formats the PIN into the specified PIN-block format.
- Checks the control vector for the OPINENC key by verifying 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
None

Format

CSNBCPE			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
PIN_encrypting_key_identifier	Input	String	64 bytes
rule_array_count	Input	Integer	0 or 1
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 9.

PIN_encrypting_key_identifier
The *PIN_encrypting_key_identifier* parameter points to a string containing an internal key token or a key label of an internal key token. The internal key token contains the key that encrypts the PIN block. The control vector in the internal key token must specify an OPINENC key type and have the CPINENC bit set to 1.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0 or 1 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
PIN source (one, optional)	
ENCRYPT	Causes the verb to use the PIN value contained in the clear_PIN variable. This is the default.
RANDOM	Causes the verb to use a randomly generated PIN value. The length of the PIN is based on the value in the clear_PIN variable. Value the clear PIN to 0 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 string variable containing the clear PIN. and either the offset or the Visa PIN-validation value (PVV). The values in this variable must be left-aligned and padded on the right with space characters.

PIN_profile
The *PIN_profile* parameter points to a string variable 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 273.

PAN_data
The *PAN_data* parameter points to a personal account number (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 area in application storage. The information in this variable is ignored, but the variable must be declared.

sequence_number
The *sequence_number* parameter points to an integer. Specify a value of 99999.

encrypted_PIN_block
The *encrypted_PIN_block* parameter points to a string variable containing the encrypted PIN-block returned by the verb.

Required commands

The Clear_PIN_Encrypt verb requires the Format and Encrypt PIN command (offset X'00AF') to be enabled in the active role.

Clear_PIN_Generate (CSNBPGN)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Clear_PIN_Generate verb generates an A-PIN or an O-PIN using one of the following PIN-calculation methods that you specify with a rule-array keyword (see “PIN-calculation methods” on page 427):
- IBM 3624 PIN (**IBM-PIN**)
- IBM 3624 PIN Offset (**IBM-PINO**)

You can use this verb to perform the following tasks:
- 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 this data:
- 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 (for example, 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 performs the following tasks:
1. Verifies that the CPINGEN bit is set to 1 in the control vector for the PINGEN key.
2. 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 427.
3. Uses the specified PIN length to determine the length of the PIN.
4. Returns the clear A-PIN or O-PIN in the variable identified by the returned_result parameter.

Restrictions

None

Format

CSNBPGN			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
PIN_generating_key_identifier	Input	String	64 bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
PIN_length	Input	Integer	
PIN_check_length	Input	Integer	
data_array	Input	String array	3 * 16 bytes
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 9.

PIN_generating_key_identifier
The *PIN_generating_key_identifier* parameter points to a string variable containing an 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 bit set to 1.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
PIN-calculation method (one required)	
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 variable from 4 – 16 containing the length of the PIN.

PIN_check_length
The *PIN_check_length* parameter points to an integer variable from 4 – 16 containing 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 storage. The information in this variable is 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 a string variable containing 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 1 – 16 bytes in length, left-aligned, and padded on the right with space characters. The verb converts the space characters to zeros.

When using the **IBM-PIN** and **IBM-PINO** keywords, identify the following elements in the data array:

Element	Description
decimalization_table	This element contains the decimalization table of 16 characters (0 – 9) that are used to convert the hexadecimal digits (X'0' – X'F') of the encrypted validation data to decimal digits (X'0' – X'9').
validation_data	This 16-byte element contains 1 – 16 characters of account data. The data must be left-aligned and padded on the right with spaces.
clear_PIN	When using the **IBM-PINO** keyword, this 16-byte element contains the clear customer-selected PIN. This value must be left-aligned and padded on the right with spaces.

returned_result

The *returned_result* parameter points to a string variable containing the result returned by the verb. The result is left-aligned and padded on the right with space characters.

Required commands

The Clear_PIN_Generate verb requires the following command to be enabled in the active role, 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)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The Clear_PIN_Generate_Alternate verb is used to obtain a value, the O-PIN (either as an offset or VISA-PVV), that relates the institution-assigned PIN to the customer-known PIN. The verb uses these PIN-calculation methods:
- IBM 3624 PIN Offset (**IBM-PINO**)
- Netherlands PIN-1 (**NL-PIN-1**)
- Visa PIN Validation Value (**VISA-PVV**)

You supply the customer PIN, known as a C-PIN as an encrypted PIN-block. The verb performs the following functions:
- Decrypts a PIN block
- Extracts a customer-selected or institution-assigned PIN
- Generates an A-PIN from the input account number, PIN-generating key, and so forth
- 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 the following data:
- 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 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
- A decimalization table and account validation data
- A 16-byte variable for the O-PIN

The verb performs the following processing:
1. Checks the control vector of the IPINENC key to ensure that the CPINGENA bit is 1.
2. Decrypts the PIN block in ECB mode.
3. 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 input_PIN_profile variable or the PAN specified in the *PAN_data* variable.
4. Verifies that the CPINGENA bit is 1 in the control vector for the PINGEN key.
5. 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.
6. Calculates the O-PIN.

7. Returns the clear O-PIN in the variable identified by the returned_result parameter.

Restrictions

None

Format

CSNBCPA

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	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	3 * 8 bytes
PAN_data	Input	String	12 bytes
encrypted_PIN_block	Input	String	8 bytes
rule_array_count	Input	Integer	1 or 2
rule_array	Input	String array	rule_array_count * 8 bytes
PIN_check_length	Input	Integer	
data_array	Input	String array	3 * 16 bytes
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 9.

inbound_PIN_encrypting_key_identifier
The *inbound_PIN_encrypting_key_identifier* parameter points to a string variable containing an 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 string variable containing an 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 string variable containing a character array with three 8-byte elements: the PIN-block format keyword, the format control (**NONE**), a pad digit, if needed. See “PIN profile” on page 273 for more information.

PAN_data
The *PAN_data* parameter points to a string variable containing personal account number (PAN) data. If the PIN profile specifies the **ISO-0** keyword, 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 to a string variable containing the encrypted PIN-block of the customer-selected C-PIN value.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

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 indicates the PIN-calculation method, as shown in Table 27.

Table 27. Clear_PIN_Generate_Alternate rule_array keywords (first element)

PIN-calculation method	Meaning
IBM-PINO	This keyword specifies that the IBM 3624 PIN Offset calculation method is to be used.
NL-PIN-1	This keyword specifies that the Netherlands PIN-1 calculation method is to be used.
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 Table 28. For more information about extraction methods, see "PIN-extraction methods" on page 275.

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 one, the PIN-extraction method keyword that is listed first in the following table is the default.

Table 28. Clear_PIN_Generate_Alternate rule_array keywords (second element)

PIN-block format keyword	PIN-extraction method keyword	Meaning
3624	**PADDIGIT** **HEXDIGIT** **PINLEN04** – **PINLEN16** **PADEXIST**	These 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.

Table 28. Clear_PIN_Generate_Alternate rule_array keywords (second element) (continued)

PIN-block format keyword	PIN-extraction method keyword	Meaning
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 variable from 4 - 16 containing 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 storage. The information in this variable is 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 is 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 a string variable containing 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 48-byte data array.

When using the **IBM-PINO** keyword, identify the following elements in the data array as shown in the following table:

Element	Description
decimalization_table	This element contains the decimalization table of 16 characters (0 – 9) that are used to convert the hexadecimal digits (X'0' – X'F') of the enciphered validation data to decimal digits (X'0' – X'9').
validation_data	This 16-byte element contains 1 – 16 characters of account data. The data must be left-aligned and padded on the right with space characters.
reserved_3	The information in this element is ignored, but the 16-byte 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 contains the characters 0 – 9, A – F.
validation_data	This 16-byte element contains 1 – 16 characters of account data. The data must be left-aligned and padded on the right with space characters.
reserved_3	The information in this element is ignored, but the 16-byte 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, see "Visa PIN validation value calculation method" on page 430.

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 – 6 • Four space characters for padding
reserved_2	The information in this element is ignored, but the 16-byte element must be declared.
reserved_3	The information in this element is ignored, but the 16-byte element must be declared.

returned_result

The *returned_result* parameter points to a string variable containing the clear O-PIN returned by the verb. The 16-byte result is be left-aligned and padded on the right with space characters.

The length of the PIN offset in the returned result is determined by the value that the PIN_check_length parameter specifies.

Required commands

The Clear_PIN_Generate_Alternate verb requires the commands shown in the following table to be enabled in the active role based on the keyword specified for the PIN-calculation 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

CVV_Generate (CSNBCSG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

The CVV_Generate verb generates credit-card verification codes as defined by the Visa card-verification value (CVV) and the MasterCard card-verification code (CVC). This document uses the generic term CVV. The CVV can be found on the magnetic stripe on track 2, or printed but not embossed on a card.

The verb generates a CVV that is based on the information you specify with the PAN_data, the expiration_date, and the service_code parameters. The verb uses the key-A and key-B keys to cryptographically process this information. For details about the CVV process, see "CVV and CVC Method" on page 436.

You specify the following information:

- The length of the personal account number (PAN) through a rule-array keyword.
- The PAN data in a 16-byte variable for a 13-character or 16-character PAN, or in a 19-byte variable for a 19-character PAN.
- Two single-length keys, key-A and key-B. Specify either a MAC-class key type or a DATA-class key type. The subtype extension bit field (bits 0 – 3) in the control vectors can be B'0000'. Alternatively, you can ensure that the keys are used only in the CVV_Generate and CVV_Verify verbs by specifying a MAC-class key with subtype extension bits for key-A as B'0010' and for key-B as B'0011'. The control vectors for these keys must also indicate generation capability by having bit 20 set enabled. For more information about control vectors, see Appendix C, "CCA control-vector definitions and key encryption."

The verb returns the 5-byte variable specified by the CVV_value parameter. If the requested CVV is shorter than 5 characters, the CVV is padded on the right by space characters.

Format

CSNBCSG			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0, 1, or 2
rule_array	Input	String array	rule_array_count * 8 bytes
PAN_data	Input	String	16 or 19 bytes
expiration_date	Input	String	4 bytes
service_code	Input	String	3 bytes
CVV_key-A_identifier	Input	String	64 bytes
CVV_key-B_identifier	Input	String	64 bytes
CVV_value	Output	String	5 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 9.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0, 1, or 2 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The *rule_array* keywords are shown in the following table:

Keyword	Meaning
PAN-data length (one, optional)	
PAN-13	This keyword specifies that the length of the PAN data is 13 bytes. This is the default.
PAN-16	This keyword specifies that the length of the PAN data is 16 bytes.
PAN-19	This keyword specifies that the length of the PAN data is 19 bytes.
CVV length (one, optional)	
CVV-1	This keyword specifies the length of the CVV to be returned is 1 character. This is the default.
CVV-2	This keyword specifies the length of the CVV to be returned is 2 characters.
CVV-3	This keyword specifies the length of the CVV to be returned is 3 characters.
CVV-4	This keyword specifies the length of the CVV to be returned is 4 characters.
CVV-5	This keyword specifies the length of the CVV to be returned is 5 characters.

PAN_data
The *PAN_data* parameter points to a string variable containing PAN data in character format. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If you specify either the **PAN-13** or **PAN-16** keyword, 16 bytes are sent to the coprocessor. A 13-character PAN must be left aligned in the 16-byte variable. If you specify the **PAN-19** keyword, 19 bytes are sent to the coprocessor.

expiration_date
The *expiration_date* parameter points to a string variable containing the card expiration date. The date is in numeric character format. You must determine whether the CVV is calculated as YYMM or MMYY.

service_code
The *service_code* parameter points to a string variable containing the service code in character format. The service code is the number that the track-2 magnetic-stripe standards define.

CVV_key-A_identifier
The *CVV_key-A_identifier* parameter points to a string variable containing an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the key-A key that encrypts information in the CVV process.

CVV_key-B_identifier
The *CVV_key-B_identifier* parameter points a string variable containing an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the key-B key that decrypts information in the CVV process.

CVV_value
The *CVV_value* parameter points to a string variable containing the CVV value in character format returned by the verb.

Required commands

The CVV_Generate verb requires the Generate CVV command (offset X'00DF') to be enabled in the active role.

CVV_Verify (CSNBCSV)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The CVV_Verify verb verifies credit-card verification codes as defined by the Visa card-verification value (CVV) and the MasterCard card-verification code (CVC). This data can be found on the magnetic stripe on track 2, and printed but not embossed on a card. This document uses the generic term CVV.

The verb generates a CVV value internal to the coprocessor based on the information you specify with the PAN_data, the expiration_date, and the service_code parameters. The verb uses the key-A and key-B keys to cryptographically process this information. The internal value is compared to that which you provide and the result is returned to you in a return-code value. For details about the CVV process, see "CVV and CVC Method" on page 436.

You specify the following information:
- The length of the personal account number (PAN) through a rule-array keyword.
- The PAN data in a 16-byte variable for a 13-character or 16-character PAN, or in a 19-byte variable for a 19-character PAN.
- Two single-length keys, key-A and key-B. Specify either a MAC-class key type or a DATA-class key type. The subtype extension bit field (bits 0 - 3) in the control vectors can be B'0000'. Alternatively, you can ensure that the keys are used only in the CVV_Generate and CVV_Verify verbs by specifying a MAC-class key with subtype extension bits for key-A as B'0010' and for key-B as B'0011'. The control vectors for these keys must also indicate verification capability by having bit 21 set on. For more information about control vectors, see Appendix C, "CCA control-vector definitions and key encryption."
- The expected CVV value in a 5-byte variable, left aligned and padded with space characters.

Based on your use of the **CVV-***n* rule-array keywords, the internal CVV value is truncated to fewer characters and padded on the right with space characters. The internal CVV value is then compared to the five-character value that you specify with the CVV_value parameter. The result of this comparison is indicated in the return code.

The verb returns the results of the comparison in the return-code variable. If the return code is 0, the values correctly compared. If the CVV values do not match, the return code is set to 4 (and the reason code is set to 1).

Format

CSNBCSV			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0, 1, or 2
rule_array	Input	String array	rule_array_count * 8 bytes
PAN_data	Input	String	16 or 19 bytes
expiration_date	Input	String	4 bytes
service_code	Input	String	3 bytes
CVV_key-A_identifier	Input	String	64 bytes
CVV_key-B_identifier	Input	String	64 bytes
CVV_value	Input	String	5 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 9.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0, 1, or 2 for this verb.

rule_array

The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
PAN-data length (one, optional)	
PAN-13	This keyword specifies that the length of the PAN data is 13 bytes. This is the default.
PAN-16	This keyword specifies that the length of the PAN data is 16 bytes.
PAN-19	This keyword specifies that the length of the PAN data is 19 bytes.
CVV length (one, optional)	
CVV-1	This keyword specifies the length of the CVV to be verified is 1 character. This is the default.
CVV-2	This keyword specifies the length of the CVV to be verified is 2 characters.
CVV-3	This keyword specifies the length of the CVV to be verified is 3 characters.
CVV-4	This keyword specifies the length of the CVV to be verified is 4 characters.
CVV-5	This keyword specifies the length of the CVV to be verified is 5 characters.

PAN_data

The *PAN_data* parameter points to a string variable containing the PAN data in character format. The PAN is the account number as defined for the track-2 magnetic-stripe standards. If you specify either the **PAN-13** or **PAN-16** keyword, 16 bytes are sent to the coprocessor. A 13-character PAN must be left aligned in the 16-byte variable. If you specify the **PAN-19** keyword, 19 bytes are sent to the coprocessor.

expiration_date

The *expiration_date* parameter points to a string variable containing the card expiration date. The date is in numeric character format. You must determine whether the CVV is calculated as YYMM or MMYY.

service_code

The *service_code* parameter points to a string variable containing the service code in character format. The service code is the number that the track-2, magnetic-stripe standards define.

CVV_key-A_identifier

The *CVV_key-A_identifier* parameter points to a string variable containing an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the key-A key that encrypts information in the CVV process.

CVV_key-B_identifier

The *CVV_key-B_identifier* parameter points to a string variable containing an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the key-B key that decrypts information in the CVV process.

CVV_value

The *CVV_value* parameter points to a string variable containing the CVV value in character format.

Required commands

The CVV_Verify verb requires the Verify CVV command (command offset X'00E0') to be enabled in the active role.

Encrypted_PIN_Generate (CSNBEPG)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20 3.23	

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 3624 PIN
- IBM German Bank Pool Institution PIN
- Interbank PIN

To format the PIN, the verb uses one of the following PIN-block formats:
- IBM 3624
- ISO-0 (same as ANSI X9.8, VISA-1, and ECI-1 formats)
- ISO-1 (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 information:
- 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.
- The length of the PIN for those PIN-calculation methods with variable-length PINs in the PIN_length variable. Otherwise, the variable should be 0.
- 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 427. For information about the data-array variable, see “Data_array” on page 272.
- 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 431.
- One of the following with the PAN_data parameter:
  – When using the ISO-0 PIN-block format, specify a PAN. For information about a PAN, see “Personal account number” on page 276.
  – When using another PIN-block format, specify a 12-byte area in application storage. The information in the variable is 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 performs the following tasks:

- 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 431.
- 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

None

Format

CSNBEPG			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	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	1
rule_array	Input	String array	rule_array_count * 8 bytes
PIN_length	Input	Integer	
data_array	Input	String	16 bytes * 3
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 9.

PIN_generating_key_identifier

The *PIN_generating_key_identifier* parameter points to a string variable containing an 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 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 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 is a pointer to an integer variable containing the number of elements in the rule_array variable. This value must be 1 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Table 29. Encrypted_PIN_Generate rule_array keywords

Keyword	Meaning
Calculation method (one required)	
IBM-PIN	This keyword specifies the IBM 3624 PIN-calculation method to be used to generate a PIN.
GBP-PIN	This keyword specifies the IBM German Bank Pool Institution 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 is a pointer to an integer variable containing the PIN length for those PIN-calculation methods with variable-length PINs. Otherwise, the variable should be valued to 0.

data_array
The *data_array* parameter is a pointer to a string variable containing 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, see “Data_array” on page 272.

The numeric characters in each 16-byte string must be from 1 - 16 bytes in length, uppercase, left-aligned, and padded on the right with space characters. The verb converts the space characters to zeros.

PIN_profile
The *PIN_profile* parameter is a pointer to a string variable containing the PIN profile including the PIN-block format. See “PIN profile” on page 273.

PAN_data
The *PAN_data* parameter is a pointer to a string variable containing 12 digits of PAN data. The verb uses this parameter if the PIN profile specifies **ISO-0** for the PIN-block format. Otherwise, ensure that this parameter is a pointer to a 4-byte area in application 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 is a pointer to a string variable containing the sequence number used by certain PIN-block formats. Ensure that this parameter is a pointer to a 4-byte area in application storage.

encrypted_PIN_block
The *encrypted_PIN_block* parameter is a pointer to a string variable containing the encrypted PIN-block returned by the verb.

Required commands

The Encrypted_PIN_Generate verb requires the commands, as shown in the following table, to be enabled in the active role based on the keyword specified for the PIN-calculation methods.

PIN-Calculation Method	Command Offset	Command
IBM-PIN	X'00B0'	Generate Formatted and Encrypted 3624 PIN
GBP-PIN	X'00B1'	Generate Formatted and Encrypted German Bank Pool PIN
INBK-PIN	X'00B2'	Generate Formatted and Encrypted Interbank PIN

Encrypted_PIN_Translate (CSNBPTR)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The Encrypted_PIN_Translate verb changes PIN block encryption and, optionally, formats a PIN into a different PIN-block format. You can use this verb in an interchange-network application, or to change the PIN block to conform to the format and encryption key used in a PIN-verification database. With this verb you can also use derived unique key per transaction (UKPT) PIN-block encryption (ANSI X9.24) for both input and output PIN blocks.

You can specify these 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

The verb operates in one of two modes:
- In *translate* mode the verb decrypts a PIN block using an input key that you supply, or that is derived from other information that you supply. The clear text information is then encrypted using an output key that you supply, or that is derived from other information that you supply. The clear text is not examined.
- In *reformat* mode the verb performs the translate-mode functions and, in addition, processes the clear text information. Following rules that you specify, the PIN is recovered from the input cleartext PIN-block and formatted into an output PIN-block for encryption.

To use this verb, specify the following information:
- The mode of operation with a keyword in the rule array: **TRANSLAT** or **REFORMAT**.
- Optionally, the method of PIN extraction with a rule-array keyword.
- | Optionally, UKPT option on input or output with rule-array keywords: **UKPTIPIN**,
  | **UKPTOPIN**, or **UKPTBOTH** for single-DEA method, or **DUKPT-IP**, **DUKPT-OP**,
  | or **DUKPT-BH** for triple-DEA method.
- Input and output PIN-block encrypting keys, or the base keys used to derive the PIN-block enciphering keys.
- Input and output PIN profiles, which for UKPT processing are extended with the current key serial number (CKSN). See "PIN profile" on page 273 and "Unique-key-per-transaction calculation methods" on page 434.
- Input and output PAN data as required by the selected PIN-block formats.
- An output PIN-block sequence number. Specify a value of 99999.

The verb does the following:
- Decrypts the input PIN-block by using the supplied IPINENC key in ECB mode, or derives the decryption key using the specified KEYGENKY key and current key serial number, and then uses ANSI X9.24-specified special decryption or
  | triple-DEA method.
- Checks the control vector to ensure that for an IPINENC key that the TRANSLAT bit is set to 1 for translate mode and the REFORMAT bit is set to 1 for reformat

mode, or for a KEYGENKY key that the UKPT bit is set to 1. Likewise the OPINENC key must have one or both of the TRANSLAT and REFORMAT bits set according to the requested mode.

- In reformat mode, the verb performs these additional steps:
  - Extracts the PIN from the specified PIN-block format using the method specified by default or by a rule-array keyword. If required by the PIN-block format, PAN data is 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.
- Encrypts the output PIN-block by using the supplied OPINENC key in ECB mode, or derives the decryption key using the specified KEYGENKY key and output current key serial number and uses ANSI X9.24-specified special encryption or triple-DEA method. The TRANSLAT bit must be set to 1 in the OPINENC control vector, or the UKPT bit must be set to 1 in the KEYGENKY control vector.

Restrictions

Some CCA implementations might enforce a specific order of the rule array keywords with this verb. See the product-specific literature for more information.

Format

CSNBPTR

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	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	24 or 48 bytes
input_PAN_data	Input	String	12 bytes
input_PIN_block	Input	String	8 bytes
rule_array_count	Input	Integer	1, 2, or 3
rule_array	Input	String array	rule_array_count * 8 bytes
output_PIN_profile	Input	String array	24 or 48 bytes
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 9.

input_PIN_encrypting_key_identifier

The *input_PIN_encrypting_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.

If you do not use the UKPT process, 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 and have one or both of the TRANSLAT and REFORMAT bits set to 1 as appropriate for the requested mode.

If you use the UKPT process for the input PIN-block, specify the base derivation key as a KEYGENKY key-type with the UKPT bit set to 1.

output_PIN_encrypting_key_identifier
The *output_PIN_encrypting_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.

If you do not use the UKPT process, the internal key token must contain 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 OPINENC key type and have one or both of the TRANSLAT and REFORMAT bits set to 1 as appropriate for the requested mode.

If you use the UKPT process for the output PIN-block, specify the base derivation key as a KEYGENKY key-type with the UKPT bit set to 1.

input_PIN_profile
The *input_PIN_profile* parameter is a pointer to a string variable containing three 8-byte character strings with information defining the PIN-block format, and, optionally, an additional 24 bytes containing the input current key serial number. The strings are equivalent to 24-byte or 48-byte strings.

input_PAN_data
The *input_PAN_data* parameter is a pointer to a string variable containing the 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 *input_PIN_block* parameter is a pointer to a string variable containing the encrypted PIN-block.

rule_array_count
The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the *rule_array* variable. This value must be 1, 2, or 3 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

Keyword	Meaning
Mode (one required)	
TRANSLAT	This keyword specifies that only PIN-block encryption is changed.

Keyword	Meaning
REFORMAT	This keyword specifies that either or both the PIN-block format and the PIN-block encryption are to be changed. If the PIN-extraction method is not chosen by default, another element in the rule array must specify one of the keywords that indicates a PIN-extraction method as listed in Table 30. For more information about extraction methods, see "PIN-extraction methods" on page 275.
Unique key per transaction (one, optional)	
UKPTIPIN	Specifies the use of UKPT input-key derivation and PIN-block decryption, single-DEA method.
UKPTOPIN	Specifies the use of UKPT output-key derivation and PIN-block encryption, single-DEA method.
UKPTBOTH	Specifies the use of UKPT key-derivation and PIN-block ciphering for both input and output processing, single-DEA method.
DUKPT-IP	Specifies the use of UKPT input-key derivation and PIN-block decryption, triple-DEA method.
DUKPT-OP	Specifies the use of UKPT output-key derivation and PIN-block encryption, triple-DEA method.
DUKPT-BH	Specifies the use of UKPT key-derivation and PIN-block ciphering for both input and output processing, triple-DEA method.
PIN-extraction method (one, optional) See Table 30.	

Table 30. Encrypted_PIN_Translate rule_array keywords

PIN-block format	PIN-extraction method	Meaning
PIN-extraction method (one, optional) **Note:** You specify the PIN-block format keyword in the PIN profile variable.		
3624	**HEXDIGIT** **PADDIGIT** **PADEXIST** **PINLEN04** – **PINLEN16**	These keywords specify a PIN-extraction method for an IBM 3624 PIN-block format. **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.

Table 30. Encrypted_PIN_Translate rule_array keywords (continued)

PIN-block format	PIN-extraction method	Meaning
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 is a pointer to a string variable containing three 8-byte character strings with information defining the PIN-block format, and, optionally, an additional 24 bytes containing the output current key serial number (CKSN). The strings are equivalent to 24-byte or 48-byte strings.

output_PAN_data
The *output_PAN_data* parameter is a pointer to a string variable containing the personal account number (PAN) data. If you specify the **REFORMAT** keyword, and if the output PIN-profile specifies the ISO-0 or the VISA-4 PIN-block format, the verb uses this data to format the output PIN-block. Otherwise, ensure that this parameter points to a 12-byte area in application 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 is a pointer to an integer variable containing the sequence number. Ensure that this parameter is a pointer to an integer variable valued to 99999.

output_PIN_block
The *output_PIN_block* parameter is a pointer to a string variable containing the reenciphered and, optionally, reformatted PIN-block returned by the verb.

Required commands

The Encrypted_PIN_Translate verb requires the commands shown in Table 31 to be enabled in the active role 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.

Table 31. Encrypted_PIN_Translate required hardware commands

Mode	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

The Encrypted_PIN_Translate verb also requires the Unique Key per Transaction, ANSI X9.24 command (offset X’00E1’) to be enabled if you employ UKPT processing.

Encrypted_PIN_Verify (CSNBPVR)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 account PIN (A-PIN) calculated by using the specified PIN-calculation method. Certain PIN-calculation methods modify the value of the A-PIN with the clear offset (O-PIN) value prior to the comparison. The verb also supports derived unique key per transaction (UKPT) PIN-block encryption (ANSI X9.24) for decrypting the input PIN block.

You can specify the 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 use this verb, specify:
- Processing choices using rule-array keywords:
  - A PIN-calculation method.
  - Optionally, a PIN-extraction method.
  - | Optionally, unique-key-per-transaction (UKPT) processing with the **UKPTIPIN** | keyword for single-DEA method or **DUKPT-IP** keyword for triple-DEA method.
- An input PIN-block decrypting key, or the base key used to derive the PIN-block enciphering key.
- A PIN-verifying key to be used to calculate the PIN.
- A PIN profile for the input PIN-block, which for UKPT processing must be extended with the current key sequence number (CKSN). See “PIN profile” on page 273.
- When using the ISO-0 block format, a PAN to be used in extracting the PIN. See “Personal account number” on page 276.
- The PIN block that contains the PIN to be verified.
- The length of the PIN to be checked if you specify the **IBM-PIN** or the **IBM-PINO** calculation methods in the rule array.
- In the data array: a decimalization table, account validation data, and for certain calculation methods, an offset value.

The verb does the following:
- Decrypts the input PIN-block by using the supplied IPINENC key in ECB mode, or derives the decryption key using the specified KEYGENKY key and CKSN and | uses ANSI X9.24-specified special decryption or triple-DEA method. The EPINVER bit must be valued to 1 in the IPINENC control vector, or the UKPT bit must be valued to 1 in the KEYGENKY control vector. See “PIN profile” on page 273 and “Unique-key-per-transaction calculation methods” on page 434.
- Extracts the trial PIN (T-PIN) from the specified PIN-block format using the method specified by default or by a rule array keyword. If required by the PIN-block format, PAN data or the pad digit is used in the extraction process.

- Verifies use of a PINVER or PINGEN key type having the EPINVER bit valued to one in the control vector of the PIN-verifying key
- Calculates the account-number-based PIN (A-PIN).
- For methods that employ an offset, modifies the A-PIN value with the offset (O-PIN) value entered in the third element of the data array variable. The NOOFFSET bit must be valued to zero in the control vector of the PIN-verifying key when employing the IBM 3624 PIN Offset calculation method.
- Compares the extracted trial (T-PIN) with the possibly modified account PIN (A-PIN) and reports the results in the return code variable. Return code 4 indicates a verification failure, while return code 0 indicates success.

Restrictions

Some CCA implementations might enforce a specific order of the rule array keywords with this verb. See the product-specific literature for more information.

Format

CSNBPVR			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	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	24 or 48 bytes
PAN_data	Input	String	12 bytes
encrypted_PIN_block	Input	String	8 bytes
rule_array_count	Input	Integer	1, 2, or 3
rule_array	Input	String array	rule_array_count * 8 bytes
PIN_check_length	Input	Integer	
data_array	Input	String array	3 * 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 9.

PIN_encrypting_key_identifier

The *PIN_encrypting_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.

If you do not use the UKPT process, 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 EPINVER bit valued to one.

If you use the UKPT process for the input PIN-block, specify the base derivation key as a KEYGENKY key-type with the UKPT bit valued to one.

PIN_verifying_key_identifier

The *PIN_verifying_key_identifier* parameter points to a string variable containing an internal key-token or a key label of an internal key-token record in key storage. The internal key-token contains the key used to generate the account-number-based PIN (A-PIN). The control vector in the internal key-token must specify a PINVER or PINGEN key-type. For a PINGEN (and PINVER) key, the EPINVER bit must be one.

PIN_profile

The *PIN_profile* parameter is a pointer to a string variable containing three 8-byte character strings with information defining the PIN-block format, and optionally an additional 24 bytes containing the input current key serial number (CKSN). The strings are equivalent to 24-byte or 48-byte strings. For more information about a PIN profile, see "PIN profile" on page 273.

PAN_data

The *PAN_data* parameter is a pointer to a string variable containing the personal account number (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 is a pointer to a 12-byte variable in application 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 points to a string variable containing the encrypted PIN-block.

rule_array_count

The *rule_array_count* parameter is a pointer to an integer variable containing the number of elements in the rule_array variable. The value must be 1, 2, or 3 for this verb.

rule_array

The *rule_array* parameter points to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

Keyword	Meaning
Calculation method (one required)	
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 PIN-calculation method is to be used.
VISAPVV4	This keyword specifies that the VISA-PVV PIN-calculation method is to be used. Acceptable PINs must be exactly four digits in length.
INBK-PIN	This keyword specifies that the Interbank PIN-calculation method is to be used.
Unique key per transaction (one, optional)	

Keyword	Meaning
UKPTIPIN	Specifies the use of UKPT input-key derivation and PIN-block decryption, single-DEA method.
DUKPT-IP	Specifies the use of UKPT input-key derivation and PIN-block decryption, triple-DEA method.
PIN-extraction method (one, optional) See Table 32.	

Table 32. Encrypted_PIN_Verify PIN-extraction method

PIN-block format	PIN-extraction method	Meaning
3624	**HEXDIGIT** **PADDIGIT** **PADEXIST** **PINLEN04 – PINLEN16**	The PIN-extraction method keywords specify a PIN-extraction method for an IBM 3624 PIN-block format. **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 is a pointer to an integer variable containing 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 storage.

Note: The PIN check length must be less than or equal to the PIN length and in the range from 4 – 16.

data_array

The *data_array* parameter is a pointer to a string variable containing three 16-byte character strings, which are equivalent to a single 48-byte string. The values you specify in the data array depend on the PIN-calculation method. Each element is not always used, but you must always declare a complete 48-byte 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 – 9) that are used to convert the hexadecimal digits (X'0' – X'F') of the encrypted validation data to decimal digits (X'0' – X'9').
validation_data	This 16-byte element contains 1 – 16 characters of account data. The data must be left-aligned and padded on the right with space characters. (To conform with industry practice, any hexadecimal character can be specified.)
offset data	When using the **IBM-PINO** keyword, this 16-byte element contains the offset data that must be left-aligned and padded on the right 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** or **VISAPVV4** keywords, 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 calculation method" on page 430.

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 – 6 • Four space characters
PVV (O-PIN)	This 16-byte element contains four numeric characters, which are the referenced PVV value. This value is followed by 12 space characters.
reserved_3	The information in this element is ignored, but the 16-byte 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 430.

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 one-digit key index selector from 1 – 6 • Three numeric characters of validation data
reserved_2	The information in this element is ignored, but the 16-byte element must be declared.
reserved_3	The information in this element is ignored, but the 16-byte element must be declared.

Required commands

The Encrypted_PIN_Verify verb requires the following commands to be enabled in the active role, 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, **VISAPVV4**	X'00AD'	Verify Encrypted VISA-PVV
INBK-PIN	X'00AE'	Verify Encrypted Interbank PIN
NL-PIN-1	X'0232'	Verify Encrypted NL-PIN-1

The Encrypted_PIN_Verify verb also requires the Unique Key per Transaction, ANSI X9.24 command (offset X'00E1') to be enabled in the active role if you employ UKPT processing.

PIN_Change/Unblock (CSNBPCU)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

Use the PIN_Change/Unblock verb to prepare an encrypted message-portion for communicating an original or replacement PIN for an EMV smart-card. The verb embeds the PINs in an encrypted PIN-block from information that you supply. You incorporate the information created with the verb in a message sent to the smart card.

The processing is consistent with the specifications provided in these documents:
- *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2*
- *Design Visa Integrated Circuit Card Specification Manual*.

You specify the following information:
- Through the optional choice of one rule-array keyword, the key-diversification process to employ in deriving the session key used to encrypt the PIN block. See "Visa and EMV-related smart card formats and processes" on page 437 for processing details.

TDES-XOR	An exclusive-OR process described in the appendix. It is the default process.
TDESEMV2	The tree-based-diversification process with a branch factor of 2.
TDESEMV4	The tree-based-diversification process with a branch factor of 4.

- Through the required choice of one rule-array keyword, if you are providing a PIN for a smart card:

VISAPCU1	For a card *without* a PIN, provide the new PIN in an encrypted PIN-block in the *new_reference_PIN_block* variable. The contents of *current_reference_PIN...* variables are ignored.
VISAPCU2	For a card *with* a current PIN, provide the existing PIN in an encrypted PIN-block in the *current_reference_PIN_block* variable, and supply the new PIN-value in an encrypted PIN-block in the *new_reference_PIN_block* variable.

- Issuer-provided master-derivation keys (MDK). The card-issuer provides two keys for diversifying the same data:
  - The MAC-MDK key that you incorporate in the variable specified by the authentication_key_identifier parameter. The verb uses this key to derive an authentication value incorporated in the PIN block. The control vector for the MAC-MDK key must specify a DKYGENKY key type with DKYL0 (level-0), and DMAC or DALL permissions. See Figure 24 on page 389.
  - The ENC-MDK key that you incorporate in the variable specified by the encryption_key_identifier parameter. The verb uses this key to derive the PIN-block encryption key. The control vector for the ENC-MDK key must specify a DKYGENKY key type with DKYL0 (level-0), and DMPIN or DALL permissions.

  See "Visa and EMV-related smart card formats and processes" on page 437, which explains the derivation processes and PIN-block formation.

- The diversification_data_length to indicate the sum of the lengths of:
  - Data, 8 or 16 bytes, encrypted by the verb using the MDK keys.
  - The 2-byte application transfer counter (ATC). You receive the ATC value from the EMV smart card.
  - The optional 16-byte initial value used in the **TDESEMV**n processes.

  Valid lengths are 10, 18, 26, and 34 bytes.
- The diversification_data variable. Concatenate the 8-byte or 16-byte data, the ATC, and optionally the Initial Value.

  The 16-bit ATC counter is processed as a two-byte string, not as an integer value.
- The new-reference PIN in an encrypted PIN block. You provide:
  - The key to decrypt the PIN block
  - The PIN block
  - The format information that defines how to parse the PIN block
  - When using an ISO-0 format PIN block, personal-account number (PAN) information to enable PIN recovery from the ISO-0 format PIN block.
- If you specified **VISAPCU2** (because the target smart card already has a PIN), the current_reference_PIN in an encrypted PIN block with the associated decrypting key, PIN-block format, and PAN data. In any case, you must declare current_reference_PIN... variables.
- The output_PIN_message variable to receive the encrypted PIN block for the smart card, and the length in bytes of the PIN block (16). The PIN-block format you specify (**VISAPCU1** or **VISAPCU2**) corresponds to the one or two PIN values to be communicated to the smart card.
- You must declare two variables which are reserved for future use: output_PIN_data_length (valued to zero), and an output_PIN_data string variable (or set the associated parameter to a null pointer).

The PIN_Change/Unblock verb:

- Decrypts the MDK keys and verifies the required control vector permissions.
- Diversifies the left-most eight bytes of data using the MAC-MDK key to obtain the authentication value for placement into the PIN block.
- Recovers the supplied PIN values provided that PIN-block encrypting keys are one of IPINENC or OPINENC type, and the use of the specific type is authorized with the appropriate access-control command.
- Constructs and pads the output PIN block to a 16-byte string. See “Constructing the PIN-block for transporting an EMV smart-card PIN” on page 438.
- Generates the session key used to encrypt the output-PIN block using the ENC-MDK, the key_generation_data, the ATC counter value, and the optional Initial Value.
- Triple encrypts the 16-byte padded PIN-block in ECB mode.
- Returns the encrypted, padded PIN-block in the output_PIN_message variable.

Restrictions

This verb is supported beginning with Release 2.50. Support for the **TDESEMV2** and **TDESEMV4** keywords begins with Release 2.51.

Format

CSNBPCU

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1 or 2
rule_array	Input	String array	rule_array_count * 8 bytes
authentication_key_identifier_length	Input	Integer	64
authentication_key_identifier	Input	String	
encryption_key_identifier_length	Input	Integer	64
encryption_key_identifier	Input	String	
diversification_data_length	Input	Integer	10, 18, 26, or 34
diversification_data	Input	String	diversification_data_length bytes
new_reference_PIN_key_identifier_length	Input	Integer	64
new_reference_PIN_key_identifier	Input	String	64 bytes
new_reference_PIN_block	Input	String	8 bytes
new_reference_PIN_profile	Input	String	3*8 bytes
new_reference_PIN_PAN_data	Input	String	12 bytes
current_reference_PIN_key_identifier_length	Input	Integer	64
current_reference_PIN_key_identifier	Input	String	64 bytes
current_reference_PIN_block	Input	String	8 bytes
current_reference_PIN_profile	Input	String	3*8 bytes
current_reference_PIN_PAN_data	Input	String	12 bytes
output_PIN_data_length	Input	Integer	0
output_PIN_data	Input	String	Can be null
output_PIN_profile	Input	String	3*8 bytes
output_PIN_message_length	In/Output	Integer	16
output_PIN_message	Output	String	output_PIN_message_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 9.

rule_array_count

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

rule_array

The *rule_array* parameter points to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

Keyword	Meaning
Diversification process (one, optional)	
TDES-XOR	This keyword specifies to diversify the issuer-master-key using triple DES and an exclusive-OR process. This is the default process.
TDESEMV2	This keyword specifies to diversify the issuer-master-key using the EMV tree-based function, branch factor 2. See EMV 4.0 Book 2, Annex A1.3.1, and "Visa and EMV-related smart card formats and processes" on page 437.
TDESEMV4	This keyword specifies to diversify the issuer-master-key using the EMV tree-based function, branch factor 4.
Output PIN creation process (one required)	
VISAPCU1	This keyword specifies to create the output PIN from the new-reference PIN and the smart-card-unique, intermediate key.
VISAPCU2	This keyword specifies to create the output PIN from the new-reference PIN and the smart-card-unique, intermediate key, and the current-reference PIN.

authentication_key_identifier_length
The *authentication_key_identifier_length* parameter points to an integer variable set to 64. This is the string length of the related key identifier.

authentication_key_identifier
The *authentication_key_identifier* parameter points to a string variable containing an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the MAC-MDK key used to diversify the data to form the authentication value. The control vector for this key must specify a DKYGENKY key type with DKYL0 (level-0), and DMAC or DALL permissions. Both halves of this double-length key must be unique. See Figure 24 on page 389.

encryption_key_identifier_length
The *encryption_key_identifier_length* parameter points to an integer variable set to 64. This is the string length of the related key identifier.

encryption_key_identifier
The *encryption_key_identifier* parameter points to a string variable containing an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the ENC-MDK key used to diversify the data to form the output PIN-block encryption key. The control vector for this key must specify a DKYGENKY key type with DKYL0 (level-0), and DMPIN or DALL permissions. Both halves of this double-length key must be unique.

diversification_data_length
The *diversification_data_length* parameter points to an integer set to the byte-length of the data used in the generation of the authentication value and the PIN-block encryption key. With the **TDES-XOR** keyword use a length of 10 or 18. With the **TDESEMV2** and **TDESEMV4** keywords use a length of 10, 18, 26 or 34.

diversification_data
The *diversification_data* parameter points to a string variable. Form the variable by concatenating two or three values:
- The first 8 or 16 bytes of data contains the value used to form the smart-card-specific authentication value and the PIN-block encryption key.
- The next two bytes of data contain the 16-bit ATC counter used to further diversify the ENC-MDK key to form the session key used to encrypt the output PIN block. The high-order counter bit is in the left-most counter byte.
- When using the **TDESEMV2** or **TDESEMV4** tree-based diversification process, you can concatenate an optional 16-byte Initial Value. (Otherwise the verb substitutes 16 bytes of X'00'.)

new_reference_PIN_key_identifier_length
The *new_reference_PIN_key_identifier_length* parameter points to an integer variable set to 64. This is the string length of the related key identifier.

new_reference_PIN_key_identifier
The *new_reference_PIN_key_identifier* parameter points to a string variable containing an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the key used to decrypt the new_reference_PIN_block. The control vector for this key must specify either an IPINENC or an OPINENC key type.

new_reference_PIN_block
The *new_reference_PIN_block* parameter points to an 8-byte string variable containing an encrypted PIN block which in turn contains the new_reference_PIN.

new_reference_PIN_profile
The *new_reference_PIN_profile* parameter points to an array of three, 8-byte string variables which define the new_reference_PIN_block format. For more information about a PIN profile, see "PIN profile" on page 273.

new_reference_PIN_PAN_data
The *new_reference_PIN_PAN_data* parameter points to a 12-byte string variable containing the PAN data. PAN data is used to recover a PIN from an ISO-0 PIN block. If the PIN block is not in ISO-0 format, this value is ignored, but a 12-byte area must be specified.

current_reference_PIN_key_identifier_length
The *current_reference_PIN_key_identifier_length* parameter points to an integer variable set to 0 or 64, providing the length in bytes of the *current_reference_PIN_key_identifier* variable. If the **VISAPCU2** keyword is used, a key must be specified and this variable must be 64, else 0.

current_reference_PIN_key_identifier
The *current_reference_PIN_key_identifier* parameter points to a string variable. The contents of this variable are inspected when the **VISAPCU2** rule-array keyword is present. The variable contains an internal key token or a key label of an internal key-token record in key storage. The internal key token contains the key used to decrypt the current_reference_PIN_block. The control vector for this key must specify either an IPINENC or an OPINENC key type.

current_reference_PIN_block
The *current_reference_PIN_block* parameter points to an 8-byte string variable. The contents of this variable are inspected when the **VISAPCU2** rule-array keyword is present. The variable contains an encrypted PIN-block which in turn contains the current_reference_PIN.

current_reference_PIN_profile
The *current_reference_PIN_profile* parameter points to an array of three, 8-byte string variables. The contents of the variables are inspected when the **VISAPCU2** rule-array keyword is present. The variables define which PIN-block format is processed.

current_reference_PIN_PAN_data
The *current_reference_PIN_PAN_data* parameter points to a 12-byte string variable. The variable contains the PAN data. PAN data is used to recover a PIN from an ISO-0 PIN block. The contents of this variable are inspected when the **VISAPCU2** rule-array keyword is present and the PIN-profile specifies an ISO-0 PIN-block format.

output_PIN_data_length
The *output_PIN_data_length* parameter points to an integer, which must be set to 0.

output_PIN_data
The *output_PIN_data* parameter points to a string variable which can be a null pointer.

output_PIN_profile
The *output_PIN_profile* parameter points to an array of three, 8-byte string variables. The variables define which PIN-block format is processed. The variables should be set to these values:

- As per the rule array selection, the string 'VISAPCU1' or 'VISAPCU2'.
- Format control set to 'NONE' (followed by four space characters).
- Eight space characters.

For more information about a PIN profile, see "PIN profile" on page 273.

output_PIN_message_length
The *output_PIN_message_length* parameter points to an integer containing the length in bytes of the output_PIN_message variable. Set this variable to at least a value of at least 16 on input. On a successful response, the verb returns a value of 16 which is the length of the output_PIN_message.

output_PIN_message
The *output_PIN_message* parameter points to a 16-byte string variable to receive the output encrypted, padded PIN-block.

Required commands

The PIN_Change/Unblock verb requires the following commands to be enabled in the active role based on the permissible key-type, IPINENC or OPINENC, used in the decryption of the input PIN blocks.

PIN-block encrypting key-type	Command offset	Command	Comment
OPINENC	X'00BC'	Generate PIN Change using OPINENC	Required if either the new_reference_PIN_key or the current_reference_PIN_key are permitted to be an OPINENC key type.
IPINENC	X'00BD'	Generate PIN Change using IPINENC	Required if either the new_reference_PIN_key or the current_reference_PIN_key are permitted to be an IPINENC key type.

When a MAC-MDK or an ENC-MDK of key type DKYGENKY is specified with control vector bits (19 – 22) of B’1111’, the Generate Diversified Key (DALL with DKYGENKY Key Type) command (offset X'0290') must also be enabled in the active role.

Secure_Messaging_for_Keys (CSNBSKY)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

Use the Secure_Messaging_for_Keys verb to decrypt a key you supply for incorporation into a text block you also supply. The text block is then encrypted within the verb to preserve the security of the key value. The encrypted text block, normally the **value** field in a TLV item, can be incorporated into a message sent to an EMV smart card.

The processing is consistent with the specifications provided in these documents:
- *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2*
- *Design Visa Integrated Circuit Card Specification Manual*.

Specify the following information:
- Whether the text block is to be CBC or ECB encrypted.
- The input_key to be included within the encrypted text block. The input_key can be an internal key (encrypted under the master key), or an external key, in which case you also provide the IMPORTER or EXPORTER key required to decipher the input_key. You also specify the length of this key using the *key_field_length* variable.
- The key to encipher the secure message text block, the secmsg_key_identifier.
- The clear_text to be encrypted along with its length and the offset within the block for placement of the decrypted input_key_identifier. The text you supply must be a multiple of eight bytes.

  You also supply the encryption initialization_vector and the variable for receiving the initialization vector for encrypting additional message text. The supplied text is a portion of a larger message you are preparing for an EMV smart card. The encrypted text must be on an 8-byte boundary within the complete message. The initialization_vector is the encrypted eight bytes just before the text prepared within this verb.
- The variable to receive the enciphered_text.

The Secure_Messaging_for_Keys verb performs the following tasks:
- Recovers the input key.
- Places the deciphered input-key value within the supplied text at the specified offset.
- Encrypts the supplied text. In CBC mode, uses the supplied initialization_vector and also returns the value to be supplied as the initialization vector when enciphering subsequent data for the EMV card message using the output_chaining_vector.
- Returns the enciphered text.

Restrictions
None

Format

CSNBSKY

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0 or 1
rule_array	Input	String array	rule_array_count * 8 bytes
input_key_identifier	Input	String	64 bytes
key_encrypting_key_identifier	Input	String	64 bytes
secmsg_key_identifier	Input	String	64 bytes
clear_text_length	Input	Integer	Multiple of 8, ≤ 4096
clear_text	Input	String	clear_text_length bytes
initialization_vector	Input	String	8 bytes
key_offset	Input	Integer	(0 is at the start of the clear_text)
key_field_length	Input	Integer	key length, such as 8 or 16
enciphered_text	Output	String	clear_text_length bytes
output_chaining_vector	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 9.

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

rule_array
The *rule_array* parameter points to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

Keyword	Meaning
Enciphering mode (one, optional)	
TDES-CBC	Use CBC mode to encipher the clear_text. This is the default.
TDES-ECB	Use ECB mode to encipher the clear_text.

input_key_identifier
The *input_key_identifier* parameter is a pointer to a string variable containing the encrypted-key key-token or the label of a key record in key storage. You can identify any type of key, provided the control-vector export-allowed permission bit is on (bit 17). You can also specify an external DATA key with an all-zero control vector.

key_encrypting_key_identifier
The *key_encrypting_key_identifier* parameter is a pointer to a string variable containing the IMPORTER or EXPORTER key to decipher an external input_key_identifier. You can also specify a key label of a key storage record for such a key. For an internal-form input key, you can specify a null key-token.

secmsg_key_identifier
The *secmsg_key_identifier* parameter is a pointer to a string variable containing an internal key token or the key label of an internal key token in key storage. The control vector must specify a SECMSG type key with the SMKEY control-vector bit (bit 18) on.

clear_text_length
The *clear_text_length* parameter is a pointer to an integer containing the length of text in bytes to be encrypted. This must be a multiple of 8 and less than or equal to 4096.

clear_text
The *clear_text* parameter is a pointer to the text string to be updated and encrypted.

initialization_vector
The *initialization_vector* parameter is a pointer to an 8-byte string containing the CBC-encryption initialization vector. The data to be exclusive-ORed with the first eight bytes of the message. This can be a null pointer when ECB mode is specified.

key_offset
The *key_offset* parameter is a pointer to an integer containing the offset of the location for the decrypted input key. The start of the text is an offset of 0. The offset plus the key-offset-field-length must be less than or equal to the clear-text length.

key_field_length
The *key_field_length* parameter is a pointer to an integer containing the length of key information to be inserted into the text message. Use a length of 8 for a single-length key and a length of 16 for a double-length key.

enciphered_text
The *enciphered_text* parameter is a pointer to a string variable to receive the enciphered text message.

output_chaining_vector
The *output_chaining_vector* parameter is a pointer to an 8-byte string to receive the CBC chaining value. This is the same as the last eight bytes of returned text and is used as an initialization_vector when encrypting subsequent data for a message. This can be a null pointer when ECB mode is specified.

Required commands

The Secure_Messaging_for_Keys verb requires the Secure Messaging for Keys command (offset X'0273') to be enabled in the active role.

Secure_Messaging_for_PINs (CSNBSPN)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

Use the Secure_Messaging_for_PINs verb to decrypt an input PIN-block, reformat the PIN-block, and incorporate the PIN-block into a text block you supply. The text block is then encrypted within the verb to preserve the security of the PIN value. The encrypted text block, normally the *value* field in a Tag, Length, Value (TLV) item, can be incorporated into a message sent to an EMV smart card. TLV is defined in ISO 7816-4.

The processing is consistent with the specifications provided in these documents:
- *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2*
- *Design Visa Integrated Circuit Card Specification Manual*.

Specify the following information:
- Whether the text block is to be CBC or ECB encrypted.
- Whether the PIN block is to be self-encrypted.
- The encrypted input_PIN_block.

  The key to decrypt the input_PIN_block.

  The PIN profile for the input_PIN_block.

  When the PIN profile specifies an ISO-0 PIN-block format, the PAN data to recover the PIN.
- The key to encipher the secure message text block, the secmsg_key_identifier.
- The PIN profile for the PIN-block included within the output message.

  When the PIN profile specifies an ISO-0 PIN-block format, the PAN data to obscure the PIN.
- The clear_text to be encrypted along with its length and the offset within the block for placement of the PIN block. The text you supply must be a multiple of eight bytes.

  You also supply the encryption initialization_vector and the variable for receiving the initialization vector for encrypting additional message text. The supplied text is a portion of a larger message you are preparing for an EMV smart card. The encrypted text must be on an 8-byte boundary within the complete message. The initialization_vector is the encrypted 8 bytes just before the text prepared within this verb.
- The variable to receive the enciphered_text.

  The variable to receive a copy of the last 8 bytes of enciphered text. This can be used as an initialization vector for enciphering subsequent message text.

The Secure_Messaging_for_PINs verb performs the following tasks:
- Deciphers the input PIN block.
- Reformats the PIN block when the input and output PIN-block formats differ.
- Self-encrypts the output PIN block as specified.
- Places the PIN block within the supplied text at the specified offset.

- Encrypts the updated text. In CBC mode, uses the supplied initialization_vector and also returns the value to be supplied as the initialization vector when enciphering subsequent data for the EMV card message (the output_chaining_vector).
- Returns the enciphered text.

Restrictions

None

Format

CSNBSPN			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	0, 1, or 2
rule_array	Input	String array	rule_array_count * 8 bytes
input_PIN_block	Input	String	8 bytes
PIN_encrypting_key_identifier	Input	String	64 bytes
input_PIN_profile	Input	String	3 * 8 bytes
input_PAN_data	Input	String	12 bytes
secmsg_key_identifier	Input	String	64 bytes
output_PIN_profile	Input	String	3 * 8 bytes
output_PAN_data	Input	String	12 bytes
clear_text_length	Input	Integer	multiple of 8, ≤ 4096
clear_text	Input	String	clear_text_length bytes
initialization_vector	Input	String	8 bytes
PIN_offset	Input	Integer	(0 is at the start of the clear_text)
PIN_offset_field_length	Input	Integer	8 bytes
enciphered_text	Output	String	clear_text_length bytes
output_chaining_vector	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 9.

rule_array_count

The *rule_array_count* parameter points to an integer variable containing the number of elements in the *rule_array* variable. The value must be 0, 1, or 2.

rule_array

The *rule_array* parameter points to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters.

Keyword	Meaning
Enciphering mode (one, optional)	
TDES-CBC	Use CBC mode to encipher the clear_text. This is the default.
TDES-ECB	Use ECB mode to encipher the clear_text.
PIN encryption (one, optional)	
CLEARPIN	Do not encrypt the PIN block prior to encrypting the complete text message. This is the default.
SELFENC	Append the PIN-block self-encrypted to the clear PIN block within the unencrypted output message. See "PIN-block self-encryption" on page 439.

input_PIN_block
The *input_PIN_block* parameter is a pointer to an 8-byte string variable containing the input, encrypted PIN-block.

PIN_encrypting_key_identifier
The *PIN_encrypting_key_identifier* parameter is a pointer to a string variable containing an internal key token or the key label of an internal key token in key storage. The key is used to decipher the input PIN block and must be an IPINENC key-type.

input_PIN_profile
The *input_PIN_profile* parameter is a pointer to a string variable containing three 8-byte character strings with information defining the input PIN-block format. You can use PIN-block formats ISO-0, ISO-1, and ISO-2 with this verb.

input_PAN_data
The *input_PAN_data* parameter is a pointer to a string variable containing the PAN data. The verb uses the PAN data when it must output the PIN in a different PIN-block format and the input format is ISO-0. You supply the 12 rightmost PAN digits, excluding the check digit.

Note: When using the VISA-4 format, supply the 12 leftmost PAN digits, excluding the check digit.

secmsg_key_identifier
The *secmsg_key_identifier* parameter is a pointer to a string variable containing an internal key-token, or the key label of an internal key-token in key storage. The control vector must specify a SECMSG type key with the SMPIN control-vector bit (bit 19) on.

output_PIN_profile
The *output_PIN_profile* parameter is a pointer to a string variable containing three 8-byte character strings with information defining the output PIN-block format. You can use PIN-block formats ISO-0, ISO-1, and ISO-2 with this verb.

output_PAN_data
The *output_PAN_data* parameter is a pointer to a string variable containing the PAN data. The verb uses the PAN data when it must output the PIN in a different PIN-block format and the output format is ISO-0. You supply the 12 rightmost PAN digits, excluding the check digit.

Note: When using the VISA-4 format, supply the 12 leftmost PAN digits, excluding the check digit.

clear_text_length
The *clear_text_length* parameter is a pointer to an integer containing the length of text to be encrypted. This must be a multiple of 8, and less than or equal to 4096.

clear_text
The *clear_text* parameter is a pointer to the text string to be updated with a PIN block and encrypted.

initialization_vector
The *initialization_vector* parameter is a pointer to an 8-byte string containing the CBC-encryption initialization vector. The data to be exclusive-ORed with the first eight bytes of the text. This can be a null pointer when ECB mode is specified.

PIN_offset
The *PIN_offset* parameter is a pointer to an integer containing the offset to the location for the PIN block. Specify the start of the text as offset 0. The offset plus PIN_offset_field_length must be less than or equal to the clear_text_length.

PIN_offset_field_length
The *PIN_offset_field_length* parameter is a pointer to an integer value of 8.

enciphered_text
The *enciphered_text* parameter is a pointer to a string variable to receive the enciphered text message.

output_chaining_vector
The *output_chaining_vector* parameter is a pointer to an 8-byte string to receive the CBC chaining value. This is the same as the last eight bytes of returned enciphered text and can be used as an initialization_vector when encrypting subsequent data for a message. This can be a null pointer when ECB mode is specified.

Required commands

The Secure_Messaging_for_PINs verb requires the Secure Messaging for PINs command (offset X'0274') to be enabled in the active role.

SET_Block_Compose (CSNDSBC)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

The SET_Block_Compose verb creates a SET-protocol RSA-OAEP block and DES encrypts the data block using the SET protocols. Optionally, the verb computes the SHA-1 hash of the supplied data block and includes this in the OAEP block.

The DES_encrypted_block can be as many as eight bytes longer than the data_to_encrypt due to padding. Ensure the DES_encrypted_block buffer is large enough.

Restrictions

The data-block length variable is restricted to 32 megabytes.

The *DES_key_block_length* parameter must point to an integer value of zero. The *DES_key_block* parameter should be a null address pointer, or point to an unused 64-byte application variable.

The *chaining_vector* parameter must be a null address pointer, or point to an unused 18-byte application variable.

Format

CSNDSBC

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1
rule_array	Input	String array	rule_array_count * 8 bytes
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	updated data_to_encrypt_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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Block type (required)	
SET1.00	Specifies that the 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 string variable containing a binary value that is 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 in the *XData_string* parameter.

XData_string_length
The *XData_string_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *XData_string* variable. The maximum length is 94 bytes.

XData_string
The *XData_string* parameter is a pointer to a string containing extra-encrypted data within the OAEP-processed and RSA-encrypted block. If the *Xdata_string_length* variable is 0, this parameter is ignored but must still be declared.

data_to_encrypt_length
The *data_to_encrypt_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *data_to_encrypt* parameter. 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 is first padded by this service.

data_to_hash_length
The *data_to_hash_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *data_to_hash* variable.

The hash value is an optional part of the OAEP block. No hash value is computed or inserted into the OAEP block if the *data_to_hash_length* variable is 0.

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.

If the *data_to_hash_length* variable is not zero, a SHA-1 hash value of the *data_to_hash* variable is included in the OAEP block.

initialization_vector
The *initialization_vector* parameter is a pointer to a 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 number of bytes of data in the *RSA_public_key_identifier* variable. The maximum size allowed 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 or a key label of 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 number of bytes of data in the *DES_key_block* variable. The value must be 0 for this verb.

DES_key_block
The *DES_key_block* parameter is a pointer to a string variable containing DES key block. This parameter must be a null pointer, or a pointer to 64 bytes of unused application storage.

RSA-OAEP_block_length
The *RSA-OAEP_block_length* parameter is a pointer to an integer variable containing the number of bytes of data in the RSA-OAEP_block variable. The value 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* variable.

RSA-OAEP_block
The *RSA-OAEP_block* parameter is a pointer to a string variable containing the RSA-OAEP block.

chaining_vector
The *chaining_vector* parameter is a pointer to a string variable containing a work area that the security server uses to carry segmented data between calls. The parameter must contain a null pointer or a pointer to 18 bytes of unused application storage.

DES_encrypted_block
The *DES_encrypted_block* parameter is a pointer to a string variable containing the DES-encrypted data block returned by the verb (cleartext 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 Compose SET Block command (offset X'010B') to be enabled in the active role.

SET_Block_Decompose (CSNDSBD)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

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 as expressed in the Decipher service (see “Decipher (CSNBDEC)” on page 229).

The DES_key_block_length parameter must point to an integer which has a value of 0, 64, or 128. The DES_key_block parameter must point to a buffer of the size designated by DES_key_block_length. When the length is 0, the DES key block pointer can be set to NULL.

The *chaining_vector* parameter must be a null address pointer, or point to an unused 18-byte application variable.

Format

CSNDSBD

return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	exit_data_length bytes
rule_array_count	Input	Integer	1 or 2
rule_array	Input	String array	rule_array_count * 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
hash_block_length	In/Output	Integer	
hash_block	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 9.

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

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length and must be left-aligned and padded on the right with space characters. The rule_array keywords are shown in the following table:

Keyword	Meaning
Block type (required)	
SET1.00	Specifies that the structure of the RSA-OAEP encrypted block is defined by the SET 1.00 protocol.
PIN-block encryption (optional)	
PINBLOCK	Specifies that the OAEP block contains PIN information in the XDATA field, including an ISO-0 format PIN-block. The PIN block is encrypted, using an IPINENC or OPINENC key that is contained in the DES_key_block variable. The PIN information and the encrypted PIN-block are returned in the XData_string variable.

RSA-OAEP_block_length
The *RSA-OAEP_block_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *RSA-OAEP_block* variable. This value must be 128 bytes.

RSA-OAEP_block
The *RSA-OAEP_block* parameter is a pointer to a string variable containing the RSA-OAEP block.

DES_encrypted_data_block_length
The *DES_encrypted_data_block_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *DES_encrypted_data_block* variable. 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 a 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 number of bytes of data in the RSA_private_key_identifier variable. The maximum size allowed 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 or the key label of 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 number of bytes of data in the DES_key_block variable. The value can be 0, 64, or 128. These three values are used in the following way:

0	This is the normal value when the PINBLOCK keyword is not present. In this case, no DES key data is passed as input or output.
64	This value is permitted when the PINBLOCK keyword is not present, in order to improve compatibility with the SET_Block_Decompose verb defined for the S/390 ICSF. The coprocessor treats this in the same way as a value of zero.
128	This is the length when the PINBLOCK keyword is present.

DES_key_block
The *DES_key_block* parameter is a pointer to a string variable containing the PIN encrypting key used when the PINBLOCK keyword is present. For compatibility with the S/390 ICSF implementation of this verb, the parameter is also allowed to point to an unused 64-byte variable in application storage when the PINBLOCK keyword is not present.

The PIN encrypting-key token must be an internal token, and must be of type IPINENC or OPINENC. The key token must be in the last (rightmost) 64 bytes of a 128-byte buffer. The first 64 bytes of the buffer are reserved for future use, and should be set to X'00'.

The PIN encrypting-key token is returned to the caller in the same buffer on completion of the verb.

block_contents_identifier
The *block_contents_identifier* parameter is a pointer to a string variable containing 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, an XData string.

XData_string_length
The *XData_string_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *XData_string* variable. The minimum value 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* variable returned by the verb.

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.

When the XData field contains PIN information, 8 bytes of that information are an ISO-0 format PIN-block in clear text. This PIN block is enciphered using the PIN encryption-key received in the *DES_key_block* variable. The enciphered PIN-block replaces the clear text PIN-block in the *XData_string* variable returned by the verb.

chaining_vector
The *chaining_vector* parameter is a pointer to a string variable containing a

work area the security server uses to carry segmented data between calls. The parameter must contain a null pointer or a pointer to a 18 bytes of unused application storage.

data_block
The *data_block* parameter is a pointer to a string variable containing the decrypted DES-encrypted data block. The starting address must not fall inside the DES-encrypted data block area. Padding characters are removed.

hash_block_length
The *hash_block_length* parameter is a pointer to an integer variable containing the number of bytes of data in the *hash_block* variable. An error is returned if the *hash_block* variable is not large enough to hold the 20-byte SHA-1 hash.

On output, this field is updated to reflect the number of bytes of hash data returned in the *hash_block* variable, either 0 or 20 bytes.

hash_block
The *hash_block* parameter is a pointer to a string variable containing the SHA-1 hash extracted from the OAEP block returned by the verb.

Required commands

The SET_Block_Decompose verb requires the Decompose SET Block command (offset X'010C') to be enabled in the active role.

Two additional commands must be enabled in the active role when encrypting PIN data with this verb:

- When using an IPINENC type key, SET PIN Encrypt with IPINENC (offset X’0121’).
- When using an OPINENC type key, SET PIN Encrypt with OPINENC (offset X’0122’).

Transaction_Validation (CSNBTRV)

Coprocessor	AIX	i5/OS	Windows 2000
IBM 4758	2.53	2.54	2.53
IBM 4764		3.20	
		3.23	

With the Transaction_Validation verb you can generate and validate American Express card security codes (CSCs).

The Transaction_Validation verb generates and verifies transaction values based on information from the transaction and a cryptographic key. You select the validation method, and a mode, either the generate or verify mode, through rule-array keywords.

For the American Express process, the control vector supplied with the cryptographic key must indicate a MAC or MACVER class. The control vector bits zero to three can be B'0000'. Alternatively, you can ensure that a key is used only for the American Express CSC process by specifying a MAC or a MACVER-class key with control vector bits 0 - 3 valued to B'0100'. The control vector generate bit must be on (bit 20) if you request CSC generation and the verify bit (bit 21) must be on if you request verification.

The verb returns the validation within the return code. A return code of 0 indicates the transaction has been validated, and return code 4 indicates the transaction has not been validated.

Restrictions

The transaction_info and validation_values variables cannot exceed 256 bytes in length. CSC codes must be 19 bytes in length.

Format

CSNBTRV			
return_code	Output	Integer	
reason_code	Output	Integer	
exit_data_length	In/Output	Integer	
exit_data	In/Output	String	
rule_array_count	Input	Integer	1 or 2
rule_array	Input	String	rule_array_count * 8 bytes
transaction_key_identifier_length	Input	Integer	64
transaction_key_identifier	Input	String	64 bytes
transaction_info_length	Input	Integer	
transaction_info	Input	String	transaction_info_length bytes
validation_values_length	In/Output	Integer	
validation_values	In/Output	String	validation_values_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 9.

rule_array_count
The *rule_array_count* parameter points to an integer containing the number of the rule-array elements in the *rule_array* variable. The value must be 1 or 2 for this verb.

rule_array
The *rule_array* parameter is a pointer to a string variable containing an array of keywords. The keywords are 8 bytes in length, left-aligned, and padded on the right with space characters, as shown in the following table.

Keyword	**Meaning**
Operation (one, optional)	
VERIFY	Specifies verification of the value presented in the validation values variable. (This is the default when the **CSC-3**, **CSC-4**, or **CSC-5** keywords are used.)
GENERATE	Specifies generation of transaction validation values. (This is the default when the **CSC-345** keyword is used.)
American Express card security codes (one required with **VERIFY**)	
CSC-3	3-digit card security code (CSC) located on the signature panel (the default), **VERIFY** implied.
CSC-4	4-digit CSC located on the front of the card, **VERIFY** implied.
CSC-5	5-digit CSC located on the magnetic stripe, **VERIFY** implied.
American Express card security codes (required with **GENERATE**)	
CSC-345	Generates 3-byte, 4-byte, 5-byte values when given an account number and an expiration date, **GENERATE** implied.

transaction_key_identifier_length
The *transaction_key_identifier_length* parameter is a pointer to an integer variable containing 64, the length of the key token or key label variable.

transaction_key_identifier
The *transaction_key_identifier* parameter is a pointer to a string variable containing a key token or a key label of a key token in key storage.

transaction_info_length
The *transaction_info_length* parameter is a pointer to an integer variable containing the length of the *transaction_info* variable. For American Express CSC codes, this length must be 19.

transaction_info
The *transaction_info* parameter is a pointer to a string variable containing the concatenation of the 4-byte expiration date (in the format of YYMM) and the 15-byte American Express account number. Provide the information in character format.

validation_values_length
The *validation_values_length* parameter is a pointer to an integer variable containing the length of the *validation_values* variable.

validation_values

The *validation_values* parameter is a pointer to a string variable containing American Express CSC values. The data is output for **GENERATE** and input for **VERIFY**.

Operation	Element description	Validation-values length
GENERATE and **CSC-345**	555554444333, where: • 55555 = CSC 5 value • 4444 = CSC 4 value • 333 = CSC 3 value	12
VERIFY and **CSC-3**	333 = CSC 3 value	3
VERIFY and **CSC-4**	4444 = CSC 4 value	4
VERIFY and **CSC-5**	55555 = CSC 5 value	5

Required commands

| The Transaction_Validation verb requires the listed commands to be enabled in the
| active role, depending on the operation and card security code specified:

Operation	Security code	Command offset	Command
GENERATE	**CSC-345**	X'0291'	Generate CSC 3, 4, and 5 values
VERIFY	**CSC-3**	X'0292'	Verify CSC 3 values
VERIFY	**CSC-4**	X'0293'	Verify CSC 4 values
VERIFY	**CSC-5**	X'0294'	Verify CSC 5 values

Appendix A. Return codes and reason codes

This section describes the return codes and reason codes reported at the conclusion of verb processing.

Reason code numbers narrow down the meaning of a return code. All return code numbers are unique and associated with a single return code. Generally, you can base your application program design on the return codes.

Each verb supplies a return code and a reason code in the variables identified by the *return_code* and *reason_code* parameters.

Return codes

A return code provides a summary of the results of verb processing. A return code can have the values shown in Table 33.

Table 33. Return code values

Hex value	Decimal value	Meaning
00	00	This return code indicates a normal completion of verb processing. To provide additional information, there are also nonzero reason codes associated with this return code.
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 prematurely stopped processing. Generally, the application programmer needs to investigate the significance of the associated reason code to determine the origin of the problem. In some cases, due to transient conditions, retrying the verb might produce different results.
0C	12	This return code indicates that the verb prematurely stopped processing. Either a coprocessor is not available or a processing error occurred. The reason is most likely related to a problem in the set up of the hardware or in the configuration of the software.
10	16	This return code indicates that the verb prematurely stopped processing. A processing error occurred. If these errors persist, a repair of the coprocessor hardware or a correction to the coprocessor software might be required.

Reason codes

A reason code details the results of verb processing. Every reason code is associated with a single return code. A nonzero reason code can be associated with a zero return code.

User Defined Extensions (UDX) return reason codes in the range of 20480 (X'5000') through 24575 (X'5FFF').

The remainder of this appendix lists the reason codes that accompany each of the return codes. The return codes are shown in decimal form, and the reason codes are shown in decimal and in hexadecimal (hex) form.

Reason codes that accompany return code 0

Table 34. 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	701 (2BD)	A new master-key value has duplicate thirds.
0	702 (2BE)	A provided master-key part does not have odd parity.
0	10001 (2711)	A key encrypted under the old master key was used.

Reason codes that accompany return code 4

Table 35. 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	019 (013)	The financial PIN in a PIN block is not verified.
4	158 (09E)	The Key_Token_Change, Key_Record_Delete, or Key_Record_Write 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	283 (11B)	The coprocessor battery is low.
4	287 (11F)	The PIN-block format is not consistent.
4	429 (1AD)	The digital signature is not verified. The verb completed its processing normally.

Table 35. Reason codes for return code 4 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
4	1024 (400)	Sufficient shares have been processed to create a new master key.
4	2039 (7F7)	At least one control vector bit cannot be parsed.
4	2042 (7FA)	The supplied passphrase is not valid.

Reason codes that accompany return code 8

Table 36. Reason codes for return code 8

Return code Dec	Reason code Dec (Hex)	Meaning
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 is 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	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 does 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	039 (027)	A control vector violation occurred. Check all control vectors employed with the verb. For security reasons, no detail is provided.
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.

Table 36. Reason codes for return code 8 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
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 is 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 8 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.
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 is not present.
8	063 (03F)	A key token has an invalid token header (for example, not an internal token).
8	064 (040)	The RSA key is not permitted to perform the requested operation. Likely cause is key distribution usage is not enabled for the key.
8	065 (041)	The key token failed consistency checking.
8	066 (042)	The recovered encryption block failed validation checking.
8	067 (043)	RSA encryption failed.
8	068 (044)	RSA decryption failed.
8	072 (048)	The value that the size parameter specifies is not valid (too small, too large, negative, or zero).
8	081 (051)	The modulus length (key size) exceeds the allowable maximum.
8	085 (055)	The date or the time value is not valid.
8	090 (05A)	Access control checking failed. See the Required Commands descriptions for the failing verb.

Table 36. Reason codes for return code 8 *(continued)*

Return code Dec	Reason code Dec (Hex)	Meaning
8	091 (05B)	The time that was sent in your logon request was more than five minutes different from the clock in the secure module.
8	092 (05C)	The user profile is expired.
8	093 (05D)	The user profile has not yet reached its activation date.
8	094 (05E)	The authentication data (for example, passphrase) is expired.
8	095 (05F)	Access to the data is not authorized.
8	096 (060)	An error occurred reading or writing 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 value of the transaction_security_parameter is not valid.
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 or the placement of the padding 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. For example, digits other than 0 - 9 were found.
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	125 (07D)	A PKA92-encrypted key having the same EID as the local node cannot be imported.
8	153 (099)	The text length exceeds the system limits.
8	154 (09A)	The key token that the key_identifier parameter specifies is not an internal key-token or a key label.

Table 36. Reason codes for return code 8 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
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)	The MDC_Generate text length is in error.
8	170 (0AA)	Special authorization through the operating system is required to use this verb.
8	171 (0AB)	The control_array_count value is not valid.
8	175 (0AF)	The key token cannot be parsed because no control vector is present.
8	180 (0B4)	A key token presented for parsing is null.
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.
8	185 (0B9)	A disk I/O error occurred: perhaps the file is in-use, does not exist, and so forth.
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	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.

Table 36. Reason codes for return code 8 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
8	204 (0CC)	A memory allocation failed. This can occur in the host and in the coprocessor. Try closing other host tasks. If the problem persists, contact the IBM support center.
8	205 (0CD)	The X9.23 ciphering method is not consistent with the use of the CONTINUE keyword.
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 cannot 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, or a variable has caused an attempt to create an internal data structure that is too large.
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	382 (17E)	A time-limit violation occurred.
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)	The cryptographic hardware component reported that the user ID or role ID is not valid.
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	397 (18D)	The command was not processed because the active profile requires the user to be verified first.
8	398 (18E)	The command was not processed because the maximum PIN or password failure limit is exceeded.
8	407 (197)	There is a PIN-block consistency-check-error.
8	442 (1BA)	DES keys with replicated halves are not allowed.
8	605 (25D)	The number of output bytes is greater than the number that is permitted.
8	703 (2BF)	A new master-key value is one of the weak DES keys.
8	704 (2C0)	A new master key cannot have the same master key version number as the current master-key.
8	705 (2C1)	Both exporter keys specify the same key-encrypting key.
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 this hardware (DES on a CDMF-only system, or T-DES on DES-only or CDMF-only system)

Table 36. Reason codes for return code 8 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
8	714 (2CA)	A reserved parameter must be a null pointer or an expected value.
8	715 (2CB)	A parameter that must have a value of zero is not valid.
8	718 (2CE)	The hash value 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 not valid.
8	770 (302)	The PKA key token has a field that is not valid.
8	771 (303)	The user is not logged on.
8	772 (304)	The requested role does not exist.
8	773 (305)	The requested profile does not exist.
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 not valid.
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 not valid.
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 in use.

Table 36. Reason codes for return code 8 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
8	1025 (401)	The registered public key or retained private key name already exists.
8	1026 (402)	The key name (registered public key or retained private key) does not exist.
8	1027 (403)	Environment identifier data is already set.
8	1028 (404)	Master key share data is already set.
8	1029 (405)	There is an error in the EID 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 sections that are not valid.
8	1049 (419)	The master key share index is not valid.
8	1050 (41A)	The public-key encrypted-key is rejected because the EID with the key is the same as the EID for this node.
8	1051 (41B)	The private key is rejected because the key is not flagged for use in master-key cloning.
8	1100 (44C)	There is a general hardware device driver execution error.
8	1101 (44D)	There is a hardware device driver parameter that is not valid.
8	1102 (44E)	There is a hardware device driver non-valid buffer length.
8	1103 (44F)	The hardware device driver has too many opens. The device cannot open now.

Table 36. Reason codes for return code 8 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
8	1104 (450)	The hardware device driver is denied access.
8	1105 (451)	The hardware device driver device is busy and cannot perform the request now.
8	1106 (452)	The hardware device driver buffer is too small and the received data is truncated.
8	1107 (453)	The hardware device driver request is interrupted and the request is aborted.
8	1108 (454)	The hardware device driver detected a security tamper event.
8	2034 (7F2)	The environment variable that was used to set the default coprocessor is not valid, or does not exist for a coprocessor in the system.
8	2036 (7F4)	The contents of a chaining vector are not valid. Ensure that the chaining vector was not modified by your application program.
8	2038 (7F6)	No RSA private key information is provided.
8	2041 (7F9)	A default card environment variable is not valid.
8	2050 (802)	The current key serial number field in the PIN profile variable is not valid (not hexadecimal or too many one bits).
8	2051 (803)	There is an non-valid message length in the OAEP-decoded information.
8	2053 (805)	No message found in the OAEP-decoded data.
8	2054 (806)	There is a non-valid RSA Enciphered Key cryptogram: OAEP optional encoding parameters failed validation.
8	2055 (807)	The RSA public key is too small to encrypt the DES key.
8	2062 (80E)	The active role does not permit you to change the characteristic of a double-length key in the Key_Part_Import parameter.
8	2065 (811)	The specified key token is not null.
8	3001 (BB9)	The RSA-OAEP block contains a PIN block and the verb did not request PINBLOCK processing.
8	6000 (1770)	The specified device is already allocated.
8	6001 (1771)	No device is allocated.
8	6002 (1772)	The specified device does not exist.
8	6003 (1773)	The specified device is an improper type.
8	6004 (1774)	Use of the specified device is not authorized for this user.
8	6005 (1775)	The specified device is not varied online.
8	6006 (1776)	The specified device is in a damaged state.
8	6007 (1777)	The key-storage file is not designated.
8	6008 (1778)	The key-storage file is not found.
8	6009 (1779)	The specified key-storage file is either the wrong type or the wrong format.

Table 36. Reason codes for return code 8 (continued)

Return code Dec	Reason code Dec (Hex)	Meaning
8	6010 (177A)	The user is not authorized to use the key-storage file.
8	6011 (177B)	The specified CCA verb request is not permitted from a secondary thread.
8	6012 (177C)	A cryptographic resource is already allocated.
8	6013 (177D)	The length of the cryptographic resource name is not valid.
8	6014 (177E)	The cryptographic resource name is not valid, or does not refer to a coprocessor that is available in the system.

Reason codes that accompany return code 12

Table 37. Reason codes for return code 12

Return code Dec	Reason code Dec (Hex)	Meaning
12	097 (061)	File space in key storage is insufficient to complete the operation.
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)	There is a key-storage file I/O error, or the file is not found.
12	206 (0CE)	The key-storage file is not valid, or the master-key verification failed. There is an unlikely, but possible, synchronization problem with the Master_Key_Process verb.
12	207 (0CF)	The verification method flags in the profile are not valid.
12	324 (144)	There is insufficient memory available to process your request, either memory in the host computer, or memory inside the coprocessor including the flash EPROM used to store keys, profiles, and other application data.
12	338 (152)	This cryptographic hardware device driver is not installed or is not responding, or the CCA code is not loaded in the coprocessor.
12	339 (153)	A system error occurred in the interprocess communication routine.
12	764 (2FC)	The master keys are not loaded and, therefore, a key cannot be recovered or enciphered.
12	768 (300)	One or more paths for key-storage directory operations are improperly specified.
12	2045 (7FD)	The CCA software is unable to claim a semaphore. The system might be short of resources.
12	2046 (7FE)	The CCA software is unable to list all of the keys. The limit of 500 000 keys might have been reached.

Reason codes that accompany return code 16

Table 38. 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 the IBM support center.
16	336 (150)	An error occurred in a cryptographic hardware or software component.
16	337 (151)	A device software error occurred.
16	444 (1BC)	The verb-unique-data has an invalid length.
16	556 (22C)	The request parameter block failed consistency checking.
16	708 (2C4)	The cryptographic engine is returning inconsistent data.
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 occurred in the master-key manager.
16	769 (301)	The host system code or the CCA application in the coprocessor encountered an unexpected error and is unable to process the request.
16	2047 (7FF)	Unable to transfer Request Data from host to coprocessor.
16	2057 (809)	Internal error: memory allocation failure.
16	2058 (80A)	Internal error: unexpected return code from OAEP routines.
16	2059 (80B)	Internal error: OAEP SHA-1 request failure.
16	2061 (80D)	Internal error in CSNDSYI, OAEP-decode: enciphered message too long.

Appendix B. Data structures

This section 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, a key that is available to an application program or held in key storage is multiply-enciphered by some other key. When a key is enciphered by the CCA node's master key, the key is designated an internal key and is held in an internal key-token structure. Therefore, an *internal key-token* is used to hold a key and its related information for use at a specific CCA 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 IBM 4764 and IBM 4758:
- Master key verification pattern
- Token-validation value and 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) exists within an internal key token. An MKVP permits the cryptographic engine to detect whether the key within the token is enciphered by an available master key. Different internal key-verification-pattern approaches are employed depending on the version of the key token and, for DES key tokens, the value of the symmetric master key. See "Master key verification algorithms" on page 409.

An IBM 4758 or IBM 4764 does not permit the introduction of a new master key value that has the same verification value as either the current master key or as the old master key.

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 TVV is the sum (two's complement ADD), ignoring carries and overflow, of the key token by operating on 4 bytes at a time, starting with bytes 0 - 3 and ending with bytes 56 - 59. The 4-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 TVV.

When an application program supplies a key token, the CCA node checks the TVV. When a CCA verb builds a DES key-token, it generates a TVV in the key token.

The record-validation value (RVV) used in DES key-storage records uses the same algorithm as the TVV. The RVV is the sum of the bytes in positions 0 - 123, except for bytes 60 - 63.

Null key token

Table 39 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.

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.

The (DES) 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.

PKA key-storage uses an 8-byte structure, shown below, to represent a null key token. The DES_Key_Record_Read verb returns 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.

Table 39. PKA null key-token format

Offset	Length	Meaning
00	01	X'00' Indicates that this is a null key-token
01	01	X'00' Version zero
02	02	X'0008' Indicates a PKA null key-token
04	04	Reserved, binary zero

DES key tokens

DES key-token data structures are 64 bytes in length, contain an enciphered key, a control vector, various flag bits, version number, and token validation value.

An internal key token contains a key multiply-enciphered by a master key while an external key token contains a key multiply-enciphered by a key-encrypting key.

Internal DES key token

Starting with the IBM 4758 Version 2 CCA Support Program for IBM 4758 Models 002 and 023, the software accepts and outputs a version X'00' internal DES key token. This software also accepts the version X'03' internal DES key token.

Table 40. Internal DES key token, Version 0 format (Version 2 and later software)

Offset	Length	Meaning
00	1	X'01' (a flag that indicates an internal 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 Table 44 on page 351
07	1	Reserved, binary zero
08 - 15	8	Master key version number
16 - 23	8	The single-length operational (master-key encrypted) key or the left half of a double-length operational 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 right half of a double-length key
48 - 59	12	Reserved, binary zero
60 - 63	4	The token-validation value

Table 41. Internal DES key token, Version 3 format

Offset	Length	Meaning
Note: Created and processed by Version 1 Software. Version 2 and later software only accepts as input.		
00	1	X'01' (a flag that indicates an internal key-token)
01	1	Reserved, binary zero
02	2	Master key version number
04	1	The version number (X'03')
05	1	Reserved, binary zero
06	1	Flag byte 1
07	1	Reserved, binary zero
08 - 15	8	Reserved, binary zero
16 - 23	8	The single-length operational (master-key encrypted) key or the left half of a double-length operational 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 right half of a double-length key
48 - 59	12	Reserved, binary zero
60 - 63	4	The token-validation value

External DES key token

CCA generally uses a version X'00' external key-token as shown in Table 42 on page 350. The CCA Support Program also uses a version X'01' external key token to hold a double-length DATA key that is associated with a null control vector as shown in Table 43.

Table 42. External DES key-token format, Version X'00'

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
07	1	Flag byte 2 Reserved, generally X'00', except X'02' is tolerated.
08 - 15	8	Reserved, binary zero
16 - 23	8	The single-length key or the left half of a double-length key
24 - 31	8	Null, or the right half of a double-length key
32 - 39	8	The control-vector base
40 - 47	8	Null, or the control vector base for the right half of a double-length key
48 - 59	12	Reserved, binary zero
60 - 63	4	The token-validation value

Table 43. External DES key-token format, Version X'01'

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'01')
05	1	Reserved, binary zero
06	1	Flag byte 1
07	1	Reserved, binary zero
08 - 15	8	Reserved, binary zero
16 - 23	8	The left half of a double-length key
24 - 31	8	The right half of a double-length key
32 - 47	16	Null control vector, binary zero
48 - 58	11	Reserved, binary zero
59	1	Key length flag, double, X'10'
60 - 63	4	The token-validation value

DES key-token flag byte 1

Table 44. Key-token flag byte 1

Bits (MSB...LSB)[1]	Meaning
1*xxx xxxx*	The encrypted key value and the master key verification pattern are present.
0*xxx xxxx*	An encrypted key is not present.
x 0*xx xxxx*	The control-vector value is not present.
x 1*xx xxxx*	The control-vector value is present.
	All other bit combinations are reserved; undefined bits should be zero.

DES key-token flag byte 2

Table 45. Key-token flag byte 2

Bits (MSB...LSB)	Meaning
0000 0010	This key-encrypting key imports and exports external key-tokens using the transaction security system key-token format.

RSA PKA key tokens

PKA key tokens contain 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.

- A PKA internal key token contains a private key that is protected by encrypting the private key information using the CCA-node asymmetric-master-key, or by an object protection key (OPK) that is itself encrypted by the asymmetric master key. The internal key-token also contains the modulus and the public-key exponent. A master key version number is also included to enable determination that the proper master key is available to process the protected private key.

Note: 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 might be clear or might be protected by encryption using a double-length DES transport key. An external key token is an inter-product interchange data structure.

An RSA private key can be represented in one of several forms:
- By a modulus and the private-key exponent
- By a set of numbers used in the Chinese-remainder theorem (CRT). The coprocessor always generates a CRT key with p>q. If you import a CRT key from another RSA implementation with *q*>*p* the key is usable within the coprocessor, but your application encounters a performance penalty with each use of the key.

The private key can be protected by encrypting a confounder (a random number) and the private key information exclusive of the modulus. An encrypted private key in an external key token is protected by a double-length transport key and the

6. MSB is the most significant bit; LSB is the least significant bit.

EDE2 algorithm. See “CCA RSA private key encryption and decryption process” on page 403. The private key and the blinding values in an internal key-token are protected by the triple-length asymmetric master key and encryption algorithms as specified with each private key data structure.

RSA key-token sections

A PKA key token is the concatenation of an ordered set of sections. These key-token-section data structures are described in the sections referenced in the following table:

Section	Reference	Usage
Header	Table 46 on page 353	RSA Token Header
X'04'	Table 51 on page 359	RSA Public Key
X'02'	Table 47 on page 354	RSA Private Key, 1024-Bit Modulus-Exponent Format. Generated for external format for clear keys or for keys encrypted by a key-encrypting key.
X'05'	Table 48 on page 354	RSA Private Key, 2048-Bit Chinese-Remainder Format. Accepted only as an input and not generated in Version 2 and later.
X'06'	Table 49 on page 356	RSA Private Key, 1024-Bit Modulus-Exponent Format with OPK. Only used as a master-key encrypted, internal format.
X'08'	Table 50 on page 357	RSA Private Key, Chinese-Remainder Format with OPK. Internal and external format; replaces section type X'05'.
X'10'	Table 52 on page 360	RSA Private-Key Name
	Table 53 on page 360 through Table 59 on page 362	RSA Public-Key Certificates
X'FF'	Table 60 on page 363	RSA Private-Key Blinding Information

Note: You cannot use a modulus-exponent format for RSA keys with a modulus (key size) greater than 1024 bits.

You form a PKA key token by concatenating these sections:

- A token header (see Table 46 on page 353):
  – An external header (first-byte X'1E')
  – An internal header (first-byte X'1F')
- An optional private-key section in one of these formats:
  – Section identifier X'02' for a clear or key-encrypting key enciphered, modulus-exponent format key up to 1024 bits
  – Section identifier X'06' for a master-key enciphered, modulus-exponent format key up to 1024 bits
  – Section identifier X'08' for a CRT-format key up to 2048 bits
  – Section identifier X'05' for a CRT-format key up to 1024 bits is accepted as input
- A public-key section (RSA section identifier X'04', see Table 51 on page 359)
- An optional key-name section (section identifier X'10', see Table 52 on page 360)
- For internal key-tokens with private keys in X'02' or X'05' sections, a private-key blinding section (section identifier X'FF', see Table 60 on page 363)

- An optional certificate section (section identifier X'40' with subsidiary sections, see Table 53 on page 360)

The key tokens can be built with the PKA_Key_Token_Build verb.

PKA key-token integrity

If the token contains private key information, then the integrity of the information within the token can be verified by computing and comparing the SHA-1 hash values that are found in the private-key sections. The SHA-1 hash value at offset 4 within the private-key section requires access to the clear text values of the private-key components. The cryptographic engine verifies 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 optional, designated key-token information following the public-key section. The value of this SHA-1 hash is included in the computation of the hash value at offset 4. As with the offset-4 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 (for example, RACF) to ensure that the application program is entitled to employ the particular private key.

Number representation in PKA key tokens

- All length fields are in binary.
- All binary fields (exponents, lengths, and so forth) are stored with the high-order byte first (left, low-address, S/390 format); thus the least significant bits are to the right and preceded with zero-bits to the width of a field.
- In variable-length binary fields that have an associated field-length value, leading bytes that might contain X'00' can be dropped and the field shortened to contain only the significant bits.

Table 46. RSA key-token header

Offset (bytes)	Length (bytes)	Description
000	001	Token identifier (a flag that indicates token type) **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	The version number (X'00')
002	002	Length of the key-token structure
004	004	Reserved, binary zero
Note: See “Number representation in PKA key tokens.”		

Table 47. RSA private key, 1024-bit modulus-exponent format

Offset (bytes)	Length (bytes)	Description
000	001	X'02' Section identifier, RSA private key, modulus-exponent format (RSA-PRIV). This section type is created by selected IBM Cryptographic Adapters and the IBM CCA Support Program. Version 2 and later software uses this format for a clear or an encrypted RSA private key in an external key token.
001	001	The version number (X'00').
002	002	Length of the RSA private-key section X'016C' (364 decimal).
004	020	SHA-1 hash value of the private-key subsection clear text, offset 28 to and including the modulus that ends at offset 363.
024	002	Reserved, binary zero.
026	002	Master key version number 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 section; if there is no name section, then 20 bytes of X'00'.
050	001	Key-usage flag bits. The two high-order bits indicate permitted key usage in the decryption of symmetric keys and in the generation of digital signatures. Useful combinations: **X'00'** Only signature generation (SIG-ONLY) **X'C0'** Only key unwrapping (KM-ONLY) **X'80'** Both signature generation and key unwrapping (KEY-MGMT) All other bits, reserved, binary zero.
051	009	Reserved, binary zero.
060	024	Reserved, binary zero.
084		Start of the optionally encrypted subsection. Private key encryption: • External token: EDE2 process using the double-length transport key • Internal token: EDE3 process using the asymmetric master key See "Triple-DES ciphering algorithms" on page 416.
084	024	Random number (confounder).
108	128	Private-key exponent, *d*. $d=e^{-1}\text{mod}((p\text{-}1)(q\text{-}1))$, $1<d<n$, and 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 private-key exponent 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 $2^{512}<n<2^{1024}$

Note: See "Number representation in PKA key tokens" on page 353.

Table 48. Private key, 2048-bit Chinese-remainder format

Offset (bytes)	Length (bytes)	Description
000	001	X'05' Section identifier, RSA private key, CRT (RSA-OPT) format. This section type is only created by the IBM Version 1 CCA Support Program.
001	001	The version number (X'00')
002	002	Length of the RSA private-key section, 76 +*ppp* +*qqq* +*rrr* +*sss* +*ttt* +*uuu* +*xxx* +*nnn*.

Table 48. Private key, 2048-bit Chinese-remainder format (continued)

Offset (bytes)	Length (bytes)	Description
004	020	SHA-1 hash value of the private-key subsection clear text, 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 version number 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 section; if there is no name section, then 20 bytes of X'00'.
050	001	Key-usage flag bits The high-order bit indicates permitted key usage in the decryption of symmetric keys: **X'00'** Only signature generation (SIG-ONLY) **X'C0'** Only key unwrapping (KM-ONLY) **X'80'** Both signature generation and key unwrapping (KEY-MGMT) All other bits, reserved, binary zero.
051	001	Reserved, binary zero.
052		Start of the optionally encrypted subsection. Private key encryption: • External token: EDE2 process using the double-length transport key • Internal token: EDE3 process using the asymmetric master key. See "Triple-DES ciphering algorithms" on page 416.
052	008	Random number, confounder.
060	002	Length of prime number, *p*, in bytes: *ppp*.
062	002	Length of prime number, *q*, in bytes: *qqq*.
064	002	Length of d_p, in bytes: *rrr*.
066	002	Length of d_q, in bytes: *sss*.
068	002	Length of A_p, in bytes: *ttt*.
070	002	Length of A_q, in bytes: *uuu*.
072	002	Length of modulus, *n*., in bytes: *nnn*.
074	002	Length of padding field, in bytes: *xxx*.
076	*ppp*	Prime number, *p*.
076 +*ppp*	*qqq*	Prime number, *q*.
076 +*ppp* +*qqq*	*rrr*	$d_p = d \bmod(p\text{-}1)$.
076 +*ppp* +*qqq* +*rrr*	*sss*	$d_q = d \bmod(q\text{-}1)$.
076 +*ppp* +*qqq* +*rrr* +*sss*	*ttt*	$A_p = q^{p-1} \bmod(n)$.
076 +*ppp* +*qqq* +*rrr* +*sss* +*ttt*	*uuu*	$A_q = (n+1-A_p)$.

Table 48. Private key, 2048-bit Chinese-remainder format (continued)

Offset (bytes)	Length (bytes)	Description
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 8 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 $2^{512}<n<2^{2048}$.
Note: See "Number representation in PKA key tokens" on page 353.		

Table 49. RSA private key, 1024-bit modulus-exponent format with OPK

Offset (bytes)	Length (bytes)	Description
000	001	X'06' Section identifier, RSA private key, modulus-exponent format (RSA-PRIV). This section type is created by the IBM Version 2 and later CCA Support Program. This section type provides compatibility and interchangeability with the CCF hardware in S/390 processors.
001	001	The version number (X'00').
002	002	Length of the RSA private-key section X'0198' (408 decimal)
004	020	SHA-1 hash value of the private-key subsection clear text, offset 28 up to and including the modulus that ends at offset 363.
024	004	Reserved, binary zero.
028	001	Key format and security: **X'02'** Encrypted RSA private key with OPK
029	001	Private key source: **X'21'** Imported from clear text **X'22'** Imported from cipher text **X'23'** Generated using regeneration data **X'24'** Randomly generated
030	020	SHA-1 hash of all optional sections that follow the public key section, if any, else 20 bytes of X'00'.
050	001	Key-usage flag bits. The two high-order bits indicate permitted key usage in the decryption of symmetric keys and in the generation of digital signatures. Useful combinations: **X'00'** Only signature generation (SIG-ONLY) **X'C0'** Only key unwrapping (KM-ONLY) **X'80'** Both signature generation and key unwrapping (KEY-MGMT) All other bits, reserved, binary zero.
051	003	Reserved, binary zero.
054	006	Reserved, binary zero.
060	048	Object Protection Key (OPK); six 8-byte values: confounder, three key values, and two initialization vector values. The asymmetric master key encrypts the OPK using the EDE3 algorithm. See "Triple-DES ciphering algorithms" on page 416.

Table 49. RSA private key, 1024-bit modulus-exponent format with OPK (continued)

Offset (bytes)	Length (bytes)	Description
108	128	Private-key exponent, *d*. $d=e^{-1}\bmod((p-1)(q-1))$, $1<d<n$, and where *e* is the public exponent. The OPK encrypts the private key exponent using the EDE5 algorithm. See "Triple-DES ciphering algorithms" on page 416 and Figure 36 on page 419.
236	128	Modulus, *n*. $n=pq$, where *p* and *q* are prime and $2^{512}<n<2^{1024}$
364	016	Asymmetric-keys master key verification pattern.
380	020	SHA-1 hash value of the subsection clear text, offset 400 to the section end. This hash value is checked after an enciphered private key is deciphered for use.
400	002	Reserved, binary zero.
402	002	Reserved, binary zero.
404	002	Reserved, binary zero.
406	002	Reserved, binary zero.

Table 50. RSA private key, Chinese-remainder format with OPK

Offset (bytes)	Length (bytes)	Description
000	001	X'08' Section identifier, RSA private key, CRT format (RSA-CRT). This section type is created by the IBM Version 2 and later CCA Support Program.
001	001	The version number (X'00').
002	002	Length of the RSA private-key section, 132 *+ppp +qqq +rrr +sss +uuu +xxx +nnn*.
004	020	SHA-1 hash value of the private-key subsection clear text, offset 28 to the end of the modulus.
024	004	Reserved, binary zero.
028	001	Key format and security: External token: **X'40'** Unencrypted RSA private-key subsection identifier **X'42'** Encrypted RSA private-key subsection identifier Internal token: **X'08'** Encrypted RSA private-key subsection identifier
029	001	External tokens, reserved, binary zero. Internal tokens: **X'21'** Imported from clear text **X'22'** Imported from cipher text **X'23'** Generated using regeneration data **X'24'** Randomly generated
030	020	SHA-1 hash of all optional sections that follow the public key section, if any; else 20 bytes of X'00'.

Table 50. RSA private key, Chinese-remainder format with OPK *(continued)*

Offset (bytes)	Length (bytes)	Description
050	001	Key-usage flag bits. The two high-order bits indicate permitted key usage in the decryption of symmetric keys and in the generation of digital signatures. Useful combinations: **X'00'** Only signature generation (SIG-ONLY) **X'C0'** Only key unwrapping (KM-ONLY) **X'80'** Both signature generation and key unwrapping (KEY-MGMT) All other bits, reserved, binary zero.
051	003	Reserved, binary zero.
054	002	Length of the prime number, *p*, in bytes: *ppp*.
056	002	Length of the prime number, *q*, in bytes: *qqq*.
058	002	Length of d_p, in bytes: *rrr*.
060	002	Length of d_q, in bytes: *sss*.
062	002	Length of the *U* value, in bytes: *uuu*.
064	002	Length of the modulus, *n*, in bytes: *nnn*.
066	002	Reserved, binary zero.
068	002	Reserved, binary zero.
070	002	Length of the pad field, in bytes: *xxx*.
072	004	Reserved, binary zero.
076	016	External token, reserved, binary zero. Internal token, asymmetric master-key verification pattern.
092	032	External token: reserved, binary zero. Internal token: Object Protection Key (OPK), 8-byte confounder and three 8-byte keys used in the triple-DES CBC process to encrypt the private key and blinding information. These 32 bytes are triple-DES CBC encrypted by the asymmetric master key. See T-DES at “Triple-DES ciphering algorithms” on page 416.
124		Start of the (optionally) encrypted subsection. • External token: When offset 028 is X'40', the subsection is not encrypted. When offset 028 is X'42', the subsection is encrypted by the double-length transport key using the triple-DES CBC process. • Internal token: When offset 028 is X'08', the subsection is encrypted by the triple-length OPK using the triple-DES CBC process. See “Triple-DES ciphering algorithms” on page 416.
124	008	Random number, confounder.
132	*ppp*	Prime number, *p*.
132 +*ppp*	*qqq*	Prime number, *q*.
132 +*ppp* +*qqq*	*rrr*	$d_p = d \bmod (p\text{-}1)$.
132 +*ppp* +*qqq* +*rrr*	*sss*	$d_q = d \bmod (q\text{-}1)$.
132 +*ppp* +*qqq* +*rrr* +*sss*	*uuu*	$U = q^{-1} \bmod (p)$.
132 +*ppp* +*qqq* +*rrr* +*sss* +*uuu*	*xxx*	X'00' padding of length xxx bytes such that the length from the start of the confounder at offset 124 to the end of the padding field is a multiple of 8 bytes.

Table 50. RSA private key, Chinese-remainder format with OPK (continued)

Offset (bytes)	Length (bytes)	Description
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.		
132 +*ppp* +*qqq* +*rrr* +*sss* +*uuu* +*xxx*	*nnn*	Modulus, *n*. n=pq where p and q are prime and $2^{512}<n<2^{2048}$.
Note: See “Number representation in PKA key tokens” on page 353.		

Table 51. RSA public key

Offset (bytes)	Length (bytes)	Description
000	001	X'04', Section identifier, RSA public key.
001	001	The version number (X'00').
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 0. The RSA private-key section contains the modulus.
012	xxx	Public-key exponent, *e* (this field length is typically 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 does 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 is absent when the modulus is contained in the private-key-section. If present, the field length is 64 to 256 bytes.
Note: See “Number representation in PKA key tokens” on page 353.		

Table 52. RSA private-key name

Offset (bytes)	Length (bytes)	Description
000	001	X'10', Section identifier, private-key name.
001	001	The version number (X'00').
002	002	Section length, X'0044' (68 decimal).
004	064	Private-key name, left-aligned, 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. When generating a *retained private key*, the name supplied in this part of the skeleton key-token is subsequently used in the coprocessor to locate the retained key.
Note: See "Number representation in PKA key tokens" on page 353.		

RSA public-key certificate section: An optional *public key certificates* section can be included in an RSA key-token. The section consists of:
- The section header (identifier X'40')
- A public key subsection (identifier X'41')
- An optional certificate information subsection (identifier X'42') with any or all of these elements:
  – User data (identifier X'50')
  – EID (identifier X'51')
  – Serial number (identifier X'52')
- A signature subsection (identifier X'45').

The section 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.

Table 53. RSA public-key certificates section header

Offset (bytes)	Length (bytes)	Description
000	001	X'40', Section identifier, certificate.
001	001	The version number (X'00').
002	002	Section length; includes: • Section header • Public key subsection • Information subsection (optional) • Signature subsections
Note: See "Number representation in PKA key tokens" on page 353.		

Table 54. RSA public-key certificates public key subsection

Offset (bytes)	Length (bytes)	Description
000	001	X'41', Public Key Subsection identifier.
001	001	The version number (X'00').
002	002	Subsection 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*.
012	*xxx*	Public-key exponent, e (this field length is typically 1, 3, or 64 – 256 bytes). *e* must be odd, and $1<e<n$.
012 +*xxx*	*yyy*	Modulus, *n*. $n=pq$, where *p* and *q* are prime and $2^{512}<n<2^{2048}$. This field is absent when the modulus is contained in the private-key section. If present, the field length is 64 – 256 bytes.
Note: See "Number representation in PKA key tokens" on page 353.		

Table 55. RSA public-key certificates optional information subsection header

Offset (bytes)	Length (bytes)	Description
000	001	X'42', information subsection header.
001	001	The version number (X'00').
002	002	Subsection length, 4+*iii*.
004	*iii*	The information field that contains any of the includable TLV entities: • User data (ID = 50) • EID (ID = 51) • Serial number (ID = 52)
Note: See "Number representation in PKA key tokens" on page 353.		

Table 56. RSA public-key certificates user data TLV

Offset (bytes)	Length (bytes)	Description
000	001	X'50', user data TLV header.
001	001	The version number (X'00').
002	002	TLV length, 4+*uuu*.
004	uuu	User-provided data. $0 \le uuu \le 64$.
Note: See "Number representation in PKA key tokens" on page 353.		

Table 57. RSA public-key certificates environment identifier TLV

Offset (bytes)	Length (bytes)	Description
000	001	X'51', private key EID TLV header.
001	001	The version number (X'00').
002	002	X'0014', TLV length.
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 fills in the EID string prior to certifying the public key. The EID value is encoded using the ASCII character set.
Note: See "Number representation in PKA key tokens" on page 353.		

Table 58. RSA public-key certificates serial number TLV

Offset (bytes)	Length (bytes)	Description
000	001	X'52', serial number TLV header.
001	001	The version number (X'00').
002	002	X'000C', TLV length.
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 fills in the serial number prior to certifying the public key.
Note: See "Number representation in PKA key tokens" on page 353.		

Table 59. RSA public-key certificates signature subsection

Offset (bytes)	Length (bytes)	Description
000	001	X'45', signature subsection header.
001	001	The version number (X'00').
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 9796-1 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: 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.		
Note: See "Number representation in PKA key tokens" on page 353.		

RSA private-key blinding information:

Table 60. RSA private-key blinding information

Offset (bytes)	Length (bytes)	Description
000	001	X'FF', Section identifier, private-key blinding information. Used with internal key-tokens created by the CCA Support Program, Version 1 (having section identifiers X'02' or X'05').
001	001	The version number (X'00').
002	002	Section length, 34 + *rrr* + *iii*.
004	020	SHA-1 hash value of the internal information subsection clear text, 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. An internal token with section identifiers X'02' or X'05' uses the asymmetric master key and the EDE3 algorithm. See "Triple-DES ciphering algorithms" on page 416.	
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*	X'00' 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 8 bytes.
End of the encrypted subsection.		
Note: See "Number representation in PKA key tokens" on page 353.		

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. Ensure that the application program does not change the chaining-vector information. The verb uses the chaining vector to carry information between procedure calls.

Table 61. 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 - 7).
09-15	7	The bytes that are carried over and left-aligned.
16	2	The token master-key verification pattern.
Note: See "Number representation in PKA key tokens" on page 353.		

Key-storage records

Key storage exists as an online, resident data set on the hard disk drive. Key storage contains key records. Key records contain a key label, space for a key token, and control information. The stored key tokens are accessed using the key label. DES and PKA key tokens are held in independent key storage data sets.

For platforms other than i5/OS, 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.

- Table 62 on page 365 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.
- Table 63 on page 366 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.

For platforms other than i5/OS, Table 64 on page 367 shows the format of both the DES and PKA records that contain key tokens. For the i5/OS platform, the DES and PKA key tokens are held in distinct record formats.

- Table 65 on page 367 shows the format of the records in i5/OS DES key-storage that contain key tokens.
- Table 66 on page 368 shows the format of the records in i5/OS PKA key-storage that contain key tokens.

Table 62. Key-storage file header, record 1 (not i5/OS)

Offset	Length	Meaning
000	04	The total length of this key record.
004	04	The record validation value.
008	64	The key label without separators ($$FORTRESS$REL01$MASTER$KEY$VERIFY$PATTERN).
072	15	The date and time 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)
087	15	The date and time 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 (X'01') or PKA (X'1F') key token.
129	01	Version 1, X'00'; Version 2 and later, X'01'.
130	02	Token length which is a value of 64.
132	02	Reserved.
134	02	First two bytes of the SHA-1 MKVP. See "SHA-1 based master key verification method" on page 409.
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.

Table 63. Key-storage file header, record 2 (not i5/OS)

Offset	Length	Meaning
000	04	The total length of this key record.
004	04	The record validation value.
008	64	The key label without separators. For the DES key-storage file, the key label is $$FORTRESS$DES$REL01$KEY$STORAGE$FILE$HEADER . For the PKA key-storage file, the key label is $$FORTRESS$PKA$REL01$KEY$STORAGE$FILE$HEADER .
072	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)
087	15	The date and time when this record was last updated. This field has the same format as the created date in the previous row of this table.
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.

Table 64. Key-record format in key storage (not i5/OS)

Offset	Length	Meaning
000	04	The total length of this key record.
004	04	The record validation value.
008	64	The key label without separators.
072	15	The date and time 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)
087	15	The date and time when this record was last updated. This field has the same format as the created date in the previous row of this table.
102	26	Reserved.
128	??	A DES or PKA key token.

Table 65. DES key-record format, i5/OS key storage

Offset	Length	Meaning
000	056	The key label without separators.
056	002	Reserved.
058	064	The DES key token.
122	004	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)
126	004	The date and time of when this record was last updated. This field has the same format as the created date in the previous row of this table.
130	002	Reserved.
132	004	The record validation value.
136	120	Reserved.

Table 66. PKA key-record format, i5/OS key storage

Offset	Length	Meaning
000	64	The key label without separators.
064	24	Reserved.
088	04	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)
092	04	The date and time of when this record was last updated. This field has the same format as the created date in The previous row of this table.
096	04	The record validation value.
100	02	The key token length.
102	??	The 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. You locate the data set using the fully-qualified data-set name returned by the DES_Key_Record_List and PKA_Key_Record_List verbs.

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.

Table 67. Key-record-list data set format (other than i5/OS)

Offset	Length	Meaning
Header record (part 1)		
000	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).
024	02	This field contains spaces for separation.
026	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.

Table 67. Key-record-list data set format (other than i5/OS) (continued)

Offset	Length	Meaning
045	05	This field contains spaces for separation.
050	06	This field contains the number of detail records.
056	02	This field contains spaces for separation.
058	04	This field contains the length of each detail record, in character form, and left-aligned. The length is 154.
062	04	This field contains the offset to the first detail record, in character form, and left-aligned. The offset is 154.
066	09	This field is reserved filled with space characters.
075	02	This field contains carriage return/line feed (CR/LF).
Header record (part 2)		
077	64	This field contains the key-label pattern that you used to request the list.
141	11	This field is reserved filled with space characters.
152	02	This field contains a carriage return or line feeds (CR/LF).
Detail record (part 1)		
00	01	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.
01	02	This field contains spaces for separation.
03	64	This field contains the key label.
67	08	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	02	This field contains a carriage return or line feeds (CR/LF).
Detail record (part 2)		
77/0	04	For an internal token, this field contains (the first) two bytes of the Master key version number expressed in hexadecimal.
81/4	01	This field contains spaces for separation.
82/5	08	Reserved, filled with space characters.
90/13	02	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	02	This field contains spaces for separation.

Table 67. Key-record-list data set format (other than i5/OS) (continued)

Offset	Length	Meaning
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	01	This field contains a space character for separation.
133/56	08	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	02	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 2-byte and 4-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.

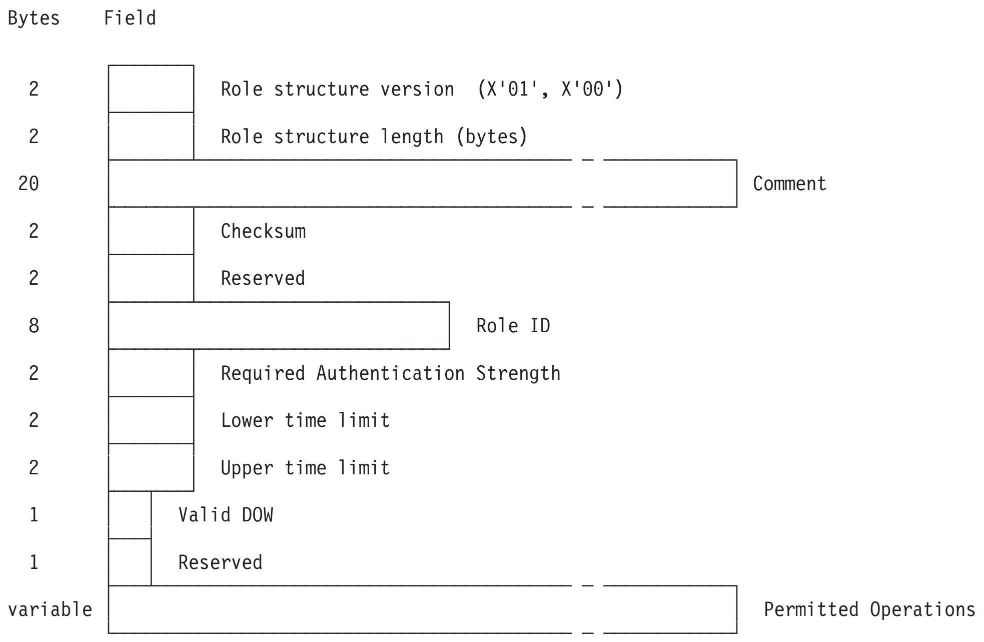


Figure 11. Role layout

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 can be verified by the IBM 4764 Cryptographic Coprocessor with a future version of the role structure.

The permitted operations are defined by the *Access-Control-Point list*, described in "Access-control-point list" on page 372.

The lower time limit and upper time limit fields are 2-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 zero, 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 370.

The header defines the number of roles that follow in the rest of the structure. Its layout is shown in Figure 12, with three concatenated role structures shown for illustration.

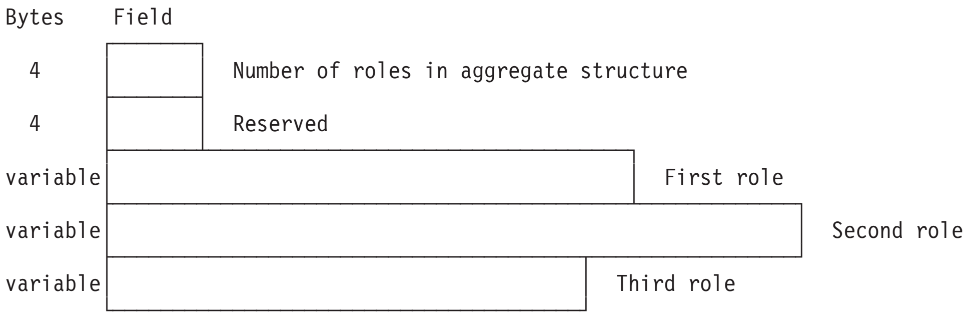


Figure 12. Aggregate role structure with header

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 2-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 waste 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 13 on page 373.

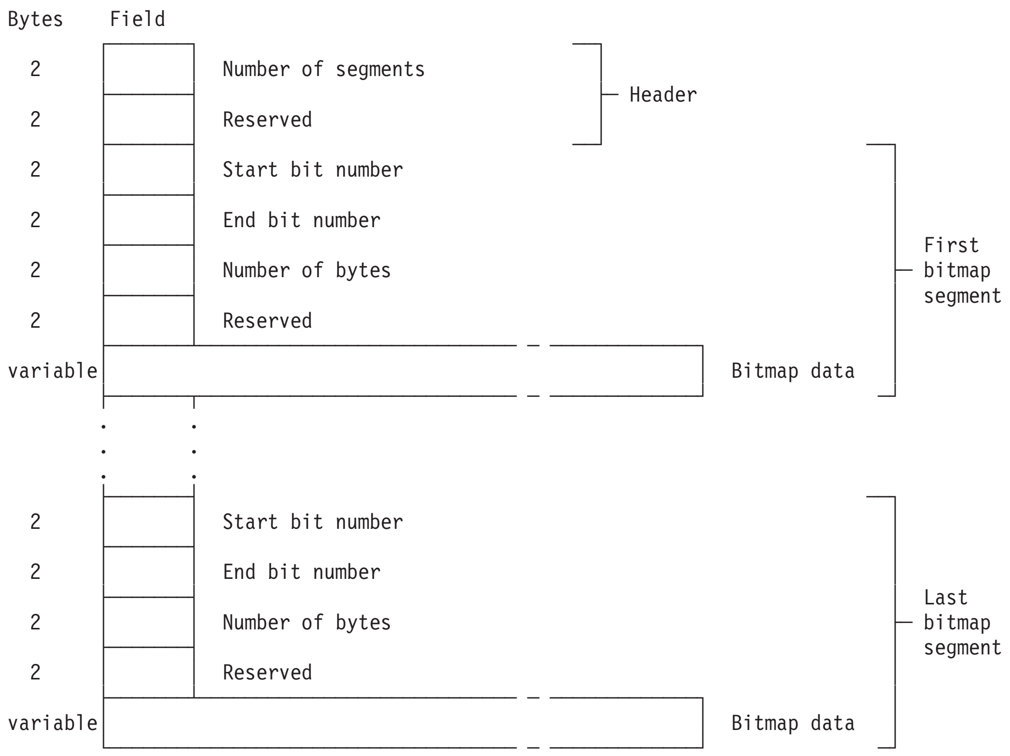


Figure 13. Access-control-point structure

Default role contents

The default role has have the following characteristics:

- The role ID is **DEFAULT**.
- The required authentication strength level is zero.
- The role is valid at all times and on all days of the week.
- The only functions that are permitted are those related to access-control initialization. This guarantees that the owner initializes the coprocessor before any 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 Table 68.

Table 68. Functions permitted in default role

Code	Function name
X'0107'	One-Way Hash, SHA-1
X'0110'	Set Clock
X'0111'	Reinitialize Device
X'0112'	Initialize ACS
X'0113'	Change User Profile Expiration Date
X'0114'	Change User Profile Authentication Data
X'0115'	Reset User Profile Login-Attempt-Failure Count
X'0116'	Read Public Access Control Information

Table 68. Functions permitted in default role (continued)

Code	Function name
X'0117'	Delete User Profile
X'0118'	Delete 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 the Access_Control_Initialization or Access_Control_Maintenance verbs.

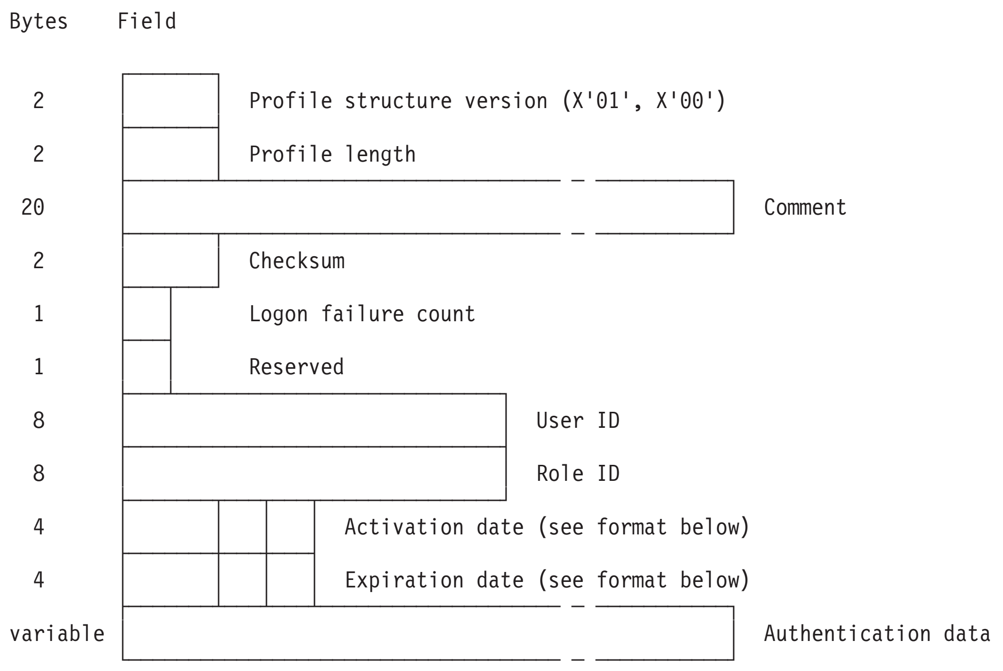


Figure 14. Profile layout

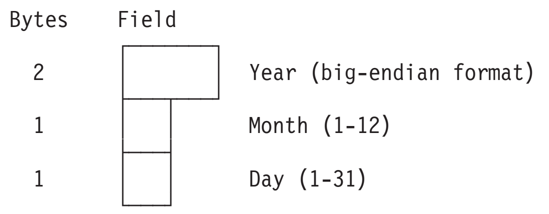


Figure 15. Layout of profile activation and expiration dates

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 must have a value of zero in the initialization data you send with Access_Control_Initialization.

The *checksum* is defined as the exclusive-OR 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.

Aggregate profile structure

For initialization, a set of zero or one 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 374.

The header defines the number of profiles that follow in the rest of the structure. Its layout is shown in Figure 16, with three concatenated profile structures shown for illustration.

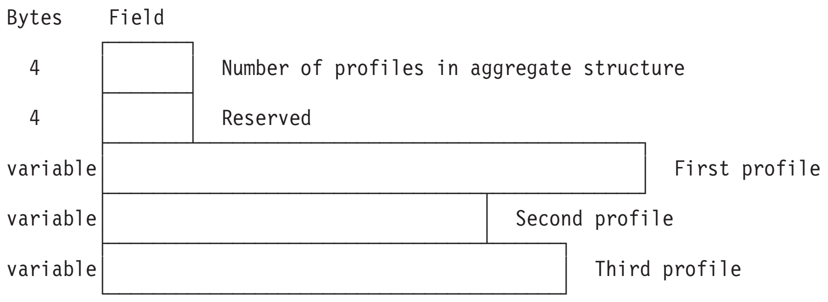


Figure 16. Aggregate profile structure with header

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 2-byte integer value defining how many bytes of authentication information are in the structure. 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 2-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 17 on page 376.

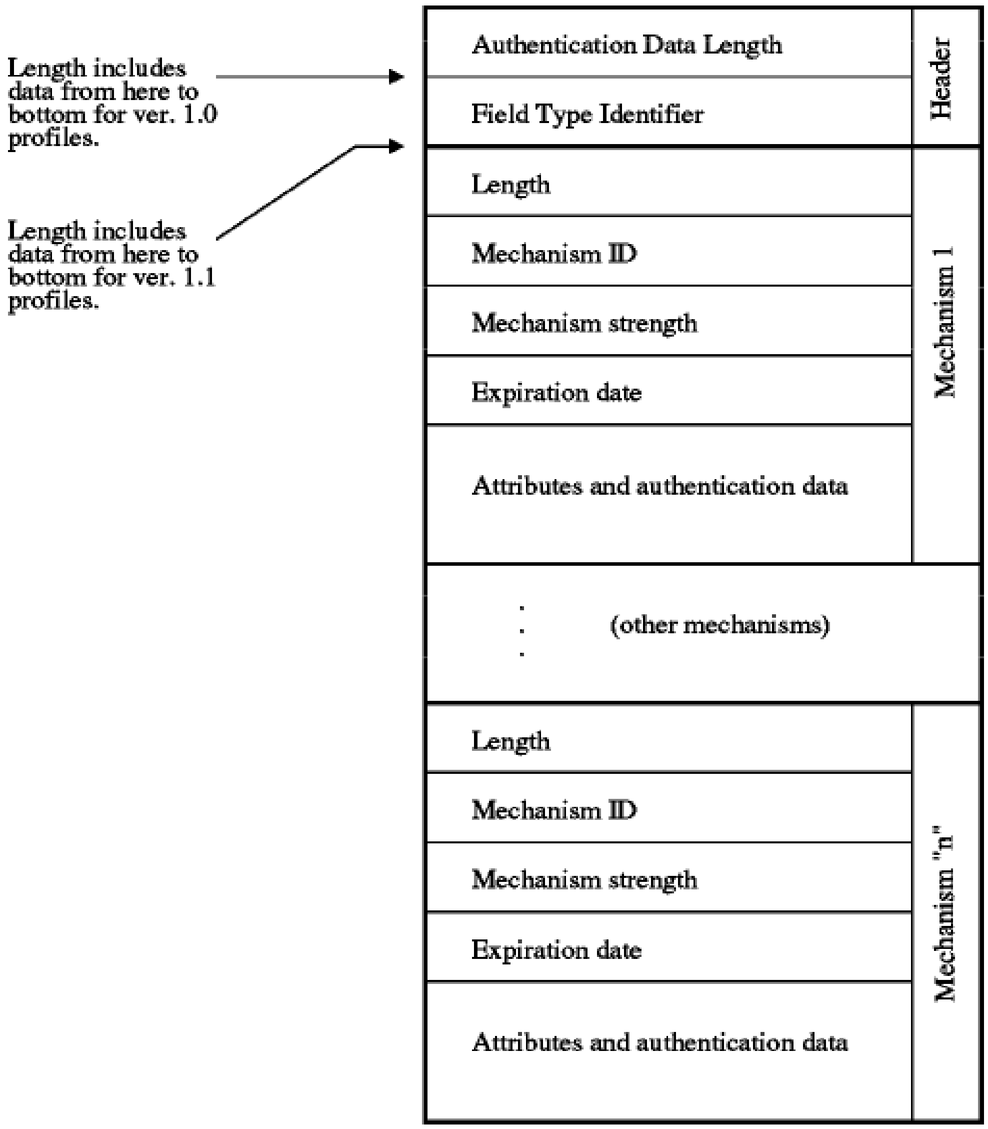

Length includes data from here to bottom for ver. 1.0 profiles.
Length includes data from here to bottom for ver. 1.1 profiles.
Authentication Data Length
Field Type Identifier
Header
Length
Mechanism ID
Mechanism strength
Expiration date
Attributes and authentication data
Mechanism 1
(other mechanisms)
Length
Mechanism ID
Mechanism strength
Expiration date
Attributes and authentication data
Mechanism "n"

Figure 17. Layout of the authentication data field

The content of the individual authentication data structures is shown in Table 69.

Table 69. Authentication data for each authentication mechanism

Field name	Length (bytes)	Description
Length	2	The size of this set of authentication mechanism data, in bytes. The length field includes all bytes of mechanism data following the length field itself.
Mechanism ID	2	An identifier that 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'.

Table 69. Authentication data for each authentication mechanism (continued)

Field name	Length (bytes)	Description
Mechanism Strength	2	An integer value which defines the strength of this identification mechanism, relative to all others. Higher values reflect greater strength. A value of 0 is reserved for users who have not been authenticated in any way.

Expiration Date	4	The last date on which this authentication data can 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 4-byte structure, as shown in the following C type definition.
		`typedef struct {`
		`   unsigned char exp_year[2];`
		`   unsigned char exp_month;`
		`   unsigned char exp_day;`
		`} expiration_date_t;`
		
		The 2-byte `exp_year` is in big-endian format. The high-order byte is at the lower numbered address.

Mechanism Attributes	4	This field contains flags and attributes needed to fully describe the operation and use of the authentication mechanism. One flag is defined for all methods: **Renewable** A Boolean value that 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 first logs 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 4-byte attributes field. The other 31 bits are unused, and must be set to 0.
Mechanism data	variable	This field contains the data needed to perform the authentication. The size, content, and complexity of this data varies according to the authentication mechanism. For example, the content might be as simple as a password that is compared to one entered by the user, or it might 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 value of the user's passphrase. The hash value is computed in the host, where it is used to construct the profile that is downloaded to the coprocessor.

Examples of the data structures

passphrase authentication data

Figure 18 shows the contents of a sample authentication mechanism data structure for a passphrase.

```
00 20 00 01 01 80 07 d8 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 18. 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).

00 01 The mechanism identifier, for passphrase authentication data.

01 80 The mechanism strength, Hex 0180, or decimal 384.

07 D8 The year of the passphrase expiration date, Hex 07D8, or decimal 2008.

06 01 The month and day 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 value of the user's passphrase. In this case, the passphrase is:

```
This is my passphrase.
```

User profile

Figure 19 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 d5 06 01     ith ADMIN1  ....
07 d8 0c 1f 00 24 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 19. 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. The data is a placeholder.

00 The logon failure count.

00 Reserved field, which must be 0.

J_Smith
The user ID for this profile.

ADMIN1
The role that defines the authority associated with this profile.

07 D5 The year of the profile's activation date. Hex 07D5 is equal to decimal 2005.

06 01 The month and day of the profile's activation date. This represents June 1.

07 D8 The year of the profile's expiration date. Hex 07D8 is equal to decimal 2008.

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 20 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 d5 06 01 07 d8 0c 1f 00 24 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 20. 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 zeros.

User profile
The remainder of this structure contains the single user profile that was described earlier in this section.

Access-control-point list

Figure 21 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 21. 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 zeros.

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 zeros.

F0 FF FF FF ... FF FF (35 bytes)
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 zeros.

8F 99 FE
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 22 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 22. 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.

00 00 A reserved field, which must be filled with zeros.

DEFAULT
The Role ID for this role. The role in this example replaces 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 23 shows the an aggregate role data structure, as you might 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 23. 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 zeros.

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 Table 70.

Table 70. 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 EID.
028	008	Origin-coprocessor serial number.
036	xxx	Cloning information TLV's: • Master key share • Signature And 1 – 7 bytes of padding to ensure that length '*xxx*' is a multiple of 8 bytes.
Note: The information from offset 036 through 036+xxx is triple encrypted with a triple-length DES key using the EDE3 encryption process, see "Triple-DES ciphering algorithms" on page 416.		

Table 71. 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.

Table 72. 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 or IBM 4764 becomes a practical cryptographic engine when it accepts and validates digitally signed software. IBM exports the IBM 4758 and IBM 4764 as non-cryptographic products, and controls and reports 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). Two capabilities are controlled:

- The length of keys used 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. Table 73 shows the format of the data structure that contains the function control vector as distributed by IBM.

Table 73. 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.

Table 73. FCV distribution structure (continued)

Offset decimal (hex)	Length decimal	Meaning
470 (1D6)	204	Function control vector (FCV). This is the information that you supply to the coprocessor using the Cryptographic_Facility_Control verb. It consists of the FCV information and a signature that is validated by the CCA code within the coprocessor.
674 (2A2)	128	Digital signature on the complete structure (excepting this signature itself).
FCV Supplied to coprocessor (offset 470 above)		
470 (1D6)	001	Record ID, X'06'.
471 (1D7)	001	Version, X'00'.
472 (1D8)	002	Padding, X'0000'.
474 (1DA)	004	FCV record structure length, X'CC', little endian.
478 (1DE)	004	Signature rules, X'FF000000'.
482 (1E2)	001	FCV format version, X'00'.
483 (1E3)	001	CCA services class, X'01', Basic.
484 (1E4)	001	X'01', CDMF only X'03', CDMF and 56-bit DES X'07', CDMF, 56-bit DES, Triple-DES.
485 (1E5)	001	SET services, X'01', CSNDSBD and CSNDSBC with 56-bit DES and 1024-bit RSA key length permitted.
486 (1E6)	004	Reserved, X'00000000'.
490 (1EA)	002	Maximum modulus length for symmetric encryption, little endian X'0008' is 2048 bits, X'0004' is 1024 bits, X'0002' is 512 bits.
492 (1EC)	054	Reserved, X'00...00'.
546 (222)	128	Signature validated by the coprocessor (using key FcvPuK).
Components of the FCV (offset 470 above)		

Appendix C. CCA control-vector definitions and key encryption

This section describes the following concepts and tasks:

- Understanding DES control-vector values
- Specifying a control-vector-base value
- Changing control vectors
- Understanding CCA key encryption and decryption processes

Note: Unless noted otherwise, a *control vector* is a DES control vector base in this appendix.

In 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.

Understanding DES control-vector values

The CCA key token includes the control vector and the encrypted 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 processes” on page 402.

Although the CCA architecture permits several advanced techniques, the techniques 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. This topic about 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.

Bits 8 to 14, and sometimes bits 18 to 22, of a control vector define the key as belonging to one of several general classes of keys as shown in Table 74.

Table 74. Key classes

Key type	Key usage
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.

Table 74. Key classes (continued)

Key type	Key usage
Data operation keys	
CIPHER, DECIPHER, ENCIPHER	Used only to encrypt or decrypt data.
DATA	Used to encrypt or decrypt data, or to generate or verify a MAC.
DATAC	Used to specify a DATA-class key that performs in the Encipher and Decipher verbs, but not in the MAC_Generate and MAC_Verify verbs.
DATAM	Used to specify a DATA-class key that performs in the MAC_Generate and MAC_Verify verbs, but not in the Encipher and Decipher verbs.
DATAMV	Used to specify a DATA-class key that performs in the MAC_Verify verb, but not in the MAC_Generate, Encipher, or Decipher verbs.
MAC	Used to generate or verify a MAC.
MACVER	Used to verify a MAC code; cannot be used in MAC-generation.
SECMSG	Used to encrypt keys or PINs in a secure message.
PIN-processing keys	
IPINENC	Used to decrypt a PIN block.
OPINENC	Used to encrypt a PIN block.
PINGEN	Used to generate and verify PIN values.
PINVER	Used to verify, but not generate, PIN values.
Special cryptographic-variable encrypting keys	
CVARENC	Used to encrypt the mask arrays in the Cryptographic_Variable_Encipher verb for the Control_Vector_Translate verb
CVARXCVL, CVARXCVR	Used to encrypt special control values in the Cryptographic_Variable_Encipher verb for use with the Control_Vector_Translate verb.
Key-generating keys	
DKYGENKY	Used to generate a key based on a key-generating key.
KEYGENKY	Used to generate or derive other keys.

Usually, there is a default control-vector associated with each of the key types previously listed; see Table 75 on page 387. The bits in positions 16 - 22 and 33 - 37 generally have different meanings for every 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 key class in the service's key-type variable. This does not apply to all key-type classes.

You can use the default control-vector for a key type, or you can create a more restrictive control-vector. The default control-vector for a key type provides basic key-separation functions. Optional usage restrictions can further tighten the security of the system.

The cryptographic subsystem creates a default control vector for a key type when you use the Key_Generate verb and specify a null key token and a key-type in the key_type parameter. Also, when you import or export a key, you can 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 or off 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 often obscure.

You can obtain the value for a control vector in one of several ways:

- To use a default-value control vector, obtain the value from Table 75.
- See "Specifying a control-vector-base value" on page 391. The material presents an ordered set of tasks to enable you to create the value for a control vector.
- Use the Key_Token_Build verb or the Control_Vector_Generate verb and keywords to construct a control vector and incorporate this control vector into a key token. See Figure 5 on page 132.

Table 75. Key-type default control-vector values

Key type	Control vector hexadecimal value for a single-length key, or left half of a double-length key	Control vector hexadecimal value for right half of a double-length key
CIPHER	`00 03 71 00  03 00 00 00`	

DATA
(Internal)
(External)
	(single-length)
	`00 00 7D 00  03 00 00 00`
	`00 00 00 00  00 00 00 00`	

DATA
(Internal)
(External)
	(double-length)
	`00 00 7D 00  03 41 00 00`
	`00 00 00 00  00 00 00 00`
		
		`00 00 7D 00  03 21 00 00`
		`00 00 00 00  00 00 00 00`

DATAC	`00 00 71 00  03 41 00 00`	`00 00 71 00  03 21 00 00`
DATAM	`00 00 4D 00  03 41 00 00`	`00 00 4D 00  03 21 00 00`
DATAMV	`00 00 44 00  03 41 00 00`	`00 00 44 00  03 21 00 00`
DECIPHER	`00 03 50 00  03 00 00 00`	
DKYGENKY	`00 71 44 00  03 41 00 00`	`00 71 44 00  03 21 00 00`
ENCIPHER	`00 03 60 00  03 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
(single-length)
(double-length)
	
	`00 05 4D 00  03 00 00 00`
	`00 05 4D 00  03 41 00 00`
		
		`00 05 4D 00  03 21 00 00`

Table 75. Key-type default control-vector values (continued)

Key type	**Control vector hexadecimal value for a single-length key, or left half of a double-length key**	**Control vector hexadecimal value for right half of a double-length key**

MACVER
(single-length)
(double-length)
	
	`00 05 44 00  03 00 00 00`
	`00 05 44 00  03 41 00 00`
		
		`00 05 44 00  03 21 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`

```
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 │
│↑        │        │        │        │        │        │        │      ↑ │
│└─Most significant bit     │        │        │    Least significant bit─┘│
│         │        │        │        │        │        │        │        │
│Common bits       │        │        │        │        │        │        │
│         │        │        │        │        │        │        │        │
│         │        │        │   ┌─────────────┴─Anti─variant bits       │
│         │        │        │   │    │    │   │        │        │        │
│....uu.P │.......P│.E.....P│......0P│......1P│fff.K..P│.......P│.....u.P│
│    ││ │ │        │ │      │        │        │──┬ │    │        │     │  │
│    ││ │ │        │ └E=XPORT─OK     │        │  │ └K=KEY─PART  │     │  │
│    ││ └P=Even parity      │        │        │  └Key─form      │     │  │
│    ││   │        │        │        │        │        │        │     │  │
│    │└u5──UDX5    │        │        │        │        NOT-CCA─u61─┘  │
│    └─u4──UDX4    │        │        │        │        │        │        │
│         │        │        │        │        │        │        │        │
│Key─encrypting keys        │        │        │        │        │        │
│         │        │┌g=IMEX │        │        │        │        │        │
│         │        ││┌k=OPEX│        │        │        │        │        │
│         │        │││┌s=EXEX        │        │        │        │        │
│         │        ││││┌i=EXPORT     │        │        │        │        │
│         │        │││││┌x=XLATE     │        │        │        │        │
│EXPORTER │        ││││││  │        │        │        │        │        │
│0000uu00 │01000001│0EgksixP│00000000│00000011│FFF0K00P│00000000│00000u00│
│OKEYXLAT │        │        │        │        │        │        │        │
│0000uu00 │01000001│0E00001P│00000000│00000011│FFF0K00P│00000000│00000u00│
│IKEYXLAT │        │        │        │        │        │        │        │
│0000uu00 │01000010│0E00001P│00000000│00000011│FFF0K00P│00000000│00000u00│
│IMPORTER │        │        │        │        │        │        │        │
│0000uu00 │01000010│0EgksixP│00000000│00000011│FFF0K00P│00000000│00000u00│
│         │        │ │││││  │        │        │        │        │        │
│         │        │ ││││└x=XLATE    │        │        │        │        │
│         │        │ │││└i=IMPORT    │        │        │        │        │
│         │        │ ││└s=IMIM│      │        │        │        │        │
│         │        │ │└k=OPIM │      │        │        │        │        │
│         │        │ └g=IMEX  │      │        │        │        │        │
│         │        │        │        │        │        │        │        │
│Data operation keys        │        │        │        │        │        │
│DATA     │        │        │        │        │        │        │        │
│0000uu00 │00000000│0Eedmv0P│00000000│00000011│fff0K00P│00000000│00000u00│
│DATAC    │        │        │        │        │        │        │        │
│0000uu00 │00000000│0E11000P│00000000│00000011│FFF0K00P│00000000│00000u00│
│DATAM    │        │        │        │        │        │        │        │
│0000uu00 │00000000│0E00110P│00000000│00000011│FFF0K00P│00000000│00000u00│
│DATAMV   │        │        │        │        │        │        │        │
│0000uu00 │00000000│0E00010P│00000000│00000011│FFF0K00P│00000000│00000u00│
│         │        │        │        │        │        │        │        │
│CIPHER   │        │        │        │        │        │        │        │
│0000uu00 │00000011│0E11000P│00000000│00000011│fff0K00P│00000000│00000u00│
│DECIPHER │        │        │        │        │        │        │        │
│0000uu00 │00000011│0E01000P│00000000│00000011│fff0K00P│00000000│00000u00│
│ENCIPHER │        │        │        │        │        │        │        │
│0000uu00 │00000011│0E10000P│00000000│00000011│fff0K00P│00000000│00000u00│
│         │        │        │        │        │        │        │        │
│MAC      │        │        │        │        │        │        │        │
│ccccuu00 │00000101│0E00110P│00000000│00000011│fff0K00P│00000000│00000u00│
```

Figure 24. Control-vector-base bit map (Part 1 of 3)

```
 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 │
│MACVER
│ccccuu00│00000101│0E00010P│00000000│00000011│fff0K00P│00000000│00000u00│
│────    │        │        │        │        │        │        │        │
│ ├─0000 ANY─MAC  │        │        │        │        │        │        │
│ ├─0001 ANSIX9.9 │        │        │        │        │        │        │
│ ├─0010 CVVKEY─A │        │        │        │        │        │        │
│ ├─0011 CVVKEY─B │        │        │        │        │        │        │
│ └─0100 AMEX─CSC │        │        │        │        │        │        │
│        │        │        │        │        │        │        │        │
│        │        │        │        │        │        │        │        │
│SECMSG  │        │        │        │        │        │        │        │
│0000uu00│00001010│0Ekp000P│00000000│00000011│FFF0K00P│00000000│00000u00│
│        │        │  │└─SMPIN       │        │        │        │        │
│        │        │  └──SMKEY       │        │        │        │        │
│        │        │        │        │        │        │        │        │
│PIN processing keys       │        │        │        │        │        │
│0000 NO─SPEC     │           Prohibit offset:        │        │        │
│0001 IBM─PIN/IBM─PINO         NOOFFSET───┐│        │        │        │
│0010 VISA─PVV    │        │        │     │  │        │        │        │
│0011 INBK─PIN    │        │        │     │  │        │        │        │
│0100 GBP─PIN/GBP─PINO     │        │     │  │        │        │        │
│0101 NL─PIN─1    │        │        │     │  │        │        │        │
│────    │        │        │        │     │  │        │        │        │
│ ↓   PINGEN      │        │        │     │  │        │        │        │
│aaaauu0P│00100010│0E.....P│00000000│00000o1P│FFF0K00P│00000000│00000u00│
│────    │CPINGEN───┘│││   │        │     │  │        │        │        │
│ │      │EPINGENA───┘││   │        │        │        │        │        │
│ │      │EPINGEN──────┘│  │        │     │  │        │        │        │
│ │      │CPINGENA──────┘  │        │        │        │        │        │
│ │      │EPINVER─────────┐│        │     │  │        │        │        │
│ │      │        │       ││        │        │        │        │        │
│ ↓   PINVER      │       ││        │     │  │        │        │        │
│aaaauu0P│00100010│0E00001P│00000000│00000o1P│FFF0K00P│00000000│00000u00│
│        │        │        │        │        │        │        │        │
│        │        │  ┌──────EPINVER  │        │        │        │        │
│        │        │  │ ┌────CPINGENA│        │        │        │        │
│IPINENC │        │  │ │   │        │        │        │        │        │
│0000uu00│00100001│0E0..trP│00000000│00000011│FFF0K00P│00000000│00000u00│
│        │        │      ││ │        │        │        │        │        │
│OPINENC │        │      ││ │        │        │        │        │        │
│0000uu00│00100100│0E..0trP│00000000│00000011│FFF0K00P│00000000│00000u00│
│        │   CPINENC─┘│  │└──REFORMAT│        │        │        │        │
│        │   EPINGEN──┘  └───TRANSLAT│        │        │        │        │
│        │        │        │        │        │        │        │        │
│Cryptographic variable─encrypting keys     │        │        │        │
│0000uu00│00111111│0EvvvvvP│00000000│00000011│fff0K00P│00000000│00000u00│
│        │        │  ───┬──         │        │        │        │        │
│        │        │     ├──────00000 CVARPINE│        │        │        │
│        │        │     ├──────00001 CVARDEC │        │        │        │
│        │        │     ├──────00010 CVARXCVL│        │        │        │
│        │        │     ├──────00011 CVARXCVR│        │        │        │
│        │        │     └──────00100 CVARENC │        │        │        │
│        │        │        │        │        │        │        │        │
│Key─generating keys       │        │        │        │        │        │
│KEYGENKY│        │        │        │        │        │        │        │
│0000uu00│01010011│0E..000P│00000000│00000011│FFF0K00P│00000000│00000u00│
│        │        │  │└─CLR8─ENC    │        │        │        │        │
│        │        │  └──UKPT        │        │        │        │        │
```

Figure 24. Control-vector-base bit map (Part 2 of 3)

```
 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 │
│DKYGENKY                                                              │
│0000uu00│0111sssP│0E0vvvvP│00000000│00000011│FFF0K00P│00000000│00000u00│
│        │    ───┬│    ───┬│        │        │        │        │        │
│   DKYL0 000───┤│       ├──0001 DDATA       │        │        │        │
│   DKYL1 001───┤│       ├──0010 DMAC        │        │        │        │
│   DKYL2 010───┤│       ├──0011 DMV         │        │        │        │
│   DKYL3 011───┤│       ├──0100 DIMP        │        │        │        │
│   DKYL4 100───┤│       ├──0101 DEXP        │        │        │        │
│   DKYL5 101───┤│       ├──0110 DPVR        │        │        │        │
│   DKYL6 110───┤│       ├──1000 DMKEY       │        │        │        │
│   DKYL7 111───┘│       ├──1001 DMPIN       │        │        │        │
│        │       │       └──1111 DALL        │        │        │        │
│        │       │        │        │        │        │        │        │
```

Figure 24. Control-vector-base bit map (Part 3 of 3)

Key-form bits, ‘fff’ and ‘FFF’

The key-form bits, 40 - 42, and for a double-length key, bits 104 - 106, are designated ‘fff’ and ‘FFF’ in the preceding diagram. These bits can have these values:

000	Single-length key (only ‘fff’, not ‘FFF’)
010	Double-length key, left half
001	Double-length key, right half

The bits can also have the following values in some CCA implementations although not created in the IBM 4758 or IBM 4764 implementations:

110	Double-length key, left half, halves guaranteed unique
101	Double-length key, right half, halves guaranteed unique

Specifying a control-vector-base value

You can determine the value of a control vector by working through the following series of questions:

1. Begin with a field of 64 bits (8 bytes) set to B'0'. The most significant bit is referred to as bit 0. Define the key type and subtype (bits 8 – 14), as follows:
   - The main key-type bits (bits 8 – 11). Set bits 8 - 11 to one of the following values:

Bits 8 – 11	**Main key type**
0000	Data operation keys, SECMSG secure messaging keys
0010	PIN keys
0011	Cryptographic variable-encrypting keys
0100	Key-encrypting keys
0101	KEYGENKY key-generating keys
0111	DKYGENKY key-generating keys

   - The key subtype bits (bits 12 – 14). Set bits 12 – 14 to one of the following values:

Bits 12 – 14	Key subtype
Data operation keys	
000	Compatibility key (DATA)
001	Confidentiality key (CIPHER, DECIPHER, or ENCIPHER)
010	MAC key (MAC or MACVER)
101	SECMSG secure messaging keys
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	KEYGENKY key-generating keys
sss	In DKYGENKY key-generating keys, *sss* is the count minus one of the number of diversifications used to obtain the final, non-diversification key. See "Diversifying keys" on page 142. The Key_Token_Build verb can set the *sss* bits when you supply the **DKYL0**, ..., and **DKYL7** keywords.
Cryptographic variable-encrypting keys	
111	Cryptographic variable-encrypting key (CVAR....)

2. For key-encrypting keys,
   - The key-encrypting key-limiting bits, previously described as bits "*hhh*, bits 35 - 37", are not supported in any current release of the coprocessor CCA support.
   - The key-generating usage bits (gks, bits 18 – 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 24 on page 389). 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 you set the '*i*' bit 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 you set the '*x*' bit to 1, the associated key-encrypting key can be used in the Key_Translate verb. The Control_Vector_Generate verb can set the '*ix*' bits to 1 when you supply the **IMPORT, EXPORT**, and **XLATE** keywords.
   - The key-form bits (fff or FFF, bits 40 – 42). The key-form bits indicate how the key was generated and how the control vector participates in multiple-enciphering. To consist of two different 8-byte values, set the key-form bits to B'110'; to 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 13 on page 395.

3. For the DATA-class keys (DATA, DATAC, DATM, DATAMV) set the “edmv” bits (bits 18 - 21) to one to respectively enable **e**ncipher, **d**ecipher, **m**ac-generation, and mac-**v**erification operations.
4. For the cipher-class keys (CIPHER, DECIPHER, ENCIPHER, DATA, DATAC) (bits 18 and 19). Set bit 18 is to 1 for the key to encipher data. Set bit 19 is to 1 for the key to decipher data.
5. For MAC, MACVER, DATAM, and DATAMV keys, set the following bits:
   - The MAC control bits (bits 20 and 21). For a MAC generation key, set bits 20 and 21 to B'11'. For a MAC verification key, set bits 20 and 21 to B'01'.
   - The key-form bits (fff or FFF, bits 40 – 42). For a single-length key, set the bits to B'000'. For a double-length key, set the bits to B'010'.
6. For SECMSG keys, set one or both of the following bits:
   - Set the SMKEY bit (*k*, bit 18) to enable this key type to operate in secure message services that imbed a key.
   - Set the SMPIN bit (*p*, bit 19) to enable this key type to operate in secure message services that imbed a PIN.

   Verbs Secure_Messaging_for_Keys and Secure_Messaging_for_PINs, as used with, for example, EMV smart cards. These are currently only supported in the IBM eServer iSeries release 2.50 CCA support.
7. For PINGEN and PINVER keys, set the following bits:
   - The PIN-calculation method bits (*aaaa*, bits 0 - 3). Set these bits to one of the following values:

Bits 0 – 3	Calculation method keyword	Description
0000	NO-SPEC	A key with this control vector can be used with any PIN-calculation method.
0001	IBM-PIN or IBM-PINO	A key with this control vector can be used only with the IBM PIN or PIN-Offset calculation method.
0010	VISA-PVV	A key with this control vector can be used only with the VISA-PVV calculation method.
0100	GBP-PIN or GBP-PINO	A key with this control vector can be used only with the German Banking Pool PIN or PIN-Offset calculation method.
0011	INBK-PIN	A key with this control vector can be used only with the Interbank PIN-calculation method.
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. Set the bit to 1 to prevent operation with the IBM 3624 PIN Offset calculation method and the IBM German Bank Pool PIN Offset calculation-method.
8. For PINGEN, IPINENC, and OPINENC keys, set bits 18 – 22 to indicate whether the key can be used with the following verbs; for the bit numbers, see Figure 24 on page 389:

Verb allowed	Bit name	Bit
Clear_PIN_Generate	CPINGEN	18
Encrypted_PIN_Generate_Alternate	EPINGENA	19

Verb allowed	Bit name	Bit
Encrypted_PIN_Generate	EPINGEN	20 for PINGEN 19 for OPINENC
Clear_PIN_Generate_Alternate	CPINGENA	21 for PINGEN 20 for IPINENC
Encrypted_Pin_Verify	EPINVER	19
Clear_PIN_Encrypt	CPINENC	18

9. For the IPINENC and OPINENC PIN-block ciphering keys, do the following:
   - Set the TRANSLAT bit (*t*, bit 21) to 1 to permit the key to be used in the PIN_Translate verb. The Control_Vector_Generate verb can set the TRANSLAT bit to 1 when you supply the **TRANSLAT** keyword.
   - Set the REFORMAT bit (*r*, bit 22) to 1 to permit the key to be used in the PIN_Translate verb. The Control_Vector_Generate verb can set the REFORMAT bit and the TRANSLAT bit to 1 when you supply the **REFORMAT** keyword.
10. For the cryptographic variable-encrypting keys (bits 18 - 22), set the variable-type bits (bits 18 - 22) to one of the following values:

Bits 18 – 22	Key type	Description
00000	CVARPINE	Used in the Encrypted_PIN_Generate_Alternate verb to encrypt a clear PIN
00010	CVARXCVL	Used in the Control_Vector_Translate verb to decrypt the left mask array
00011	CVARXCVR	Used in the Control_Vector_Translate verb to decrypt the right mask array
00100	CVARENC	Used in the Cryptographic_Variable_Encipher verb to encrypt an unformatted PIN

11. For KEYGENKY key-generating keys, set the following bits:
    - Set bit 19 to 1 if the key is be used in the Diversified_Key_Generate (CSNBDKG) verb to generate a diversified key.
    - Bit 18 is reserved for unique key per transaction (UKPT) usage.
12. For DKYGENKY key-generating keys that are used in the **TDES-ENC** or **TDES-DEC** mode of the Diversified_Key_Generate (CSNBDKG) verb, set bits 19 - 22 according to the type of final key that is to be obtained as shown in the following table:

Bits 19 – 22	Keyword	To obtain
0001	**DDATA**	single- or double-length DATA key
0010	**DMAC**	single- or double-length MAC key
0011	**DMV**	single- or double-length MACVER key
0100	**DIMP**	IMPORTER key
0101	**DEXP**	EXPORTER key
0110	**DPVR**	PIN verify key
1000	**DMKEY**	double-length SMKEY SECMSG key
1001	**DMPIN**	double-length SMPIN SECMSG key
1111	**DALL**	any of the above

13. For all keys, set the following bits:
    - The export bit (*E*, bit 17). Set this bit to 0 to prevent the receiver of a key from exporting or translating the key for use in another cryptographic subsystem. Once this bit is set to 0, it cannot be set to 1 by any verb other than the Control_Vector_Translate verb. The Prohibit_Export verb can reset the export bit.
    - The key-part bit (*K*, bit 44). Set the key-part bit to 1 in a control vector associated with a key part. When the final key part is combined with previously accumulated key parts, the key-part bit in the control vector for the final key part is set to 0. The Control_Vector_Generate verb can set the key-part bit to 1 when you supply the **KEY-PART** keyword.
    - For the user definition bits (*uu...u*, bits 4, 5, and 61), do the following:
        – Set either or both *u*4 and *u*5 as required by a user-defined extension (UDX). These bits are reserved for use by UDX.
        – Set the *u*61 bit to 1 if the key is only permitted to function in a user-defined extension. That is, the key is not be usable in CCA services defined in this publication. Keys with bits 4, 5, or 61 set on can be generated, and can be imported and exported (provided other conditions permit).
    - 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 - 127. If bits 40 - 42 are B'000' (single-length key), set bits 64 - 127 to 0. Otherwise, copy bits 0 - 63 into bits 64 - 127 and set bits 105 and 106 to B'01'.
    - Set the parity bits (low-order bit of each byte, bits 7, 15, ..., 127). These bits contain the parity bits (P) of the control vector. Set the parity bit of each byte so the number of zero-value bits in the byte is an even number.

Changing control vectors

Use the following techniques to change the control vector associated with a key:

Pre-exclusive-OR
Use this technique to import or export a key from a cryptographic node if you can exclusive-OR one or more bit patterns into the value of the key-encrypting key used to import the key.

Control_Vector_Translate Verb
Use the Control_Vector_Translate verb to change the control vector of an external key.

Note: An external key is a key enciphered by a KEK other than the master key.

Changing control vectors with the pre-exclusive-OR technique

Use the pre-exclusive-OR technique to change a key's control vector when exporting or importing the key from or to a CCA 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* (which quantity is discussed later).
- 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 where the CV_k contains the value of a default CV for that key type, has a CV_{k1} value equal to binary zero.

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 $KEK \oplus CV_{k1}$, and 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 has a changed control vector.

If you need to change the control vector for a key when importing the key, the Key_Import verb works in a similar manner. You exclusive-OR the actual control vector value and the desired control vector value for the imported key into the value of the key-encrypting key. Then when you call the Key_Import verb, be sure that the source-key token contains the control vector of the desired target key.

If you are processing a double-length key, you must process the key twice, using the key-encrypting key modified by different values each appropriate to a key half. Then you concatenate the resulting two correct key-halves.

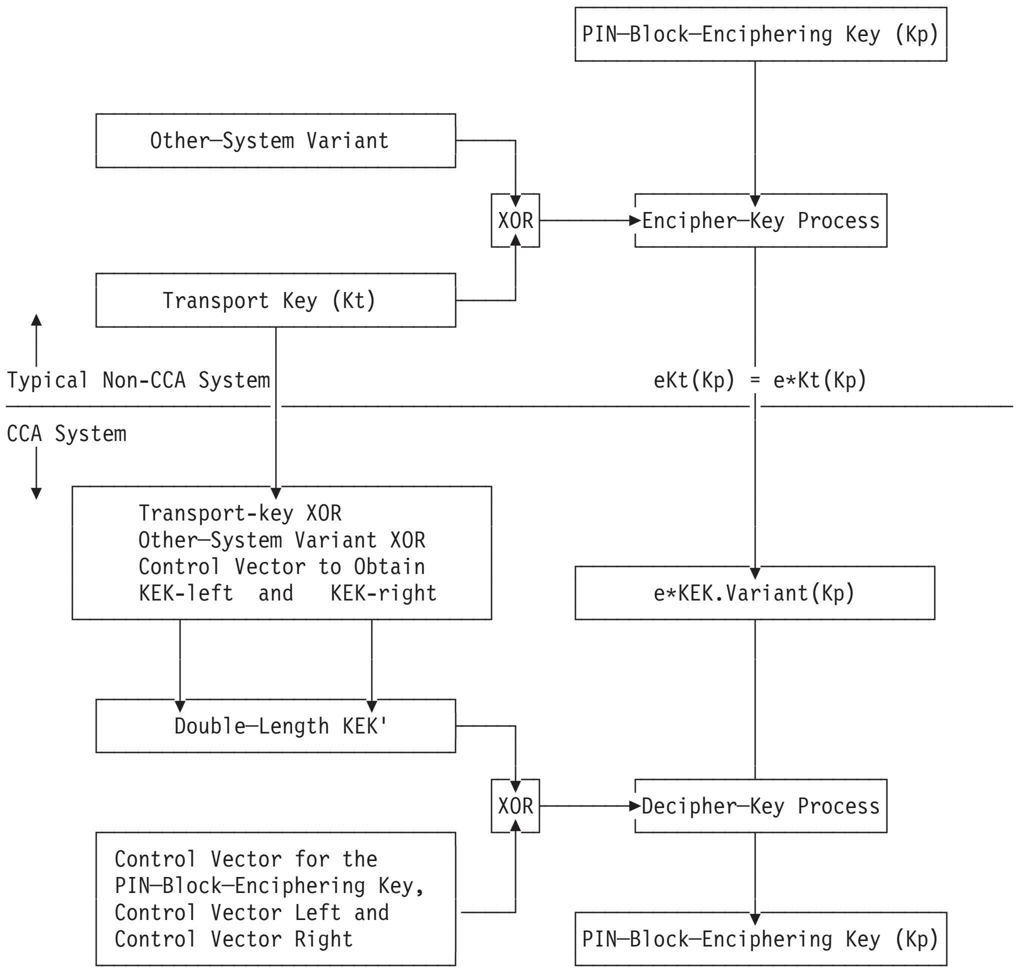

PIN—Block—Enciphering Key (Kp)
Other—System Variant
XOR
Encipher—Key Process
Transport Key (Kt)
Typical Non-CCA System
eKt(Kp) = e*Kt(Kp)
CCA System
Transport-key XOR Other—System Variant XOR Control Vector to Obtain KEK-left and KEK-right
e*KEK.Variant(Kp)
Double—Length KEK'
XOR
Decipher—Key Process
Control Vector for the PIN—Block—Enciphering Key, Control Vector Left and Control Vector Right
PIN—Block—Enciphering Key (Kp)

Figure 25. Exchanging a key with a non-control-vector system

Figure 25 shows a typical situation. In a non-CCA system, a PIN-block encrypting key is singly encrypted by a transport key. No control vector modifies the value of the transport key, Kt, used to encrypt the PIN-block encrypting key, Kp. The resulting cryptogram can be designated eKt(Kp). Because triple-encryption is the same as single-encryption when both halves of the encrypting key are equal, eKt(Kp) = e*Kt(Kp).

In the CCA system, a PIN-block decrypting key is an IPINENC key and must be double length. If both halves of the double-length key are the same, the IPINENC key effectively performs single encryption. You must import both halves of the target IPINENC key in different steps and combine the result to obtain the desired result key.

1. Create two key-encrypting keys to import each half of the target input PIN-block encrypting key. When you receive key Kt, store this as two different keys:

   e*Km⊕CViml(Kt⊕CVil) || e*Km⊕CVimr(Kt⊕CVil)

   where:

   – CViml is the control vector for the left half of an IMPORTER key

– CVimr is the control vector for the right half of an IMPORTER key
– CVil is the control vector for the left half of the target input PIN-block encrypting key

e*Km⊕CViml(Kt⊕CVir) ∥ e*Km⊕CVimr(Kt⊕CVir)

where:

– CVir is the control vector for the right half of the target input PIN-block encrypting key

2. Use the Key_Token_Build verb to build source and target key tokens with:
   - eKt(Kp) ∥ eKt(Kp)
   - CVil ∥ CVil
3. Use Key_Import and the first of the IMPORTER keys to import the left half of the target key (discard the right half).
4. Use the Key_Token_Build verb to build source and target key tokens with:
   - eKt(Kp) ∥ eKt(Kp)
   - CVir ∥ CVir
5. Use Key_Import and the second of the IMPORTER keys to import the right half of the target key (discard the left half).
6. Concatenate the two key halves. You can use the Key_Token_Parse and Key_Token_Build verbs to parse and build the required key tokens.

Changing control vectors with the Control_Vector_Translate verb

Do the following when using the Control_Vector_Translate verb:

- Provide the control information for testing the control vectors of the source, target, and key-encrypting keys to ensure that only sanctioned changes can be performed
- Select the key-half processing mode.

Providing the control information for testing the control vectors

To minimize your security exposure, the Control_Vector_Translate verb requires (*mask array* information to limit the range of allowable control vector changes. To ensure that this verb is used only for authorized purposes, the source-key control vector, target-key control vector, and key-encrypting key (KEK) control vector must pass specific tests. The tests on the control vectors are performed within the secured cryptographic engine.

The tests consist of evaluating four logic expressions, the results of which must be a string of binary zeros. The expressions operate bit-for-bit on information that is contained in the mask arrays and in the portions of the control vectors associated with the key or key-half that is being processed. If any of the expression evaluations do not result in all zero bits, the verb is ended with a *control vector violation* return code 8 and reason code 39. See Figure 26. Only the 56 bit positions that are associated with a key value are evaluated. The low-order bit that is associated with key parity in each key-byte is not evaluated.

Mask array preparation

A mask array consists of seven 8-byte elements: A_1, B_1, A_2, B_2, A_3, B_3, and B_4. You choose the values of the array elements such that each of the following four expressions evaluates to a string of binary zeros. (See Figure 26 on page 400.) Set the A bits to the value that you require for the corresponding control vector bits. In expressions 1 through 3, set the B bits to select the control vector bits to be evaluated. In expression 4, set the B bits to select the source and target control vector bits to be evaluated. Also, use the following control vector information:

C_1 is the control vector associated with the left half of the KEK.

C_2 is the control vector associated with the source key, or selected source-key half/halves.

C_3 is the control vector associated with the target key or selected target-key half/halves.

1. (C_1 exclusive-OR A_1) logical-AND B_1

   This expression tests whether the KEK used to encipher the key meets your criteria for the desired translation.
2. (C_2 exclusive-OR A_2) logical-AND B_2

   This expression tests whether the control vector associated with the source key meets your criteria for the desired translation.
3. (C_3 exclusive-OR A_3) logical-AND B_3

   This expression tests whether the control vector associated with the target key meets your criteria for the desired translation.
4. (C_2 exclusive-OR C_3) logical-AND B_4

   This expression tests whether the control vectors associated with the source key and the target key meet your criteria for the desired translation.

Encipher two copies of the mask array, each under a different cryptographic-variable key using key type CVARENC. To encipher each copy of the mask array, use the Cryptographic_Variable_Encipher verb. Use two different keys so that the enciphered-array copies are unique values. When using the Control_Vector_Translate verb, the mask_array_left parameter and the mask_array_right parameter identify the enciphered mask arrays. The array_key_left parameter and the array_key_right parameter identify the target keys for deciphering the mask arrays. The array_key_left key must have a key type of CVARXCVL and the array_key_right key must have a key type of CVARXCVR. The cryptographic process deciphers the arrays and compares the results; for the verb to continue, the deciphered arrays must be equal. If the results are not equal, the verb returns the return code 8 and reason code 385 for data that is not valid.

When using the Key_Generate verb to create the key pairs CVARENC-CVARXCVL and CVARENC-CVARXCVR, the hardware requires the Generate_Key_Set_Extended command to be enabled. Each key in the key pair must be generated for a different node. The CVARENC keys are generated for, or imported into, the node where the mask array is enciphered. After enciphering the mask array, destroy the enciphering key. The CVARXCVL and CVARXCVR keys are generated for, or imported into, the node where the Control_Vector_Translate verb is performed.

If using the **BOTH** keyword to process both halves of a double-length key, remember that bits 41, 42, 104, and 105 are different in the left and right halves of the CCA control vector and must be ignored in your mask-array tests (that is, make the corresponding B_2 or B_3 bits equal to zero).

When the control vectors pass the masking tests, the verb does the following:

- Deciphers the source key. In the decipher process, the verb uses a key that is formed by the exclusive-OR of the KEK and the control vector in the key token variable the source_key_token parameter identifies.
- Enciphers the deciphered source key. In the encipher process, the verb uses a key that is formed by the exclusive-OR of the KEK and the control vector in the key token variable the target_key_token parameter identifies.

- Places the enciphered key in the key field in the key token variable the target_key_token parameter identifies.

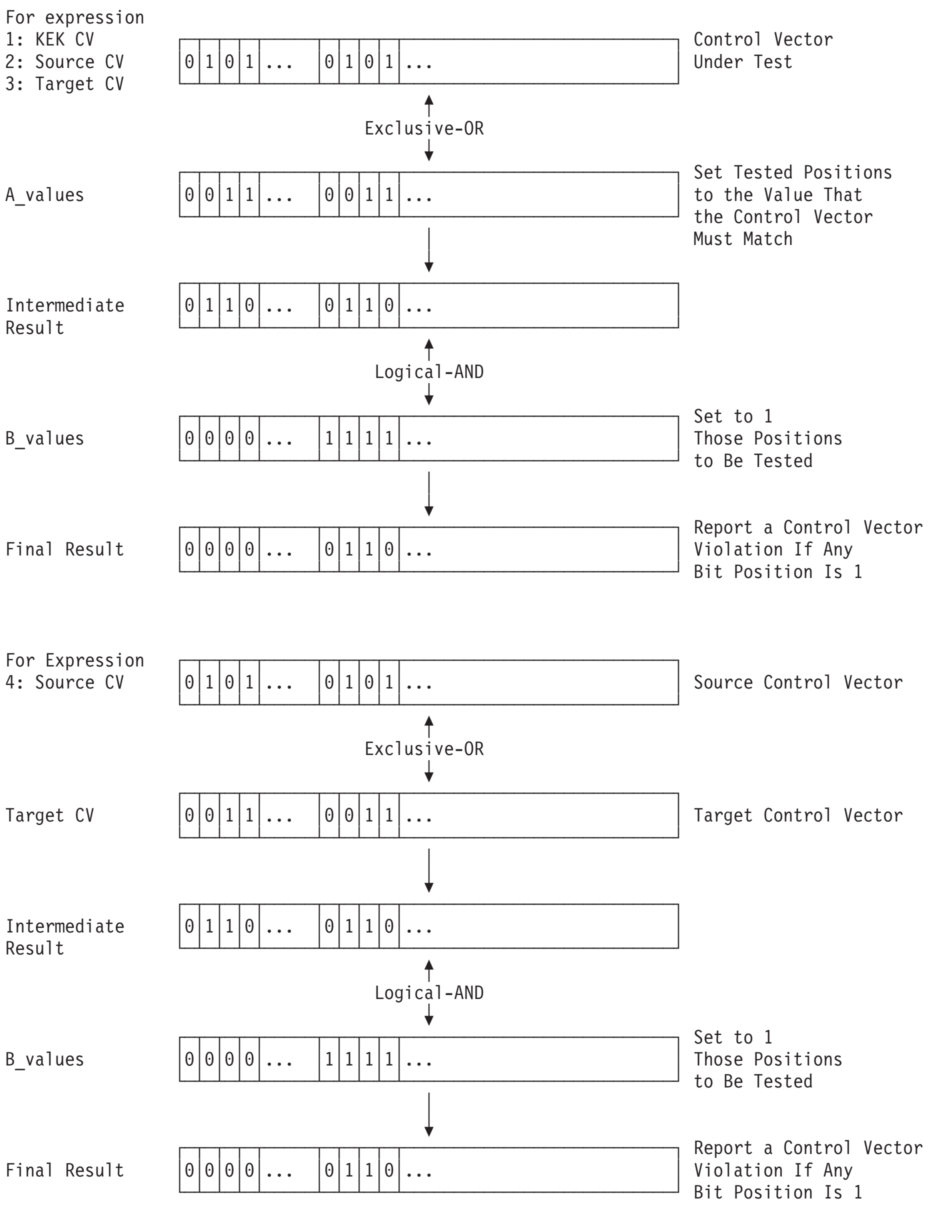


Figure 26. Control_Vector_Translate verb mask_array processing

Selecting the key-half processing mode

The Control_Vector_Translate verb rule-array keywords determine which key halves are processed in the verb call, as shown in Figure 27 on page 401. In this figure, CHANGE-CV means the requested control vector translation change; LEFT and RIGHT mean the left and right halves of a key and its control vector.

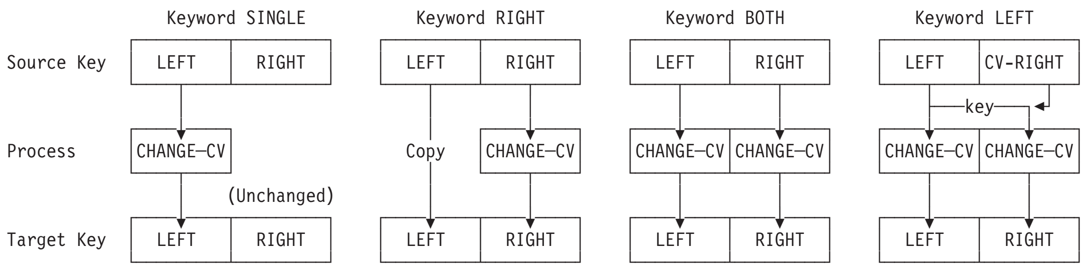


Figure 27. Control_Vector_Translate verb process

The following list provides definitions for keywords:

SINGLE
This keyword causes the control vector of the left half of the source key to be changed. The updated key half is placed into the left half of the target key in the target key token. The right half of the target key is unchanged.

The **SINGLE** keyword is useful when processing a single-length key, or when first processing the left half of a double-length key (to be followed by processing the right half).

RIGHT
This keyword causes the control vector of the right half of the source key to be changed. The updated key half is placed into the right half of the target key of the target key token. The left half of the source key is copied unchanged into the left half of the target key in the target key token.

BOTH This keyword causes the control vector of both halves of the source key to be changed. The updated key is placed into the target key in the target key token.

A single set of control information must permit the control vector changes applied to each key half. Normally, control vector bit positions 41, 42, 105, and 106 are different for each key half. Therefore, set bits 41 and 42 to B'00' in mask array elements B_1, B_2, and B_3.

You can verify that the source and target key tokens have control vectors with matching bits in bit positions 40-42 and 104-106, the form field bits. Ensuring that bits 40-42 of mask array B_4 are set to B'111'.

LEFT This keyword enables you to supply a single-length key and obtain a double-length key. The source key token must contain:
- The KEK-enciphered single-length key
- The control vector for the single-length key (often this is a null value)
- A control vector, stored in the source token where the right-half control vector is normally stored, used in decrypting the single-length source key when the key is being processed for the target right half of the key.

The verb first processes the source and target tokens as with the **SINGLE** keyword. Then the source token is processed using the single-length enciphered key and the source token right-half control vector to obtain the actual key value. The key value is then enciphered using the KEK and the control vector in the target token for the right-half of the key.

This approach is frequently of use when you must obtain a double-length CCA key from a system that only supports a single-length key. For example when processing PIN keys or key-encrypting keys received from non-CCA systems.

To prevent the verb from ensuring that each key byte has odd parity, you can specify the **NOADJUST** keyword. If you do not specify the **NOADJUST** keyword, or if you specify the **ADJUST** keyword, the verb ensures that each byte of the target key has odd parity.

When the target key-token CV is null

When you use any of the **LEFT**, **BOTH**, or **RIGHT** keywords, and when the control vector in the target key token is null (all B'0'), then bit 0 in byte 59 of the target version X'01' key token is set to B'1' to indicate that this is a double-length DATA key.

Control_Vector_Translate Example

As an example, consider the case of receiving a single-length PIN-block encrypting key from a non-CCA system. Often such a key is encrypted by an unmodified transport key (no control vector is used). In a CCA system, an inbound PIN encrypting key is double-length.

First, use the Key_Token_Build verb to insert the single-length key value into the left-half key-space in a key token. Specify **USE-CV** as a key type and a control vector value set to 16 bytes of X'00'. Also specify **EXTERNAL**, **KEY**, and **CV** keywords in the rule array. This key token is the source key key-token.

The target key token can also be created using the Key_Token_Build verb. Specify a key type of **IPINENC** and the **NO-EXPORT** rule array keyword.

Second, call the Control_Vector_Translate verb and specify a rule-array keyword of **LEFT**. The mask arrays can be constructed as follows:

- A_1 is set to the value of the KEK's control vector, most likely the value of an IMPORTER key, perhaps with the NO-EXPORT bit set. B_1 is set to 8 bytes of X'FF' so that all bits of the KEK's control vector is tested.
- A_2 is set to 8 bytes of X'00', the null value of the source key control vector. B_2 is set to 8 bytes of X'FF' so that all bits of the source-key control vector is tested.
- A_3 is set to the value of the target key's left-half control vector. B_3 is set to X'FFFF FFFF FF9F FFFF'. This causes all bits of the control vector to be tested except for the two (*fff*) bits used to distinguish between the left-half and right-half target-key control vector.
- B_4 is set to 8 bytes of X'00' so that no comparison is made between the source and target control vectors.

Understanding CCA key encryption and decryption processes

This section describes the CCA key-encryption processes:

- CCA DES key encryption
- CCA RSA private key encryption
- Encipherment of DES keys under RSA in PKA92 format
- Encipherment of a DES key-encrypting key under RSA in NL-EPP-5 format.

CCA DES key encryption and decryption processes

With the CCA, multiple enciphering or multiple deciphering a key The process 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 (K*n*) are used in the multiple-encipherment of a clear key, or the multiple-decipherment of an encrypted key; see Figure 28 on page 404 for the formation of K1 through K6 and their use with DES DEA encoding and decoding.

CCA RSA private key encryption and decryption process

RSA private keys are generally encrypted using an EDE algorithm. See “Triple-DES ciphering algorithms” on page 416.

With the CCA Support Program Version 1, a private key in an internal key token encrypted by the master key is encrypted using the EDE3 process. The secret key is deciphered using the DED3 process. A private key in an external key token encrypted by a transport key is encrypted using the EDE2 process. The secret key is deciphered using the DED2 process.

Beginning with the CCA Support Program Version 2, the private key is encrypted using an object protection key (OPK). The OPK is encrypted with the asymmetric master key. For internal keys, the secret key values are then encrypted by the OPK. For external encrypted private keys encryption is provided by the DES transport key. See Table 49 on page 356 and Table 50 on page 357.

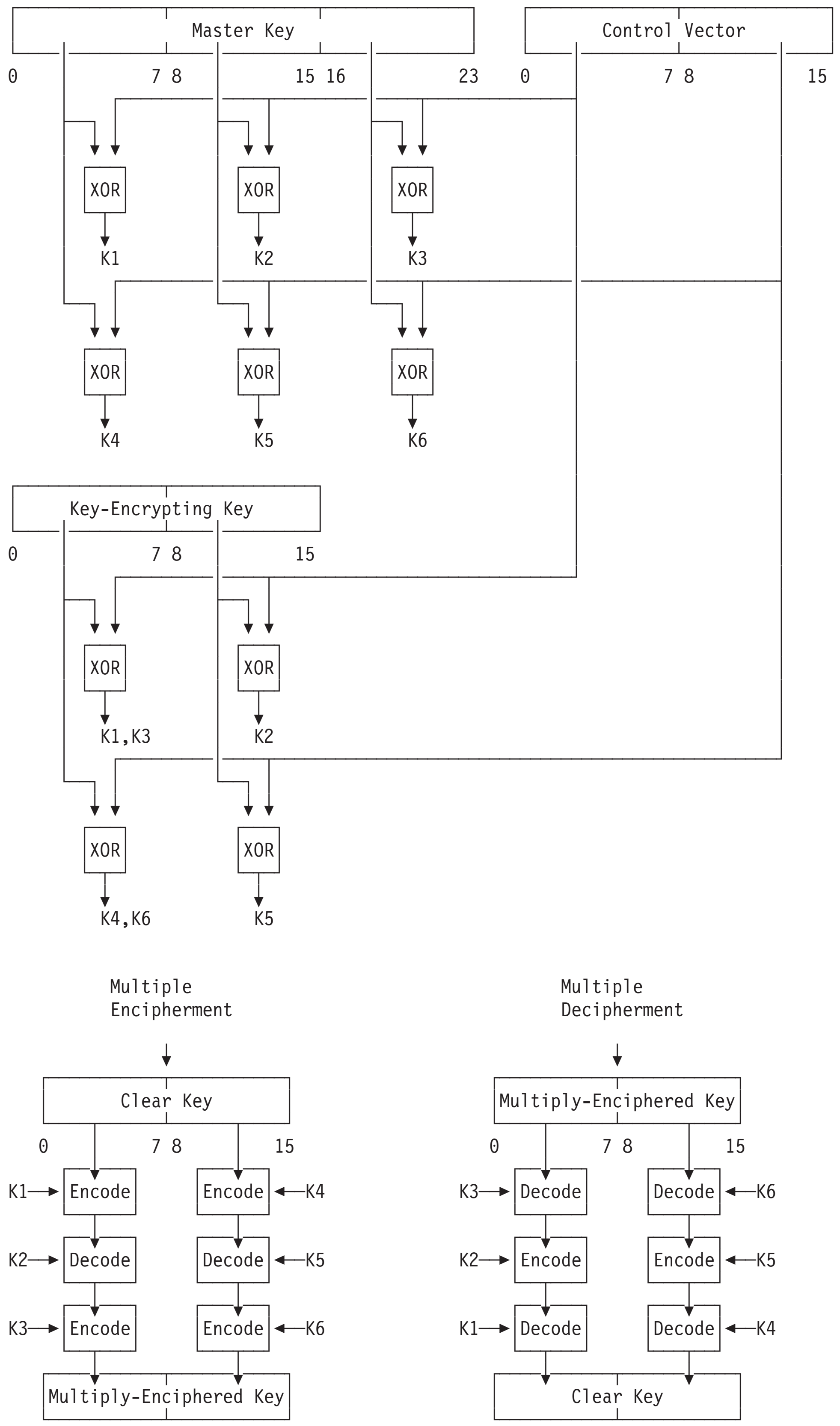


Figure 28. Multiply-enciphering and multiply-deciphering CCA keys

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.

PKA92 key format and encryption process

With the PKA_Symmetric_Key_Export, PKA_Symmetric_Key_Generate, and the PKA_Symmetric_Key_Import verbs you can use 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 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 use the actual RSA-encrypted portion of a TSS PKA92 key token, the AS External Key Block.

Forming an external key block

The PKA96 implementation forms an AS External Key Block by RSA-encrypting a key block using a public key. The key block is formed by padding the key record detailed in Table 76 with zero bits on the left, high-order end of the key record. The process completes the key block with three subprocesses: masking, overwriting, and RSA encrypting.

Table 76. PKA96 clear DES key record

Offset (bytes)	Length (bytes)	Description
The public key modulus is constrained to a multiple of 64 bits in the range of 512 - 1024 bits. 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	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'

Mask subprocess:

1. 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.
2. Exclusive-OR the mask with the key record and call the result PKR.

Overwrite subprocess:

1. Set the high-order bits of PKR to B'01'.
2. Set the low-order bits to B'0110'.
3. Exclusive-OR K and IV.
4. Write the result at offset 45 in PKR.
5. Write IV at offset 53 in PKR. This causes the masked and overwritten PKR to have IV at its original position.

RSA-encrypt subprocess: 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 subprocesses.

Decrypting subprocess: 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 subprocess: Verify that the high-order 2 bits of the PKR record are valued to B'01' and that the low-order 4 bits of the PKR record are valued to B'0110'.

Unmasking subprocess: Set IV to the value of the 8 bytes at offset 53 of the PKR record. There is a variable quantity of padding prior to offset 0. See Table 76 on page 405.

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.

Copy K to offset 45 in the PKR record.

Extraction subprocess: Confirm that:

- The 4 bytes at offset 1 in the key record are valued to X'0000 0000'
- The two control vector fields at offsets 21 and 29 are identical
- If the control vector is an IMPORTER or EXPORTER key class, 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' or B'110', the key is double length. Set the right half of the received key's control vector equal to the left half and reverse bits 41 and 42 in the right half.

The recovered key is at offset 37 and is either 8 or 16 bytes long based on the control vector base bits 40 to 42. If these bits are valued to B'000', the key is single length. If these bits are valued to B'010' or B'110', the key is double length.

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 as shown in Table 77.
2. Insert random padding data into the record.
3. Insert the count of pad bytes plus one.

Table 77. NL-EPP-5 key record format

Offset (bytes)	**Length (bytes)**	**Description**
000	02	Header and null cancellation bytes, X'0B00'
002	08 16	Single length key-encrypting key Double length key-encrypting key
010 or 018		Random padding data
063	01	Padding count byte: • With an RSA key of length 512 bits: X'36' for a single length key-encrypting key, or X'2E' for a double length key-encrypting key • With an RSA key of length 1024 bits: X'76' for a single length key-encrypting key, or X'6E' for a double length key-encrypting key

Appendix D. Algorithms and processes

This section provides processing details for the following aspects of the CCA design:
- Cryptographic key-verification techniques
- Modification detection code (MDC) calculation methods
- Ciphering methods
- Triple-DES algorithms, EDE2 and EDE3
- MAC calculation methods
- Access-control algorithms
- Master-key splitting algorithm
- RSA key-pair generation

Cryptographic key verification techniques

The key-verification implementations described in this document employ several mechanisms for assuring the integrity and value of the key. These topics are discussed:
- Master key verification algorithms
- CCA DES-key and key-part verification algorithm
- Encrypt zeros algorithm

Master key verification algorithms

The IBM 4764 and IBM 4758 implementations employ triple-length master keys (three DES keys) that are 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.

The IBM 4758 Model 2 and Model 23 employ several verification pattern generation methods.

SHA-1 based master key verification method

A SHA-1 hash algorithm is calculated on the quantity X'01' prepended to the 24-byte register contents. The resulting 20-byte hash value is used in the following ways:
- The Key_Test verb uses the first 8 bytes of the 20-byte hash value as the random number variable, and uses the second 8 bytes as the verification pattern.
- A SHA-1 based master-key verification pattern stored in a 2-byte or an 8-byte verification pattern field in a key token consists of the first two or the first 8 bytes of the calculated SHA-1 value.

S/390–based master key verification method

When the first and third portions of the symmetric master key have the same value, the master key is effectively a double-length DES key. In this case, the master key verification pattern (MKVP) is based on this algorithm:
- C = X'4545454545454545'
- $IR = MK_{first\text{-}part} \oplus e_C(MK_{first\text{-}part})$
- $MKVP = MK_{second\text{-}part} \oplus e_{IR}(MK_{second\text{-}part})$

where:
- $e_x(Y)$ is the DES encoding of Y using x as a key
- $\oplus$ represents the bitwise exclusive-OR function.

Version X'00' internal DES key tokens use this 8-byte master key version number.

Asymmetric master key MDC-based verification method

The verification pattern for the asymmetric master keys is based on hashing the value of the master key using the MDC-4 hashing algorithm. The master key is not parity adjusted.

The RSA private key sections X'06' and X'08' use this 16-byte master key version number.

Key token verification patterns

The verification pattern techniques used in the several types of key tokens are:

- DES key tokens:
  - Triple-length master key, key token version X'00': 8-byte SHA-1
  - Triple-length master key, key token version X'03': 2-byte SHA-1
  - Double-length master key, key token version X'00': 8-byte S/390
  - Double-length master key, key token version X'03': 2-byte SHA-1
- RSA key tokens:
  - Private-key section types X'06' and X'08': MDC-based
  - Private-key section types X'02' and X'05': two-byte SHA-1

CCA DES-key verification algorithm

The cryptographic engines provide a method for verifying the value of a DES cryptographic key or key part without revealing information about the value of the key or key part.

The CCA verification method first creates a random number. A one-way cryptographic function combines the random number with the key or key part. The verification method returns the result of this one-way cryptographic function (the *verification pattern*) and the random number.

Note: A one-way cryptographic function is a function in which it is easy to compute the output from a given input, but it is not computationally feasible to compute the input given an output.

For information about how you can use an application program to invoke this verification method, see the verb "Key_Test (CSNBKYT)" on page 183.

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. The leftmost 32 bits of the result are compared to the trial input value or returned from the Key_Test verb.

For a single-length key, the key DEA encodes an 8-byte, all-zero-bits value.

For a double-length key, the key DEA triple-encodes an 8-byte, all-zero-bits value. The left half (high-order half) key encodes the zero-bit value, this result is DEA decoded by the right key half, and that result is DEA encoded by the left key half.

Modification detection code calculation methods

The Modification Detection Code (MDC) calculation method defines a one-way cryptographic function. A one-way cryptographic function is a function in which it is easy to compute the input into output but not easy to compute the output into input. MDC uses DES encryption only and a default key of X'5252 5252 5252 5252 2525 2525 2525 2525'.

The MDC_Generate verb supports four versions of the MDC calculation method that you specify by using one of the keywords shown in Table 78. All versions use the MDC-1 calculation.

Table 78. Versions of the MDC calculation method

Keyword	Version of the MDC calculation
MDC-2, **PADMDC-2**	Specifies two encipherments for each 8-byte input data block.
MDC-4, **PADMDC-4**	Specifies four encipherments for each 8-byte input data block.

When the keywords **PADMDC-2** and **PADMDC-4** are used, the supplied text is *always* padded as follows:

- If the supplied text is less than 16 bytes in length, pad bytes are appended to make the text length equal to 16 bytes.
- If the supplied text is at least 16 bytes in length, pad bytes are appended to make the text length equal to the next-higher multiple of 8 bytes, pad bytes are always added.
- All appended pad bytes, other than the last pad byte, are set to X'FF'.
- The last pad byte is set to a binary value equal to the count of all appended pad bytes (X'01' to X'10').

Use the resulting pad text in the following procedures. The MDC_Generate verb uses these MDC calculation methods. See “MDC_Generate (CSNBMDG)” on page 117 for more information about the MDC_Generate verb.

Notation used in calculations

The MDC calculations use the following notations:

eK(X) Denotes DES encryption of plaintext X using key K

ll	Denotes the concatenation operation
XOR	Denotes the exclusive-OR operation
:=	Denotes the assignment operation
T8<1>	Denotes the first 8-byte block of text
T8<2>	Denotes the second 8-byte block of text, and so on
KD1, KD2, IN1, IN2, OUT1, OUT2	Denotes 64-bit quantities

MDC-1 calculation

The MDC-1 calculation, which is used in the MDC-2 and MDC-4 calculations, consists of the following procedure:

```
MDC-1 (KD1, KD2, IN1, IN2, OUT1, OUT2);
  Set KD1mod := set bit 1 and bit 2 of KD1 to "1" and "0" respectively.
  Set KD2mod := set bit 1 and bit 2 of KD2 to "0" and "1" respectively.
  Set F1 := IN1 XOR eKD1mod(IN1)
  Set F2 := IN2 XOR eKD2mod(IN2)
  Set OUT1 := (bits 0..31 of F1) || (bits 32..63 of F2)
  Set OUT2 := (bits 0..31 of F2) || (bits 32..63 of F1)
End procedure
```

MDC-2 calculation

The MDC-2 calculation consists of the following procedure:

```
MDC-2 (n, text, KEY1, KEY2, MDC);
  For i := 1,2,...,n do
    Call MDC-1(KEY1, KEY2, T8<i>, T8<i>, OUT1, OUT2)
    Set KEY1 := OUT1
    Set KEY2 := OUT2
  End do
  Set output MDC := (KEY1 || KEY2).
End procedure
```

MDC-4 calculation

The MDC-4 calculation consists of the following procedure:

```
MDC-4 (n, text, KEY1, KEY2, MDC);
  For i := 1,2,...n do
    Call MDC-1(KEY1,KEY2,T8<i>,T8<i>,OUT1,OUT2)
    Set KEY1int := OUT1
    Set KEY2int := OUT2
    Call MDC-1(KEY1int,KEY2int,KEY2,KEY1,OUT1,OUT2)
    Set KEY1 := OUT1
    Set KEY2 := OUT2
 End do
 Set output MDC := (KEY1 || KEY2)
End procedure
```

Ciphering methods

The Data Encryption Standard (DES) algorithm defines operations on 8-byte data strings. The DES algorithm is used in many different processes within CCA:
- Encrypting general data
- Triple-encrypting PIN blocks
- Triple-encrypting CCA DES keys
- Triple-encrypting RSA private keys with several processes
- Deriving keys, hashing data, generating CVV values, and so on

The Encipher and Decipher verbs describe how you can request encryption of application data. See the following topic: “General data encryption processes” for a description of the two standardized processes you can use.

In CCA, PIN blocks are encrypted with double-length keys. The PIN block is encrypted with the left-half key, for which the result is decrypted with the right-half key and this result is encrypted with the left-half key.

See “CCA DES key encryption and decryption processes” on page 402, and

“Triple-DES ciphering algorithms” on page 416, which describe how CCA DES keys are enciphered.

General data encryption processes

Although the fundamental concepts of enciphering and deciphering data are simple, different methods exist to process data strings that are not a multiple of 8 bytes in length. Two widely used methods for enciphering general data are defined in these ANSI standards:

- ANSI X3.106 (CBC)
- ANSI X9.23.

These methods also differ in how they define the initial chaining value (ICV).

This section describes how the Encipher and Decipher verbs implement these methods.

Single-DES and Triple-DES encryption algorithms for general data

Using the IBM 4758 Model 002 you can use the triple-DES algorithm in addition to the classical single-DES algorithm. In the subsequent descriptions of the CBC method and ANSI X9.23 method, the actions of the Encipher and Decipher verbs encompass both single-DES and triple-DES algorithms. The triple-DES processes are depicted in Figure 29 on page 414 where “left key” and “right key” refer to the two halves of a double-length DES key.

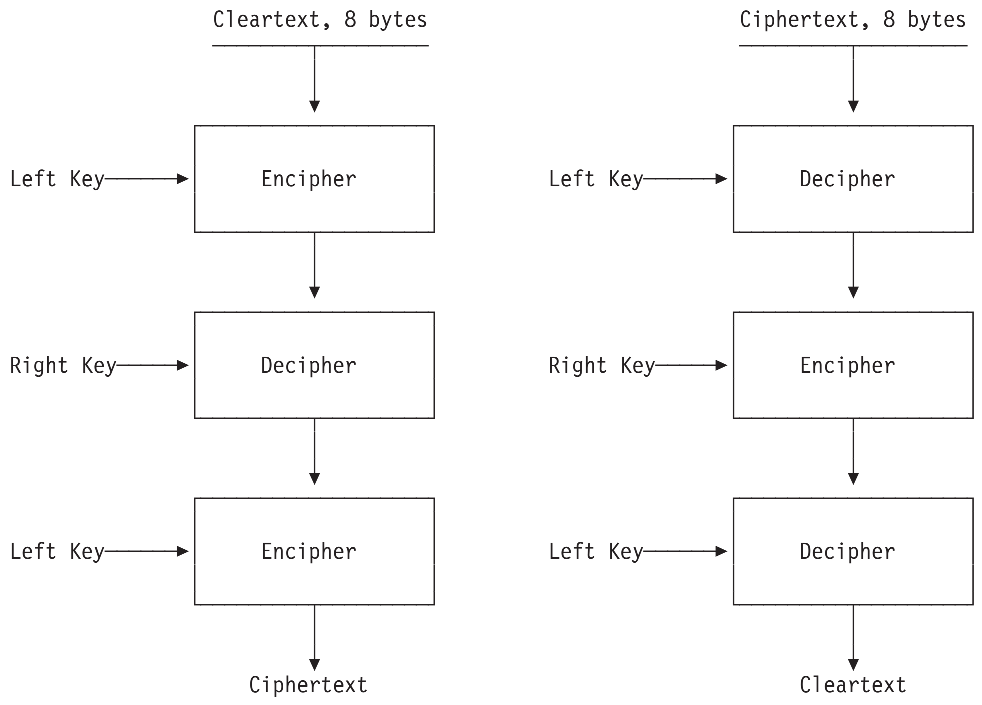


Figure 29. Triple-DES data encryption and decryption

ANSI X3.106 cipher block chaining method

ANSI standard X3.106 defines four modes of operation for ciphering. One of these modes, Cipher Block Chaining (CBC), defines the basic method for ciphering multiple 8-byte data strings. Figure 30 on page 415 and Figure 31 on page 415 show Cipher Block Chaining using the Encipher and the Decipher verbs. A plaintext data string that must be a multiple of 8 bytes, is processed as a series of 8-byte blocks. The ciphered result from processing an 8-byte block is exclusive-ORd with the next block of 8 input bytes. The last 8-byte ciphered result is defined as an output chaining value (OCV). The security server stores the OCV in bytes 0 through 7 of the *chaining_vector* variable.

An ICV is exclusive-ORd with the first block of 8 bytes. When you call the Encipher verb or the Decipher verb, specify the **INITIAL** or **CONTINUE** keywords. If you specify the **INITIAL** keyword, the default, the initialization vector from the verb parameter is exclusive-ORd with the first 8 bytes of data. If you specify the **CONTINUE** keyword, the OCV identified by the chaining_vector parameter is exclusive-ORd with the first 8 bytes of data.

ANSI X9.23

An enhancement to the basic CBC mode of X3.106 is defined so that the system can process data lengths that are not exact multiples of 8 bytes.

The ANSI X9.23 method always adds 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'. 1–7 bytes to the plaintext when the length of the plaintext is not a multiple of 8 bytes. In this method, the last added byte is the count of the added bytes and is within the range of X'01' to X'07'.

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.

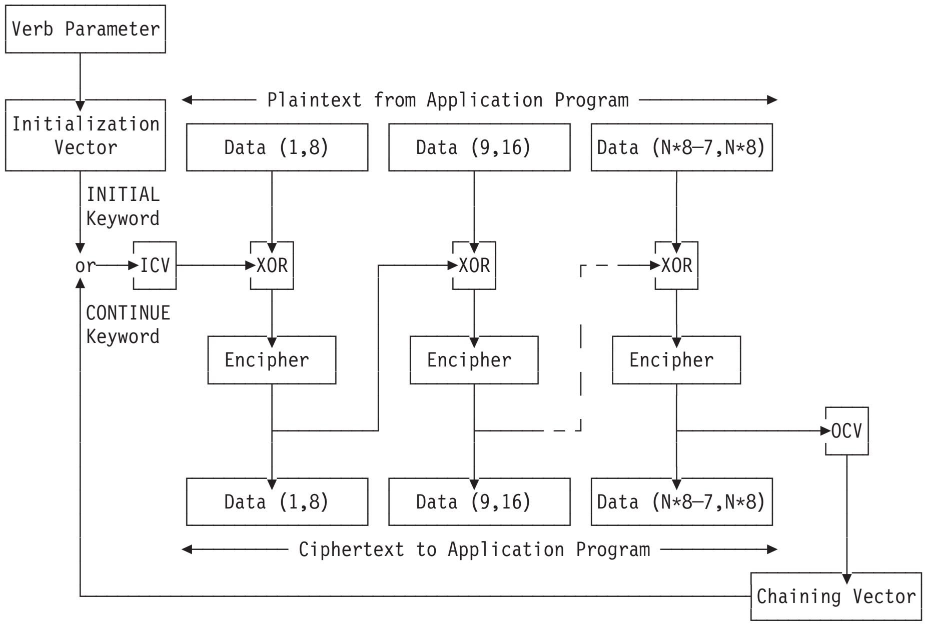


Figure 30. Enciphering using the ANSI x3.106 CBC method

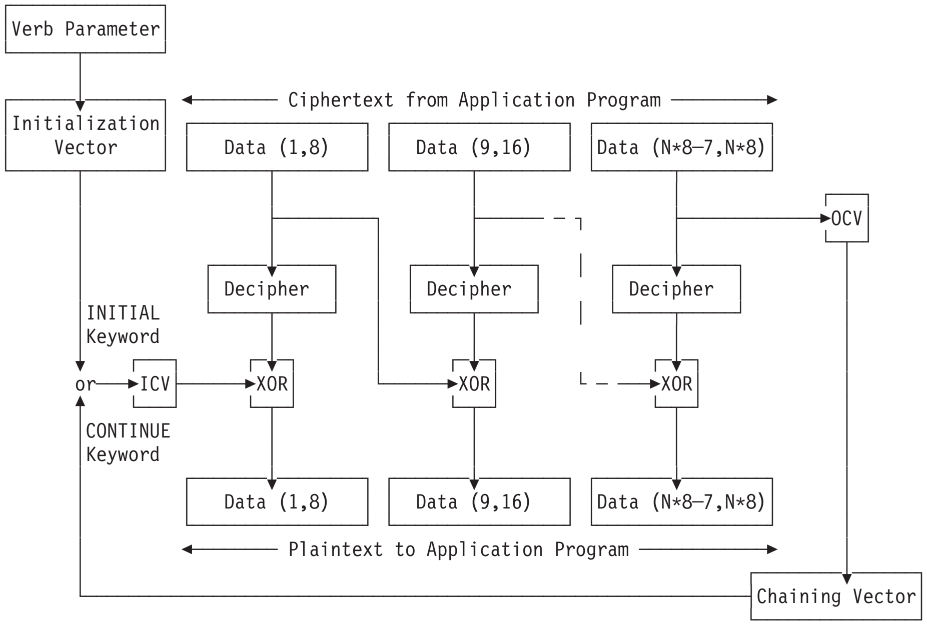


Figure 31. Deciphering using the CBC method

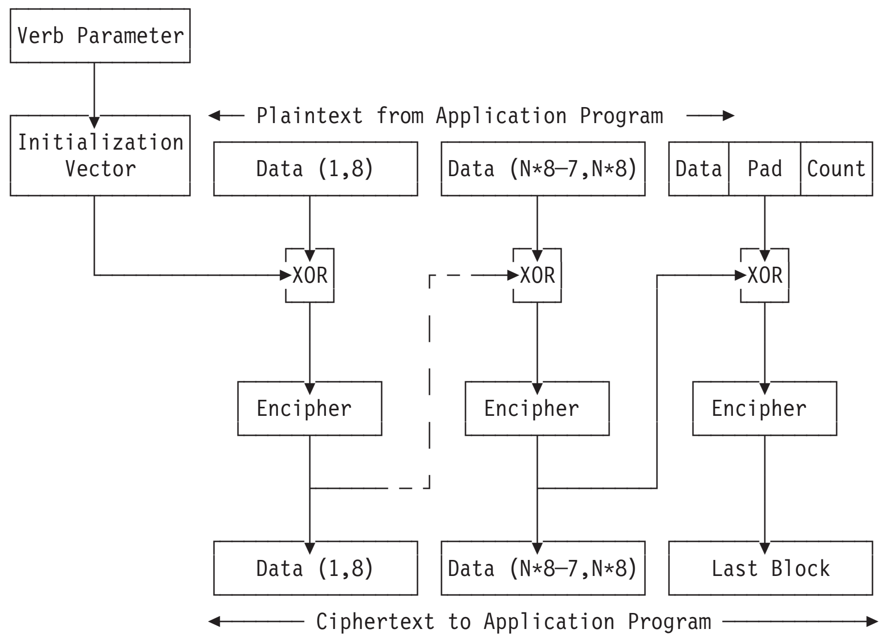


Figure 32. Enciphering using the ANSI X9.23 method

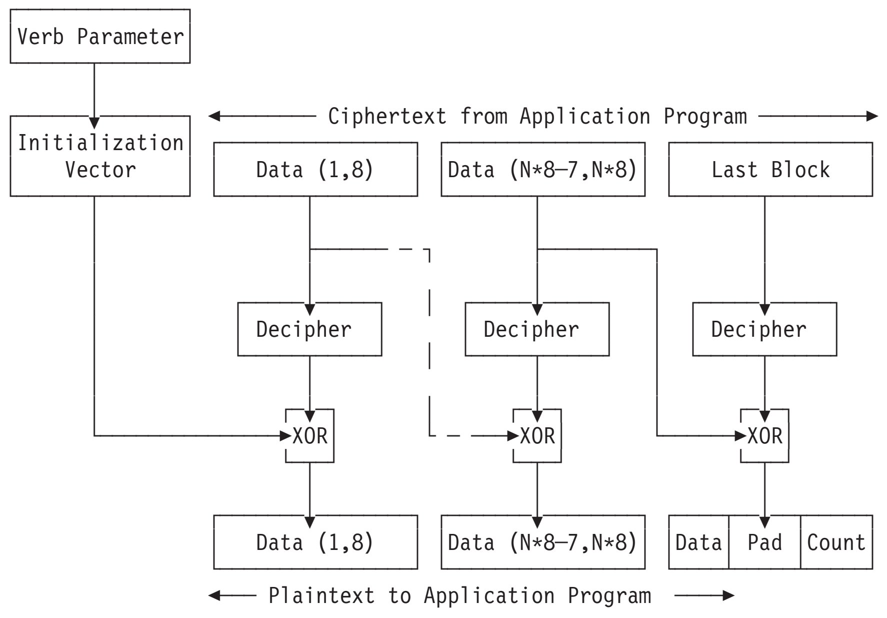


Figure 33. Deciphering using the ANSI X9.23 method

Triple-DES ciphering algorithms

A triple-DES algorithm is used to encrypt keys, PIN blocks, and general data. Several techniques are employed:

T-DES ECB
DES keys, when triple encrypted under a double-length DES key, are ciphered using an e-d-e scheme without feedback. See Figure 28 on page 404.

Triple-DES CBC

Encryption of general data, and RSA section type X'08' CRT-format private keys and OPK keys, employs the scheme depicted in Figure 34 and Figure 35 on page 418. This is often referred to as "outer CBC mode".

This CCA supports double-length DES keys for triple-DES data encryption using of the Decipher and Encipher verbs. The triple-length asymmetric master key is used to CBC encrypt CRT-format OPK keys. (See also Table 50 on page 357.)

EDE*x* / DEDx

CCA employs EDE*x* processes for encrypting several of the RSA private key formats (section types X'02', X'05', and X'06') and the OPK key in section type X'06'. The EDE*x* processes make successive use of single-key DES CBC processes. EDE2, EDE3, and EDE5 processes have been defined, based on the number of keys and initialization vectors used in the process. See Figure 36 on page 419 and Figure 37 on page 420. K1, K2, and K3 are true keys while "K4" and "K5" are initialization vectors. See Figure 36 on page 419 and Figure 37 on page 420.

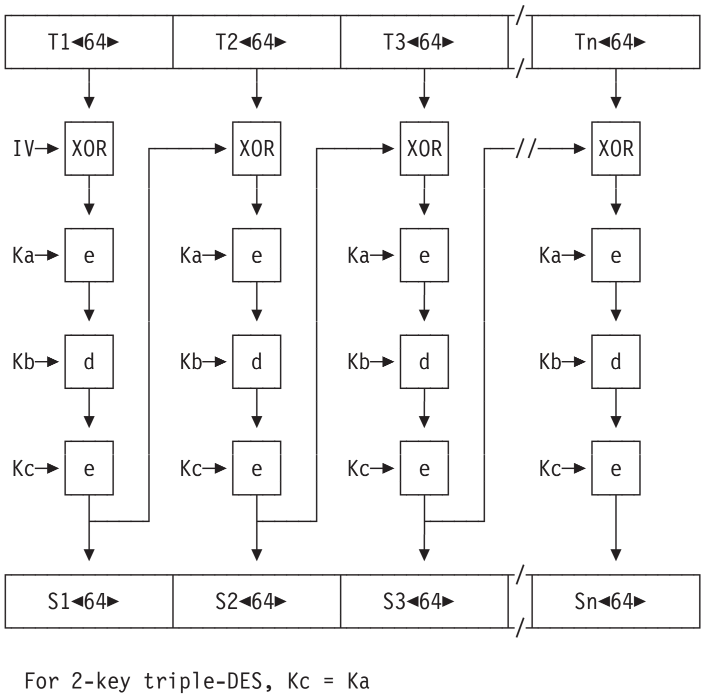


Figure 34. Triple-DES CBC encryption process

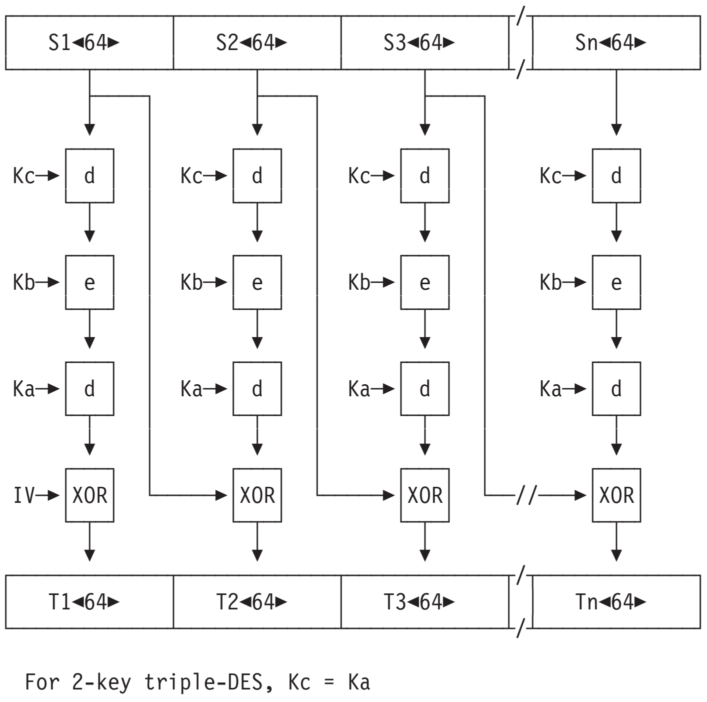


Figure 35. Triple-DES CBC decryption process

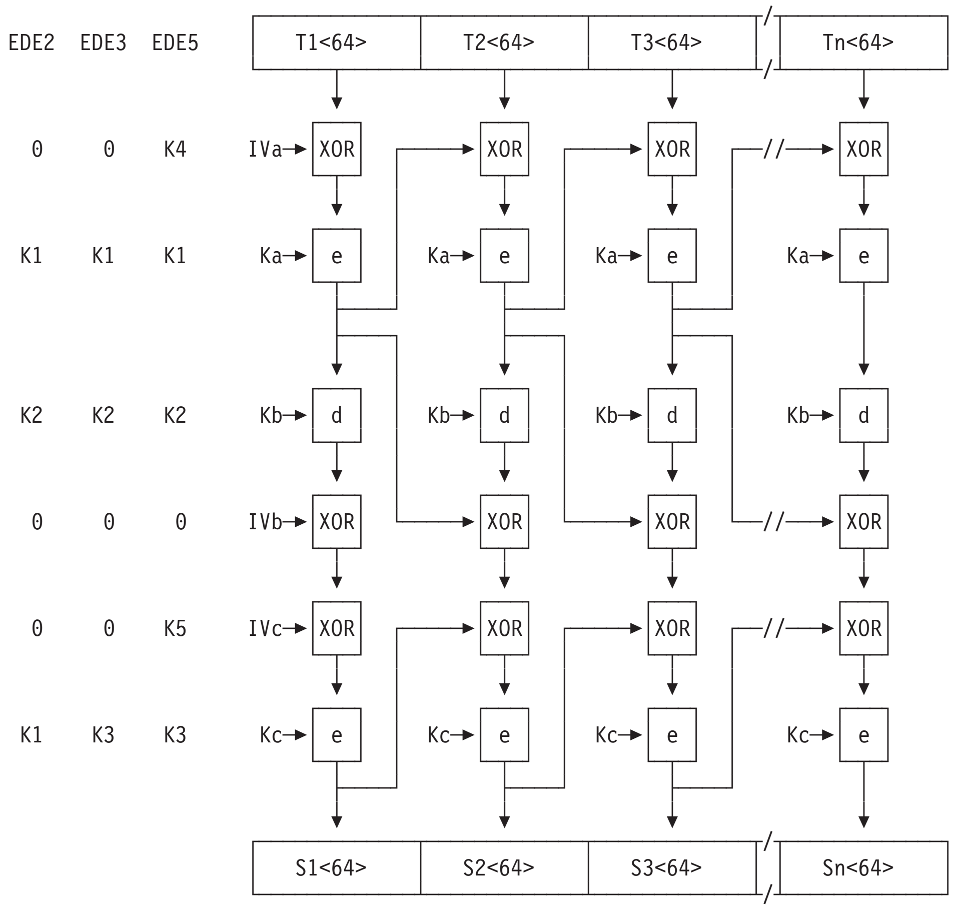


Figure 36. EDE algorithm

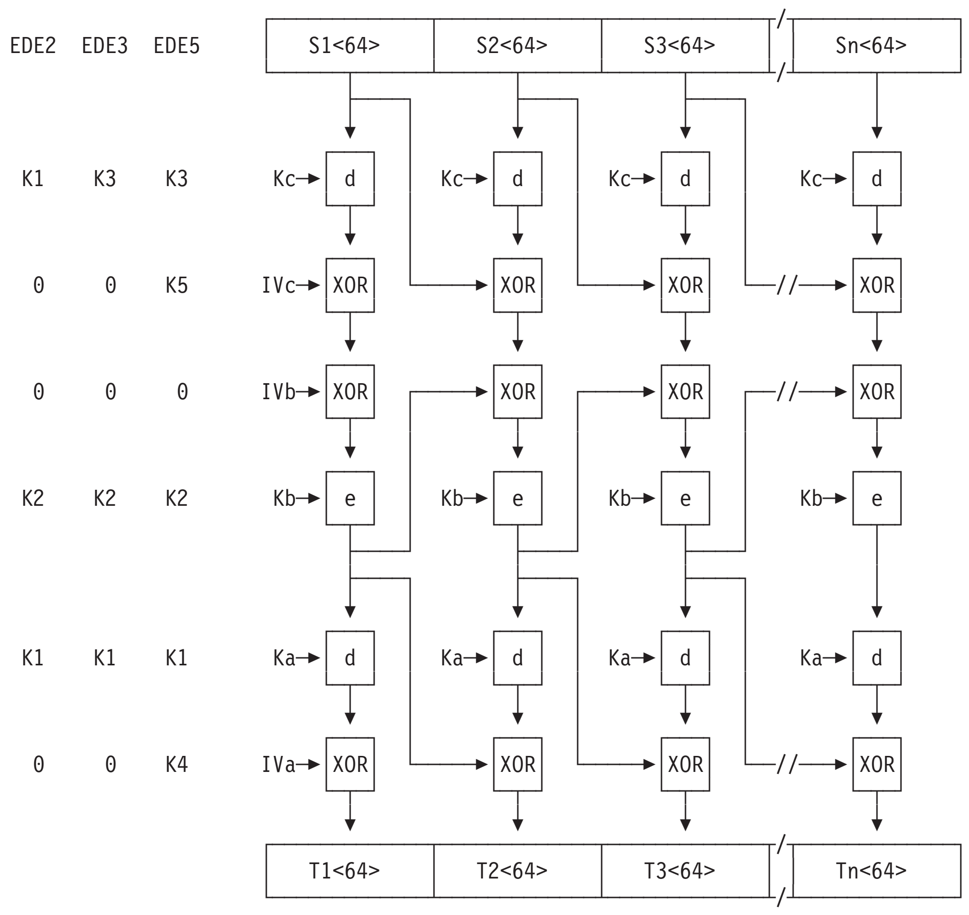


Figure 37. DED process

MAC calculation methods

Three variations of DES-based message authentication can be used by the MAC_Generate and MAC_Verify verbs:

- ANSI X9.9
- ANSI X9.19 optional Procedure 1
- EMV post-padding of X'80'

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.[7]

Figure 38 on page 421 shows the MAC calculation for binary data. In this figure, KEY is a 64-bit key, and T_1 through T_n are 64-bit data blocks of text. If T_n is less than 64 bits long, binary zeros are appended to the right of T_n. Data blocks T_1...T_n are DES CBC-encrypted with all output discarded except for the final output block, O_n.

7. The ANSI X9.9 standard defines five options. The MAC_Generate and MAC_Verify verbs implement option 1, binary data.

The Financial Institution (Retail) Message Authentication Standard, ANSI X9.19 Optional Procedure 1, specifies additional processing of the 64-bit O_n MAC value. The CCA "X9.19OPT" process employs a double-length DES key. After calculating the 64-bit MAC as above with the left half of the double-length key, the result is decrypted using the right half of the double-length key. This result is then encrypted with the left half of the double-length key. The resulting MAC value is processed according to other specifications supplied to the verb call.

The EMV smart card standards define MAC generation and verification processes that are the same as ANSI X9.9 and ANSI X9.19 Optional Procedure 1, except for padding added to the end of the message. Append 1 byte of X'80' to the original message. Then append additional bytes, as required, of X'00' to form an extended message, which is a multiple of 8 bytes in length.

In the X9.9 and X9.19 Optional Procedure 1 standards, the leftmost 32 bits (4 bytes) of O_n are taken as the MAC. In the EMV standards, the MAC value is 4-8 bytes in length. CCA provides support for the leftmost 4, 6 and 8 bytes of MAC value.

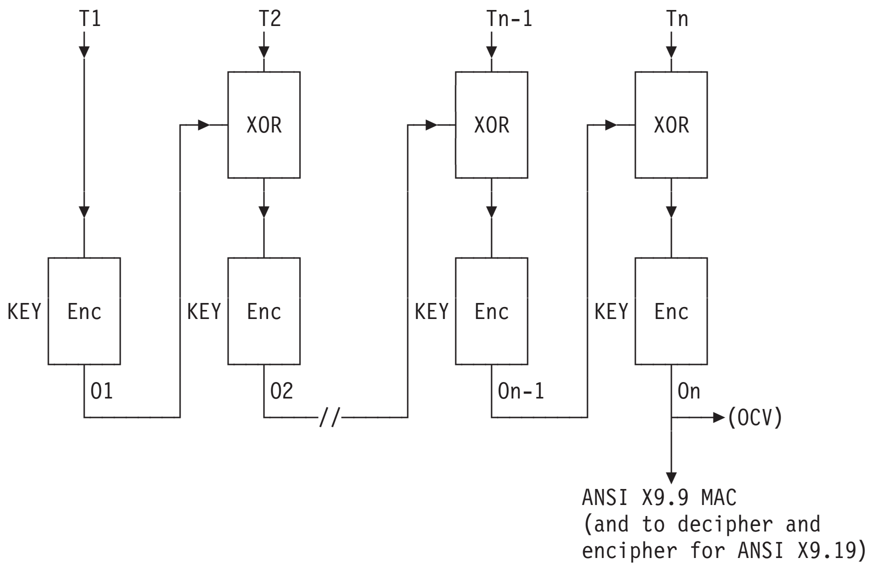


Figure 38. MAC calculation method

RSA key-pair generation

RSA key-pair generation is determined based on user input of the modulus bit length, public exponent, and key type. The output is based on creating primes p and q in conformance with ANSI X9.31 requirements as follows:

- prime *p* bit length = ((modulus_bit_length +1)/2)
- prime *q* bit length = modulus_bit_length - *p*_bit_length
- *p* and *q* are randomly chosen prime numbers
- $p > q$
- The Rabin-Miller Probabilistic Primality Test is iterated 8 times for each prime. This test determines that a false prime is produced with probability no greater then $1/4^c$, where *c* is the number of iterations. Refer to the ANSI x9.31 standard and see the section entitled "Miller-Rabin Probabilistic Primality Test".
- Primes *p* and *q* are relatively prime with the public exponent.

- Primes p and q are different in at least one of the first 100 most significant bits, that is, $|p\text{-}q| > 2^{(\text{prime bit length - 100})}$. For example, when the modulus bit length is 1024, then both primes bit length are 512 bits and the difference of the two primes is $|p\text{-}q| > 2^{412}$.

An RSA key is generated in the following manner with respect to random numbers:

1. For each key-generation, and for any size of key, the PKA Manager[8] seeds an internal FIPS-approved, SHA-1 based pseudo random number generator (PRNG) with the first 20 bytes (160 bits) of information that it receives from three successive calls to the RNG Manager's PRNG interface.
2. The RNG Manager can supply random numbers in three ways, but with the CCA Support Program only one way is used, the PRNG method. The PKA Manager seeds an internal FIPS-approved, SHA-1 based PRNG with the first 160 bits out of 192 bits it obtains from a *hardware random number pool*. The PRNG responds with eight random bytes (64 bits) per request. After every eight requests, the PRNG is reseeded from the hardware random number pool. The RNG Manager can respond to requests for random numbers from other processes with such responses interspersed between responses to PKA Manager requests.

   The RNG Manager collects a stream of random bits from a hardware random-bit source into a 20,000 bit pool. The manager then turns off the hardware random-bit generator until additional bits are needed. The goal is to always have 20,000 bits in the pool. Bits are supplied first-in, first-out from the pool.
3. An RSA key is generated from random information obtained from two cascaded SHA-1 PRNGs. An RSA key is based on one or more 160-bit seeds from the hardware random-bit source depending on the dynamic mix of tasks running within the coprocessor.

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 coprocessor.

Design criteria

The passphrase verification protocol is designed to meet the following criteria.

- 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.
- 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.
- An attacker should not be able to guess the cleartext content of the logon request message.
- No special hardware should be required on the client workstation.
- The logon process must result in the establishment of a session key known only to the Cryptographic Coprocessor and the client. This key is used on subsequent transactions to prove the identity of the sender, and to secure transmitted data.

8. The "PKA Manager" (public-key architecture) and the "RNG Manager" (random number) are components of the control program which support the CCA application within the coprocessor.

- The session key is generated in the coprocessor.

Description of the protocol

The protocol is comprised of the following steps:

1. The user provides the user ID (UID) and passphrase.
2. The passphrase is hashed in the client workstation, using SHA-1 algorithm. The resulting hash value 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 8 bytes of the hash, $K2_L$ is comprised of the second 8 bytes, and $K3_L$ is comprised of the last 4 bytes, concatenated with 4 bytes of X'00'. Figure 39 shows an example to clarify this.

```
Passphrase is "This is my passphrase!"

   SHA-1 hash of the passphrase is hex 42BED1CD 1DB68934 6319E315 F3C096A8 B2E08DB2
                                       └───────┬───────┘ └───────┬───────┘ └──┬───┘
                                               │                 │            │
   K1 is 42BED1CD 1DB68934 ◄───────────────────┘                 │            │
   K2 is 6319E315 F3C096A8 ◄─────────────────────────────────────┘            │
   K3 is B2E08DB2 00000000 ◄──────────────────────────────────────────────────┘
```

Figure 39. 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[9] mode, performed by the 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 1 byte except for the year which is held in a 2-byte value.
5. The Cryptographic Coprocessor retrieves the user profile, which it has in secure coprocessor memory. It uses the received user ID value to locate the corresponding 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, obtaining K_L.
7. The coprocessor uses K_L to decrypt the user's logon data, recovering the UID, timestamp, and RN. It compares the recovered UID with the cleartext UID it received, and abnormally ends 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.

9. For a description of the EDE3 encryption process, see Figure 36 on page 419.

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. The coprocessor generates a 192-bit session key, K_S, and returns this key to the client in the form of $eK_L(K_S)$.
10. In a secure internal table, the coprocessor stores the user ID, the value of K_S, and the user's role identifier, which is extracted from the profile. This information 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 the system.
11. The client workstation software (SAPI) saves K_S for use in validating subsequent verb calls. The SAPI code in the client and the coprocessor compute the industry-standard HMAC keyed-hash algorithm over sensitive portions of subsequent verb calls and responses. An HMAC is computed using K_S as the key.

Master-key-splitting algorithm

This section describes the mathematical and cryptographic basis for the *m*-of-*n* key shares scheme.

The key splitting is based on Shamir's secret sharing algorithm:

The value to be shared is the master key, K*m*, which is a triple-DES key and 168 bits long. Let *P* be the first prime number larger than 2^{168}. All operations are carried out modulo P.

Shamir's secret sharing allows the sharing of K*m* among *n* trustees in a way that no set of *t* or less of trustees has any information about K*m*, while *t*+1 trustees can reconstruct K*m*.

Sharing phase:

1. Randomly choose *a_t*,...,*a*_1 in [0..P-1]
2. Consider the polynomial *f*(*x*) = *a_t x t* + ... + *a*_1 *x* + *a*_0, where *a*_0=K*m*.
   Compute *mk_i* = *f*(i) mod P for all *i*=1,...*n*
3. Proceed to distribute the values *mk_i* as described above.

Reconstruction phase:

After generating the set of authentic values proceed as follows:

1. Take *t*+1 values and interpolate the polynomial f(x) of degree *t*. Pass through these values using Lagrange interpolation. This defines a polynomial f(x) such that: f(i)=mk_i, and further more f(0) = MK. Use the mathematical formula to reconstruct the free term of the polynomial f(x). Let k_1,...,k_{t+1} be the indices of the mk_i's used for reconstruction. Then

   a_0=SUM_j(b_{k_j} PROD_h (x_{k_h} / (x_{k_h}- x_{k_j}))) mod P
2. Install key Km = a_0 = f(0) mod P.

Formatting hashes and keys in public-key cryptography

The Digital_Signature_Generate and Digital_Signature_Verify verbs can use several methods for formatting a hash value, and in some cases a descriptor for the hashing method, into a bit-string to be processed by the cryptographic algorithm. This section discusses the ANSI X9.31 and PKCS #1 methods. The ISO 9796-1 method can be found in the ISO standard.

This section also describes the PKCS #1 methods for placing a key in a bit string for RSA ciphering as part of a key exchange.

ANSI X9.31 Hash Format

With ANSI X9.31, the string that is processed by the RSA algorithm is formatted by the concatenation of a header, padding, the hash value and a trailer, from the most significant bit to the least significant bit, so that the resulting string is the same length as the modulus of the key. For CCA, the modulus length must be a multiple of 8 bits.

- The header consists of the value X'6B'
- The padding consists of the value X'BB', repeated as many times as required, and ended with X'BA'
- The hash value follows the padding
- The trailer consists of a hashing mechanism specifier and final byte. These specifiers are defined as the following values:
  - X'31': RIPEMD-160
  - X'32': RIPEMD-128
  - X'33': SHA-1
- A final byte of X'CC'

PKCS #1 Hash Formats

Version 2.0 of the PKCS #1 standard[10] defines methods for formatting keys and hash values prior to RSA encryption of the resulting data structures. Earlier versions of the PKCS #1 standard defined block types 0, 1, and 2, but in the current standard that terminology is dropped.

The CCA products described in this document implement these processes using the terminology of the Version 2.0 standard:

- For formatting keys for secured transport:
  - RSAES-OAEP is the preferred method for key encipherment[11] when exchanging DATA keys between systems. In CCA, keyword **PKCSOAEP** is used to invoke this formatting technique. The “P” parameter described in the standard is not used and its length is set to zero.
  - RSAES-PKCS1-v1_5, is an older method for formatting keys. In CCA, keyword **PKCS-1.2** is used to invoke this formatting technique.
- For formatting hash values for digital signatures:
  - RSASSA-PKCS1-v1_5 is the name for the block-type 1 format. In CCA, keyword **PKCS-1.1** is used to invoke this formatting technique.

10. PKCS standards can be retrieved from http://www.rsasecurity.com/rsalabs/pkcs.

11. The PKA92 method and the method incorporated into the SET standard are other examples of the OAEP technique. The “OAEP” technique is attributed to Bellare and Rogaway and stands for “Optimal Asymmetric Encryption Padding”.

Using the terminology from older versions of the PKCS #1 standard, block types 0 and 1 are used to format a hash and block type 2 is used to format a DES key. The blocks consist of the following ("||" means concatenation):

X'00' || BT || PS || X'00' D

where:

– BT is the block type, X'00', X'01', or X'02'.
– PS is the padding of as many bytes as required to make the block the same length as the modulus of the RSA key, and is bytes of X'00' for block type 0, X'FF' for block type 1, and random and non-X'00' for block type 2. The length of PS must be at least 8 bytes.
– D is the key, or the concatenation of the BER-encoded hash identifier and the hash value.

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

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

Appendix E. Financial system verbs calculation methods and data formats

This section describes the following:

- PIN-calculation methods
- PIN-block formats
- Unique-key-per-transaction calculation methods
- MasterCard and Visa card verification techniques
- Visa and EMV smart card PIN-related formats and processes.

The PIN calculation methods are independent from PIN-block formats. A PIN can be calculated by any method and generally used in any PIN-block 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.

Table 79. Financial PIN calculation methods, data formats, and other items

Item	Page
IBM 3624 PIN-calculation method	428
IBM 3624 PIN offset calculation method	428
Netherlands PIN-1 calculation method	429
IBM German Bank Pool Institution PIN-calculation method	429
Visa PIN validation value (PVV) calculation method	430
Interbank PIN-calculation method	430
3624 PIN-block format	431
ISO-0 PIN-block format	431
ISO-1 PIN-block format	432
ISO-2 PIN-block format	433
UKPT calculation methods (ANSI X9.24)	434
CVV, CVC (Visa, MasterCard)	436
Visa and EMV formats and processes	437

PIN-calculation methods

The financial PIN verbs support some or all of these PIN-calculation methods, see Table 24 on page 269:

- 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.

IBM 3624 PIN-calculation method

The IBM 3624 PIN-calculation method calculates a PIN that is from 4 – 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 – 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-by-digit, 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 – 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-by-digit, 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 institution key 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 calculation method

The Visa PIN validation value (PVV) calculation method calculation method calculates a VISA-PVV that is 4 digits in length.

The 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 (a portion of the account number) with the 4-numeric-digit customer-entered PIN. (C-PIN when calculating the PVV, or T-PIN when validating a transaction.)
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 (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 40. 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. 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 41. 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 • The leftmost 12 digits of the primary account-number (excluding the check digit) if the format of the PIN block is VISA-4. Each PAN digit has a value from X'0' 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 42. 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 43. 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'.
```

Unique-key-per-transaction calculation methods

This section describes the calculation methods for deriving the unique-key-per-transaction (UKPT) key according to ANSI X9.24 and performing the special encryption and special decryption processes.[12]

Deriving an ANSI X9.24 unique-key-per-transaction key

To determine the current-transaction encrypting key used by a terminal which is encrypting PIN-blocks under the ANSI X9.24 standard, the ANSI X9.24 algorithm uses a derivation key and the Current® Key Serial Number (CKSN) as inputs.

- The derivation key must be a double-length KEYGENKY key-type with the UKPT control vector bit set on. The right half of the derivation key cannot be the same as the left half of the derivation key.
- The *initial key serial number* is a 59-bit value that contains terminal identification information that is unique among the set of terminals initialized under a given derivation key.
- The *encryption counter* is a 21-bit counter value. The value in the counter is set to 0 when the terminal is initialized. The counter increments each time the terminal performs a PIN-block encryption. The counter increments such that a maximum of 10 bits can be set on; the counter can record 1 000 000 encryptions.
- The *current key serial number* (CKSN) is the concatenation of the initial key serial number and the encryption counter. This concatenation is an 80-bit (10-byte) value.

The calculation method consists of the following steps:

1. Calculate the initial encrypting key. To calculate the initial encrypting key, do the following:
   a. Move the leftmost 8 bytes of the current key serial number to a work area (C_a).
   b. Perform an AND operation with the last byte of C_a and X'EO'. This operation clears the high-order bits of the encryption counter. The value that C_a now contains is the initial serial number that was loaded when the PIN keypad was initialized.
   c. Encrypt C_a, using the left half of the derivation key; name the result C_b.
   d. Decrypt C_b, using the right half of the derivation key; name the result C_c.
   e. Encrypt C_c, using the left half of the derivation key; name the result C_d. C_d is the initial PIN encrypting key that was loaded when the terminal was initialized.
   f. Rename C_d to be K_a, the initial PIN encrypting key.
2. Calculate the current encrypting key. To calculate the current encrypting key, do the following:
   a. Move the rightmost 8 bytes of the current key serial number to a work area (W_a).
   b. Move the rightmost 3 bytes of W_a to another work area (C_a).
   c. Perform an AND operation with the rightmost 3 bytes of W_a with X'E00000'. This operation clears the encryption counter from W_a.
   d. Perform an AND operation with C_a and X'1FFFFF'. This operation clears the low-order bits of the initial serial number from the encryption counter.

12. This material is adapted from the *Visa Point-of-Sale Equipment Requirements: PIN Processing and Data Authentication* publication.

e. Initialize a 3-byte area to X'100000'; name the result S_a.

f. Initialize a 1-byte counter to X'00'; name the result B_a.

g. Test each bit of the encryption counter, looking for B'1' bits by doing the following loop:

   - When a B'1' bit is found, it ORs this bit into the initial serial number. It then special encrypts the result with K_a.
   - The result of this special encryption is the new K_a.
   - When all B'1' bits are processed, a *variant* of the value in K_a becomes the current encrypting key.

   Use the following procedure to do this loop:

```
DO i=1 to 21
 a. IF ($C_a$ AND $S_a$) is not equal to 0 THEN DO
    1) ADD 1 to $B_a$
    2) IF $B_a$ > 10 THEN exit algorithm with an error
       indicating too many B'1' bits were set in the encryption
       counter
    3) OR $S_a$ into the rightmost 3 bytes of $W_a$;
       store the result in $T_a$
    4) XOR $T_a$ and $K_a$; store the result in $T_b$
    5) Encrypt $T_b$ with $K_a$; store the result in $T_c$
    6) XOR $T_c$ with $K_a$; store the result in $K_a$
 b. END IF
 c. Shift $S_a$ one bit to the right.
    Fill in on the left with a B'0' bit.
 END DO
```

   The value in K_a is the current encrypting key.

   Note: The CCA implementation does not adjust key parity on any of the bytes of the derived encrypting key before encrypting them under its master key. Parity adjustment is not done because the key value is used in two XOR operations during the *special decrypt* process of recovering the clear PIN-block.

The following is an example of calculating the initial PIN encrypting key:

```
Derivation key = X'5152 5457 585B 5D5E 6162 6467 686B 6D6E'
Current key serial number = X'0123 4567 89AB CDF0 0001'

$C_a$ = X'0123 4567 89AB CDE0'
$C_b$ = X'6497 E2F4 C59D 952E'
$C_c$ = X'0163 CE85 359F F599'

Initial PIN encrypting key = $K_{a_1}$ = $C_d$ = X'21EE 7C08 DBE8 20AB'
```

The following is an example of calculating the current PIN encrypting key:

```
$W_a$ = X'4567 89AB CDE0 0000'
$C_a$ = X'10001'
$S_{a_1}$ = X'100000'


$T_{a_1}$ = X'4567 89AB CDF0 0000'
$T_{b_1}$ = X'6489 F5A3 1618 20AB'
$T_{c_1}$ = X'F9AC C638 1939 44BC'
$K_{a_2}$ = X'D842 BA30 C2D1 6417'
   ⋮
$S_{a_{20}}$ = X'000001'
$T_{a_{20}}$ = X'4567 89AB CDF0 0001'
$T_{b_{20}}$ = X'9D25 339B 0F21 6416'
$T_{c_{20}}$ = X'BF49 836E AE2A 042A'
```

$K_{a_{20}}$ = X'670B 395E 6CFB 603D'

```
Current PIN encrypting key = X'670B 395E 6CFB 60C2'
```

Performing the special encryption and special decryption processes

The special encryption process consists of the following steps:

1. Name the derived unique key for the current transaction K_u.
2. Name the clear PIN-block that was built from the user-entered PIN P_c.
3. Perform an XOR operation with the rightmost byte of K_u and X'FF' to produce a variant of the key; name the result K_{u_v}.
4. Perform an XOR operation with K_{u_v} and P_c; store the result in T_1.
5. Encrypt T_1 with K_{u_v}; store the result in T_2.
6. Perform an XOR operation with K_{u_v}; store the result in P_e.

The value in P_e is the encrypted PIN-block that the POS terminal sends.

The special decryption process consists of the previous steps, but in reverse.

The following is an example of the special encryption process:

```
Current encrypting key = Ku = X'670B 395E 6CFB 603D'
User-entered PIN = 1234
User's primary account-number = X'4012 3456 7890'
Clear PIN-block (unformatted) = X'0412 34FF FFFF FFFF'
Primary account-number (formatted) = X'0000 4012 3456 7890'
Clear PIN-block (ANSI format) = Pc = X'0412 74ED CBA9 876F'
Variant of PIN encrypting key = Kuv = X'670B 395E 6CFB 60C2'

T1 = X'6319 4DB3 A752 E7AD'
T2 = X'5145 3CA3 E474 2148'
Pe = X'364E 05FD 888F 418A'
```

CVV and CVC Method

Figure 44 on page 437[13] shows the method used to generate a card-verification value (CVV) for track 2. Each (decimal) digit is represented as a 4-bit, binary value and packed two digits per byte.

13. Adapted from *VisaNet Electronic Value Exchange Standards Manual*, pages AA-8 and AA-9.

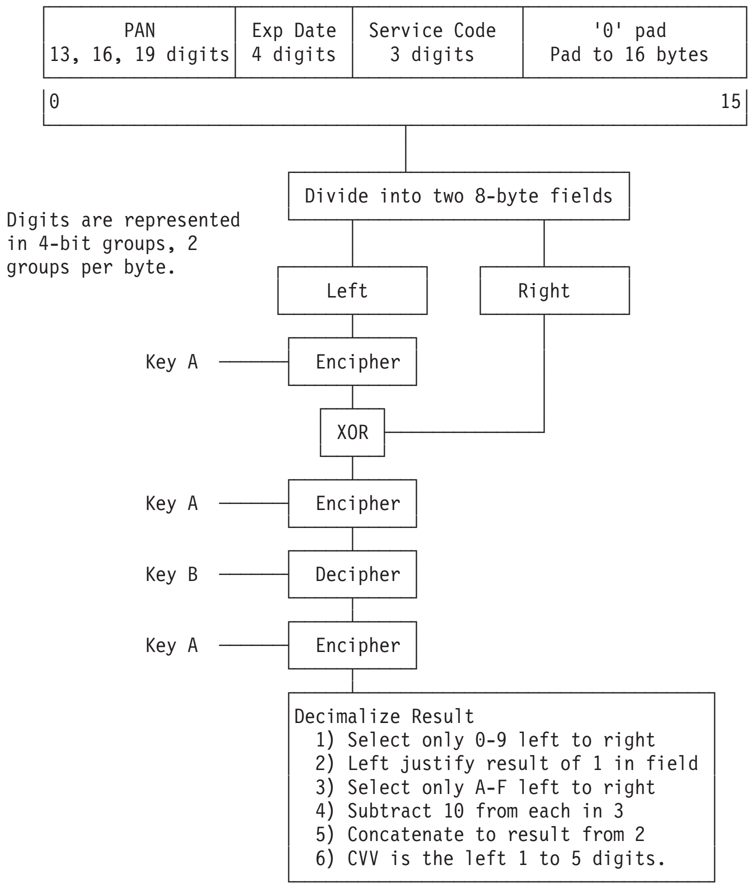

PAN
13, 16, 19 digits
Exp Date
4 digits
Service Code
3 digits
'0' pad
Pad to 16 bytes
0
15
Divide into two 8-byte fields
Digits are represented in 4-bit groups, 2 groups per byte.
Left
Right
Key A
Encipher
XOR
Key A
Encipher
Key B
Decipher
Key A
Encipher
Decimalize Result
1) Select only 0-9 left to right
2) Left justify result of 1 in field
3) Select only A-F left to right
4) Subtract 10 from each in 3
5) Concatenate to result from 2
6) CVV is the left 1 to 5 digits.

Figure 44. CVV track 2 algorithm

At the security application programming interface, the CVV_Generate and CVV_Verify verbs require two key identifiers, key-A and key-B, as defined in the CVV method. The identifiers can be key labels or internal key-tokens.

The key-A and key-B key pair can include the following key types:

1. Both keys can be DATA keys.
2. Both keys can be MAC-class keys with the ANY subtype extension.
3. Both keys can be MAC-class keys with the KEY-A and KEY-B subtype extensions as appropriate.

The CVV_Generate verb requires the control-vector bit 20 to be set to 1. The CVV_Verify verb requires the control-vector bit 21 to be set to 1.

Visa and EMV-related smart card formats and processes

The VISA and EMV specifications for performing secure messaging with an EMV compliant smart card are covered in these documents:

- *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2*
- *Design Visa Integrated Circuit Card Specification Manual*

Book 2, Annex A1.3, describes how a smart-card, card-specific authentication code is derived from a card-issuer-supplied authentication key (MAC-MDK).

Annex A1.3 describes how a smart-card, card-specific session key is derived from a card-issuer-supplied PIN-block-encryption key (ENC-MDK). The encryption key is derived using a "tree-based-derivation" technique. IBM CCA offers two variations of the tree-based technique (**TDESEMV2** and **TDESEMV4**), and a third technique CCA designates **TDES-XOR**.

In addition, Book 2 describes construction of the PIN block sent to an EMV card to initialize or update the user's PIN.

Design Visa Integrated Circuit Card Specification Manual, Annex B.4, contains a description of the session-key derivation technique CCA designates **TDES-XOR**.

Augmented by the above-mentioned documentation, the relevant processes are described in these sections:
- Derivation of the smart-card-specific authentication code
- Constructing the PIN-block for transporting an EMV smart-card PIN
- Derivation of the CCA TDES-XOR session key
- Derivation of the EMV TDESEMVn tree-based session-key
- PIN-block self-encryption

Deriving the smart-card-specific authentication code

To ensure that an original or replacement PIN is received from an authorized source, the EMV PIN-transport PIN-block incorporates an authentication code.

The authentication code is the rightmost four bytes resulting from the ECB-mode triple-DES encryption of (the first) eight bytes of card specific data.

Constructing the PIN-block for transporting an EMV smart-card PIN

The PIN block is used to transport a new PIN value. The PIN block also contains an authentication code, and optionally the "current" PIN value, enabling the smart card to further ensure receipt of a valid PIN value. To enable incorporation of the PIN block into the a message for an EMV smart-card, the PIN block is padded to 16 bytes prior to encryption.

PINs of length 4 to 12 digits are supported.

PIN block construction:
1. Form three 8-byte, 16-digit blocks, -1, -2, and -3, and set all digits to X'0'.
2. Replace the rightmost four bytes of block-1 with the authentication code described in the previous section.
3. Set the second digit of block-2 to the length of the new PIN (4 to 12), followed by the new PIN, and padded to the right with X'F'.
4. Include any current PIN by placing it into the leftmost digits of block-3.
5. Exclusive-OR blocks -1, -2, and -3 to form the 8-byte PIN block.
6. Pad the PIN block with other portions of the message for the smart card:
   - Prepend X'80'
   - Append X'80'
   - Append and additional six bytes of X'00'.

The resulting message is ECB-mode triple-encrypted with an appropriate session key.

Deriving the CCA TDES-XOR session key

In the Diversified_Key_Generate and PIN_Change/Unblock verbs, the **TDES-XOR** process first derives a smart-card-specific intermediate key from the issuer-supplied ENC-MDK key and card-specific data. (This intermediate key is also used in the **TDESEMV2** and **TDESEMV4** processes. See the next section.) The intermediate key is then modified using the application transaction counter (ATC) value supplied by the smart card.

The double-length session-key creation steps:

1. Obtain the left-half of an intermediate key by ECB-mode triple-DES encrypting the (first) eight bytes of card specific data using the issuer-supplied ENC-MDK key.
2. Again using the ENC-MDK key, obtain the right-half of the intermediate key by ECB-mode triple-DES encrypting:
   - The second 8 bytes of card-specific derivation data when 16 bytes have been supplied, else
   - The exclusive-OR of the supplied 8 bytes of derivation data with X'FFFFFFFF FFFFFFFF'.
3. Pad the ATC value to the left with six bytes of X'00' and exclusive-OR the result with the left-half of the intermediate key to obtain the left-half of the session key.
4. Obtain the one's complement of the ATC by exclusive-ORing the ATC with X'FFFF'. Pad the result on the left with six bytes of X'00'. Exclusive-OR the 8-byte result with the right-half of the intermediate key to obtain the right-half of the session key.

Deriving of the EMV TDESEMV*n* tree-based session key

In the Diversified_Key_Generate and PIN_Change/Unblock verbs, the **TDESEMV2** and **TDESEMV4** keywords call for the creation of the session key with this process:

1. The intermediate key is obtained as explained above for the **TDES-XOR** process.
2. Combine the intermediate key with the two-byte Application Transaction Counter (ATC) and an optional Initial Value. The process is defined in the *EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (EMV4.0) Book 2* Book 2, Annex A1.3.
   - **TDESEMV2** causes processing with a branch factor of 2 and a height of 16.
   - **TDESEMV4** causes processing with a branch factor of 4 and a height of 8.

PIN-block self-encryption

In the Secure_Messaging_for_PINs (CSNBSPN) verb, you can use the **SELFENC** rule-array keyword to specify that the 8-byte PIN block shall be used as a DES key to encrypt the PIN block. The verb appends the self-encrypted PIN block to the clear PIN-block in the output message.

Appendix F. Verb list

This section lists the verbs supported by the CCA Support Program.

Table 80 lists each verb by the verb's pseudonym and entry-point name and lists the page for the verb. Be sure to look at the top of the first page of a verb description to determine which platform and product supports the verb.

Table 80. Security API verbs in supported environments

Pseudonym	**Entry point**	**Page**
DES key-processing and key-storage verbs		
Clear_Key_Import	CSNBCKI	145
Control_Vector_Generate	CSNBCVG	146
Control_Vector_Translate	CSNBCVT	148
Cryptographic_Variable_Encipher	CSNBCVE	151
Data_Key_Export	CSNBDKX	153
Data_Key_Import	CSNBDKM	155
DES_Key_Record_Create	CSNBKRC	244
DES_Key_Record_Delete	CSNBKRD	245
DES_Key_Record_List	CSNBKRL	247
DES_Key_Record_Read	CSNBKRR	249
DES_Key_Record_Write	CSNBKRW	250
Diversified_Key_Generate	CSNBDKG	157
Key_Encryption_Translate	CSNBKET	164
Key_Export	CSNBKEX	167
Key_Generate	CSNBKGN	169
Key_Import	CSNBKIM	176
Key_Part_Import	CSNBKPI	179
Key_Storage_Initialization	CSNBKSI	65
Key_Test	CSNBKYT	183
Key_Test_Extended	CSNBKYTX	187
Key_Token_Build	CSNBKTB	191
Key_Token_Change	CSNBKTC	194
Key_Token_Parse	CSNBKTP	196
Key_Translate	CSNBKTR	200
Multiple_Clear_Key_Import	CSNBCKM	202
PKA_Decrypt	CSNDPKD	204
PKA_Encrypt	CSNDPKE	206
PKA_Symmetric_Key_Export	CSNDSYX	209
PKA_Symmetric_Key_Generate	CSNDSYG	212
PKA_Symmetric_Key_Import	CSNDSYI	216
Prohibit_Export	CSNBPEX	220
Prohibit_Export_Extended	CSNBPEXX	221

Table 80. Security API verbs in supported environments (continued)

Pseudonym	Entry point	Page
Random_Number_Generate	CSNBRNG	222
Data confidentiality and data integrity verbs		
Decipher	CSNBDEC	229
Digital_Signature_Generate	CSNDDSG	110
Digital_Signature_Verify	CSNDDSV	114
Encipher	CSNBENC	232
MAC_Generate	CSNBMGN	235
MAC_Verify	CSNBMVR	238
MDC_Generate	CSNBMDG	117
One_Way_Hash	CSNBOWH	120
Random_Number_Tests	CSUARNT	79
Coprocessor control verbs		
Access_Control_Initialization	CSUAACI	35
Access_Control_Maintenance	CSUAACM	38
Cryptographic_Facility_Control	CSUACFC	44
Cryptographic_Facility_Query	CSUACFQ	48
Cryptographic_Resource_Allocate	CSUACRA	59
Cryptographic_Resource_Deallocate	CSUACRD	61
Key_Storage_Designate	CSUAKSD	63
Logon_Control	CSUALCT	67
Master_Key_Distribution	CSUAMKD	70
Master_Key_Process	CSNBMKP	74
RSA key-administration and key-storage verbs		
Key_Storage_Initialization	CSNBKSI	65
PKA_Key_Generate	CSNDPKG	87
PKA_Key_Import	CSNDPKI	91
PKA_Key_Token_Build	CSNDPKB	93
PKA_Key_Token_Change	CSNDKTC	99
PKA_Key_Record_Create	CSNDKRC	251
PKA_Key_Record_Delete	CSNDKRD	253
PKA_Key_Record_List	CSNDKRL	255
PKA_Key_Record_Read	CSNDKRR	257
PKA_Key_Record_Write	CSNDKRW	259
PKA_Public_Key_Extract	CSNDPKX	101
PKA_Public_Key_Hash_Register	CSNDPKH	103
PKA_Public_Key_Register	CSNDPKR	105
Retained_Key_Delete	CSNDRKD	261
Retained_Key_List	CSNDRKL	262
Financial services support verbs		
Clear_PIN_Encrypt	CSNBCPE	278

Table 80. Security API verbs in supported environments *(continued)*

Pseudonym	Entry point	Page
Clear_PIN_Generate	CSNBPGN	281
Clear_PIN_Generate_Alternate	CSNBCPA	284
CVV_Generate	CSNBCSG	289
CVV_Verify	CSNBCSV	292
Encrypted_PIN_Generate	CSNBEPG	295
Encrypted_PIN_Translate	CSNBPTR	299
Encrypted_PIN_Verify	CSNBPVR	304
PIN_Change/Unblock	CSNBPCU	310
Secure_Messaging_for_Keys	CSNBSKY	317
Secure_Messaging_for_PINs	CSNBSPN	320
SET_Block_Compose	CSNDSBC	324
SET_Block_Decompose	CSNDSBD	327
Transaction_Validation	CSNBTRV	331

Appendix G. Access-control-point codes

The table in this section lists the CCA access-control commands (control points). The role to which a user is assigned determines the commands available to that user.

Important: By default, you should disable commands. Do not enable an access-control point unless you know why you are enabling it.

The table includes the following columns:

Offset The hexadecimal offset for the command; offsets between X'0000' and X'FFFF' not listed in this table are reserved.

Command name
The name of the command required by the following verbs.

Verb name
The names of the verbs that require that command to be enabled; for example, the Encipher (CSNBENC) verb fails without permission to use the Encipher command.

Entry The entry_point_name of the verb.

Usage Usage recommendations for the command. The abbreviations in this column are explained at the bottom of the page.

See the "Required Commands" section at the end of each verb description for access-control guidance for each verb.

Table 81. Supported CCA commands

Offset	Command name	Verb name	Entry	Usage
X'000E'	Encipher	Encipher	CSNBENC	O
X'000F'	Decipher	Decipher	CSNBDEC	O
X'0010'	Generate MAC	MAC_Generate	CSNBMGN	O
X'0011'	Verify MAC	MAC_Verify	CSNBMVR	O
X'0012'	Reencipher to Master Key	Key_Import	CSNBKIM	O
X'0013'	Reencipher from Master Key	Key_Export	CSNBKEX	O
X'0018'	Load First Master Key Part	Master_Key_Process†	CSNBMKP	SC, SEL
X'0019'	Combine Master Key Parts	Master_Key_Process†	CSNBMKP	SC, SEL
X'001A'	Set Master Key	Master_Key_Process†	CSNBMKP	SC, SEL
X'001B'	Load First Key Part	Key_Part_Import†	CSNBKPI	SC, SEL
X'001C'	Combine Key Parts	Key_Part_Import†	CSNBKPI	SC, SEL
X'001D'	Compute Verification Pattern	Key_Test Key_Test_Extended Key_Storage_Initialization DES_Key_Record_Create DES_Key_Record_Delete DES_Key_Record_List DES_Key_Record_Read DES_Key_Record_Write PKA_Key_Record_Create PKA_Key_Record_Delete PKA_Key_Record_List PKA_Key_Record_Read PKA_Key_Record_Write	CSNBKYT CSNBKYTX CSNBKSI CSNBKRC CSNBKRD CSNBKRL CSNBKRR CSNBKRW CSNDKRC CSNDKRD CSNDKRL CSNDKRR CSNDKRW	R
X'001F'	Translate Key	Key_Translate	CSNBKTR	O
X'0020'	Generate Random Master Key	Master_Key_Process†	CSNBMKP	O, SEL
X'0032'	Clear New Master Key Register	Master_Key_Process†	CSNBMKP	O, SUP

Table 81. Supported CCA commands (continued)

Offset	Command name	Verb name	Entry	Usage
X'0033'	Clear Old Master Key Register	Master_Key_Process†	CSNBMKP	O, SUP
X'0040'	Generate Diversified Key (CLR8-ENC)	Diversified_Key_Generate‡	CSNBDKG	O, SEL
X'0041'	Generate Diversified Key (TDES-ENC)	Diversified_Key_Generate‡	CSNBDKG	O, SEL
X'0042'	Generate Diversified Key (TDES-DEC)	Diversified_Key_Generate‡	CSNBDKG	O, SEL
X'0043'	Generate Diversified Key (SESS-XOR)	Diversified_Key_Generate‡	CSNBDKG	O, SEL
X'0044'	Enable DKG Single Length Keys and Equal Halves for TDES-ENC, TDES-DEC	Diversified_Key_Generate‡	CSNBDKG	SC, SEL
X'0045'	Generate Diversified Key (TDES-XOR)	Diversified_Key_Generate‡	CSNBDKG	O, SEL
X'0046'	Generate Diversified Key (TDESEMVn)	Diversified_Key_Generate‡	CSNBDKG	O, SEL
X'0053'	Load First Asymmetric Master Key Part	Master_Key_Process†	CSNBMKP	SC, SEL
X'0054'	Combine Asymmetric Master Key Parts	Master_Key_Process†	CSNBMKP	SC, SEL
X'0057'	Set Asymmetric Master Key	Master_Key_Process†	CSNBMKP	SC, SEL
X'0060'	Clear New Asymmetric Master Key Buffer	Master_Key_Process†	CSNBMKP	SC, SEL
X'0061'	Clear Old Asymmetric Master Key Buffer	Master_Key_Process†	CSNBMKP	SC, SEL
X'008A'	Generate MDC	MDC_Generate	CSNBMDG	R
X'008C'	Generate Key Set	Key_Generate‡	CSNBKGN	O
X'008E'	Generate Key	Key_Generate‡ Random_Number_Generate	CSNBKGN CSNBRNG	R
X'0090'	Reencipher to Current Master Key	Key_Token_Change	CSNBKTC	R
X'00A0'	Generate Clear 3624 PIN	Clear_PIN_Generate	CSNBPGN	O
X'00A4'	Generate Clear 3624 PIN Offset	Clear_PIN_Generate_Alternate†	CSNBCPA	O
X'00AB'	Verify Encrypted 3624 PIN	Encrypted_PIN_Verify†	CSNBPVR	O
X'00AC'	Verify Encrypted German Bank Pool PIN	Encrypted_PIN_Verify†	CSNBPVR	O
X'00AD'	Verify Encrypted Visa PVV	Encrypted_PIN_Verify†	CSNBPVR	O
X'00AE'	Verify Encrypted Interbank PIN	Encrypted_PIN_Verify†	CSNBPVR	O
X'00AF'	Format and Encrypt PIN	Clear_PIN_Encrypt	CSNBCPE	O
X'00B0'	Generate Formatted and Encrypted 3624 PIN	Encrypted_PIN_Generate†	CSNBEPG	O
X'00B1'	Generate Formatted and Encrypted German Bank Pool PIN	Encrypted_PIN_Generate†	CSNBEPG	O
X'00B2'	Generate Formatted and Encrypted Interbank PIN	Encrypted_PIN_Generate†	CSNBEPG	O
X'00B3'	Translate PIN with No Format-Control to No Format-Control	Encrypted_PIN_Translate†	CSNBPTR	O
X'00B7'	Reformat PIN with No Format-Control to No Format-Control	Encrypted_PIN_Translate†	CSNBPTR	O
X'00BB'	Generate Clear Visa PVV Alternate	Clear_PIN_Generate_Alternate†	CSNBCPA	O
X'00BC'	Generate PIN Change using OPINENC	PIN_Change/Unblock†	CSNBPCU	O
X'00BD'	Generate PIN Change using IPINENC	PIN_Change/Unblock†	CSNBPCU	O
X'00C3'	Encipher Under Master Key	Clear_Key_Import Multiple_Clear_Key_Import	CSNBCKI CSNBCKM	SC
X'00CD'	Lower Export Authority	Prohibit_Export	CSNBPEX	O
X'00D6'	Translate Control Vector	Control_Vector_Translate	CSNBCVT	SC
X'00D7'	Generate Key Set Extended	Key_Generate‡	CSNBKGN	SC, SUP
X'00DA'	Encipher Cryptovariable	Cryptographic_Variable_Encipher	CSNBCVE	NRP, O, SUP
X'00DB'	Replicate Key	Key_Generate‡	CSNBKGN	NR, SC
X'00DF'	Generate CVV	CVV_Generate	CSNBCSG	O
X'00E0'	Verify CVV	CVV_Verify	CSNBCSV	O
X'00E1'	Unique Key Per Transaction, ANSI X9.24	Encrypted_PIN_Translate† Encrypted_PIN_Verify†	CSNBPTR CSNBPVR	O
X'0100'	Digital Signature Generate	Digital_Signature_Generate	CSNDDSG	O, SC
X'0101'	Digital Signature Verify	Digital_Signature_Verify	CSNDDSV	O
X'0102'	Key Token Change	PKA_Key_Token_Change	CSNDKTC	O
X'0103'	PKA Key Generate	PKA_Key_Generate†	CSNDPKG	O, SUP
X'0104'	PKA Key Import	PKA_Key_Import	CSNDPKI	O, SUP

Table 81. Supported CCA commands (continued)

Offset	Command name	Verb name	Entry	Usage
X'0105'	Symmetric Key Export PKCS-1.2/OAEP	PKA_Symmetric_Key_Export	CSNDSYX	SC
X'0106'	Symmetric Key Import PKCS-1.2/OAEP	PKA_Symmetric_Key_Import†	CSNDSYI	O
X'0107'	One-Way Hash, SHA-1	One_Way_Hash	CSNBOWH	R
X'0109'	Data Key Import	Data_Key_Import	CSNBDKM	O
X'010A'	Data Key Export	Data_Key_Export	CSNBDKX	O
X'010B'	Compose SET Block	SET_Block_Compose	CSNDSBC	O
X'010C'	Decompose SET Block	SET_Block_Decompose	CSNDSBD	O
X'010D'	Symmetric Key Generate	PKA_Symmetric_Key_Generate†	CSNDSYG	SC
X'010E'	NL-EPP-5 Symmetric Key Generate	PKA_Symmetric_Key_Generate†	CSNDSYG	O
X'010F'	Reset Intrusion Latch	Cryptographic_Facility_Control†	CSUACFC	SUP
X'0110'	Set Clock	Cryptographic_Facility_Control†	CSUACFC	ID, SUP
X'0111'	Reinitialize Device	Cryptographic_Facility_Control†	CSUACFC	ID, SUP
X'0112'	Initialize Access-Control System	Access_Control_Initialization†	CSUAACI	ID, NRP, SUP
X'0113'	Change User Profile Expiration Date	Access_Control_Initialization†	CSUAACI	ID, SUP
X'0114'	Change User Profile Authentication Data	Access_Control_Initialization†	CSUAACI	ID, NRP, SUP
X'0115'	Reset User Profile Logon-Attempt-Failure Count	Access_Control_Initialization†	CSUAACI	ID, SUP
X'0116'	Read Public Access-Control Information	Access_Control_Maintenance†	CSUAACM	O, ID
X'0117'	Delete User Profile	Access_Control_Maintenance†	CSUAACM	ID, SUP
X'0118'	Delete Role	Access_Control_Maintenance†	CSUAACM	ID, SUP
X'0119'	Load Function-Control Vector	Cryptographic_Facility_Control†	CSUACFC	ID, NRP, SUP
X'011A'	Clear Function-Control Vector	Cryptographic_Facility_Control†	CSUACFC	NR, ID
X'011B'	Force User Logoff	Logon_Control†	CSUALCT	O, SUP
X'011C'	Set EID	Cryptographic_Facility_Control†	CSUACFC	O, SUP
X'011D'	Initialize Master Key Cloning	Cryptographic_Facility_Control†	CSUACFC	O, SUP
X'011E'	PKA Encipher Clear Key	PKA_Encrypt	CSNDPKE	O, SEL
X'011F'	PKA Decipher Key Data	PKA_Decrypt	CSNDPKD	SC, SEL
X'0120'	Generate Random Asymmetric Master Key	Master_Key_Process†	CSNBMKP	SC, SEL
X'0121'	SET PIN Encrypt with IPINENC	SET_Block_Decompose†	CSNBSBD	O
X'0122'	SET PIN Encrypt with OPINENC	SET_Block_Decompose†	CSNBSBD	O
X'0200'	PKA Register Public Key Hash	PKA_Public_Key_Hash_Register	CSNDPKH	O
X'0201'	PKA Public Key Register	PKA_Public_Key_Register†	CSNDPKR	O, SEL
X'0202'	PKA Public Key Register with Cloning	PKA_Public_Key_Register†	CSNDPKR	O, SEL
X'0203'	Delete Retained Key	Retained_Key_Delete	CSNDRKD	O, SEL
X'0204'	PKA Clone Key Generate	PKA_Key_Generate†	CSNDPKG	O, SUP
X'0205'	PKA Clear Key Generate	PKA_Key_Generate†	CSNDPKG	O, SUP
X'0211' – X'021F'	Clone-info (Share) Obtain	Master_Key_Distribution†	CSUAMKD	O, SUP
X'0221' – X'022F'	Clone-info (Share) Install	Master_Key_Distribution†	CSUAMKD	O, SUP
X'0230'	List Retained Key Names	Retained_Key_List	CSNDRKL	O
X'0231'	Generate Clear NL-PIN-1 Offset	Clear_PIN_Generate_Alternate†	CSNBCPA	O
X'0232'	Verify Encrypted NL-PIN-1	Encrypted_PIN_Verify†	CSNBPVR	O
X'0235'	Symmetric Key Import	PKA_Symmetric_Key_Import†	CSNDSYI	O
X'0236'	Symmetric Key Import with PIN Keys	PKA_Symmetric_Key_Import†	CSNDSYI	O
X'023C'	Symmetric Key Generate ZERO-PAD	PKA_Symmetric_Key_Generate†	CSNDSYG	O
X'023D'	Symmetric Key Import ZERO-PAD	PKA_Symmetric_Key_Import†	CSNDSYI	O, SC
X'023E'	Symmetric Key Export ZERO-PAD	PKA_Symmetric_Key_Export†	CSNDSYX	O, SC
X'023F'	Symmetric Key Generate PKCS-1.2/OAEP	PKA_Symmetric_Key_Generate†	CSNDSYG	O, SC
X'0273'	Secure Messaging for Keys	Secure_Messaging_for_Keys	CSNBSKY	O
X'0274'	Secure Messaging for PINs	Secure_Messaging_for_PINs	CSNBSPN	O
X'0276'	Unrestrict Reencipher from Master Key	Key_Export	CSNBKEX	O, SC

Table 81. Supported CCA commands (continued)

Offset	Command name	Verb name	Entry	Usage
X'0277'	Unrestrict Data Key Export	Data_Key_Export	CSNBDKX	O, SC
X'0278'	Add Key Part	Key_Part_Import†	CSNBKPI	SC, SEL
X'0279'	Complete Key Part	Key_Part_Import†	CSNBKPI	SC, SEL
X'027A'	Unrestrict Combine Key Parts	Key_Part_Import	CSNBKPI	O, SC
X'027B'	Unrestrict Reencipher to Master Key	Key_Import	CSNBKIM	O, SC
X'027C'	Unrestrict Data Key Import	Data_Key_Import	CSNBDKM	O, SC
X'027D'	Permit Regeneration Data	PKA_Key_Generate†	CSNDPKG	NRP
X'027E'	Permit Regeneration Data for Retained Keys	PKA_Key_Generate†	CSNDPKG	NRP
X'0290'	Generate Diversified Key (DALL with DKYGENKY Key Type)	Diversified_Key_Generate‡ PIN_Change/Unblock‡	CSNDDKG CSNBPCU	O, SC
X'0291'	Generate CSC 3, 4 and 5 Values	Transaction_Validation†	CSNBTRV	O, SEL
X'0292'	Verify CSC 3 Values	Transaction_Validation†	CSNBTRV	O
X'0293'	Verify CSC 4 Values	Transaction_Validation†	CSNBTRV	O
X'0294'	Verify CSC 5 Values	Transaction_Validation†	CSNBTRV	O
X'0301'	Lower Export Authority, Extended	Prohibit_Export_Extended	CSNBPEXX	O
X'0304'	Error Injection 1	Cryptographic_Facility_Control†	CSUACFC	SC, SUP
X'030B'	Reset Battery-Low Indicator	Cryptographic_Facility_Control†	CSUACFC	ID, SUP
X'030C'	Override DSG Zero-Pad Length Restriction	Digital_Signature_Generate	CSNDDSG	SC
X'030D'	Translate Key from CBC to ECB	Key_Encryption_Translate	CSNBKET	O
X'030E'	Translate Key from ECB to CBC	Key_Encryption_Translate	CSNBKET	O

The following codes are used in this table:

ID	Initial default.
O	Usage of this command is optional; enable it as required for authorized usage.
R	Enabling this command is recommended.
NR	Enabling this command is **not** recommended.
NRP	Enabling this command is **not** recommended for production.
SC	Usage of this command requires special consideration.
SEL	Usage of this command is normally restricted to one or more selected roles.
SUP	This command is normally restricted to one or more supervisory roles.
†	This verb performs more than one function, as determined by the keyword in the *rule_array* parameter of the verb call. Not all functions of the verb require the command in this row.
‡	This verb does not always require the command in this row. Use as determined by the control vector for the key and the action being performed.

Appendix H. Observations on secure operations

This section offers a series of observations about the setup and use of the IBM 4764 and IBM 4758 CCA cryptographic node that you might consider in order to enhance secure operations. Topics include:

- Ensuring code levels match and IBM CCA code is installed
- Using access controls
- Using cryptographic keys
- Using PIN data
- Using status data

Ensuring code levels match and IBM CCA code is installed

The level of the CCA code in the host system should match that used within the coprocessor. Follow the procedures for obtaining code for your system type and ensure you have matching code for both the host environment and for the coprocessor. You should ensure that the host-system code supplied for use with CCA and the coprocessor has not been altered from that obtained from IBM.

Using access controls

The access-control system and the grouping of permissible commands that you can employ are designed to support a variety of security policies. In particular, you can set up the CCA node to enforce a dual-control, split-knowledge policy. Under this policy, once the node is fully activated, no one person should be able to cause detrimental actions other than a denial-of-service attack. To implement this policy, and many other approaches, you must limit your use of certain commands. Therefore, as you design your application, you should consider the commands you must enable or restrict in the access-control system and the implications to your security policy. See Appendix G, “Access-control-point codes,” on page 445 for a table of commands with general guidance in the right-hand column.

The following sections describe:

- Locking the access-control system
- Changing a passphrase
- Defining roles and profiles

Locking the access-control system

For secure operation after performing any initializing processes, consider locking the access-control system. You can render the access-control system unchangeable by deleting any profile that would allow use of the Access Control Initialization command (X’0112’, invoked with the Access_Control_Initialization verb and INIT-AC keyword), and the Delete Role command (X’0118’, invoked with the Access_Control_Maintenance verb and DEL-ROLE keyword). Without these commands, further changes to the access-control roles are not possible.

Before the CCA node is put into normal operation, the access-control setup can be audited using the Access_Control_Maintenance and Cryptographic_Facility_Query verbs. If for any reason the status response is not as anticipated, the node should not be activated for application purposes.

Note: With authority to use either the Initialize Access Control or Delete Role commands, you can delete the DEFAULT role. Deleting the DEFAULT role causes the automatic recreation of the initial DEFAULT role. The initial DEFAULT role permits setting up any capabilities. Users with access to these commands have unlimited authority through manipulation of the access-control system.

Changing a passphrase

The passphrase used to authenticate access to a profile is not communicated out of the DLL or shared library you call with the Logon_Control verb. Rather, the passphrase is hashed to form a cryptographic key that is used to pass the profile identifier and other information to permit the coprocessor to validate access to the profile.

When you change a passphrase with the Access_Control_Initialization verb, use the **PROTECTD** keyword. This causes the passphrase to be encrypted within the DLL or shared-library layer before it is communicated to the coprocessor. This can block a lower-level sniffer program or the CCA trace facility from capturing the new, clear passphrase.

Note: The IBM CCA Support Program prior to Release 2.41 incorporates a trace facility that can be used by IBM support to diagnose obscure problems. This trace facility can capture the clear passphrase information as it flows in the host system. This and other techniques could be used by an adversary to capture clear passphrase information.

If a role contains permission to change a passphrase, the passphrase of any profile can be changed. You should consider if passphrase changing should be permitted and, if so, which roles should have this authority (command X'0114').

If any user reports an inability to log on, this should be reported to someone other than (or certainly in addition to) an individual with passphrase-changing permission.

Defining roles and profiles

The access-control system permits users to define roles and profiles as suits their operation and security needs. Roles and profiles you might consider include the following:

Table 82. Example roles

Setup	A Setup role can be defined that enables loading of required roles, profiles, and other special values such as the Environment ID (EID), Function-Control Vector (FCV), set up of the master-key shares-cloning *m*-of-*n* values, and registration of public keys for later use in key distribution.

Table 82. Example roles (continued)

Administrator	You can establish an Administrator role with extensive supervisory capabilities. The administrative roles could be permitted to change the passphrase of any profile and reset the failure count of any profile (Access_Control_Initialization verb). An individual entrusted with these responsibilities can log on to any role by changing the passphrase of an associated profile and thereby gain the permissions of any role. However, he might not be able to restore the passphrase of the normal user of the profile because in a secure installation he should not know another user's passphrase. You can address this problem in the following ways: • Disabling a role that permits passphrase changing • Ensuring that any suspected authentication problems are reported to someone other than the administrator, who uses roles that permit passphrase changing **Note:** You should set up a duplicate administrator role and associated profiles with a different expiration date to ensure that you have access to those services appropriate to the administrator. This might give you an opportunity to recover should the primary administrator make an error that cannot be rectified.
Security Officer 1 (SO1)	Security Officer 1's role could be permitted to: • Randomly generate a master key • Import a key-encrypting key **Note:** If you employ introduction of keys in parts (Key_Part_Import or Master_Key_Process verbs; see "Using cryptographic keys" on page 452 for more information), the first-part and second-part permissions should be assigned to SO1 and SO2, respectively.
Security Officer 2 (SO2)	Security Officer 2's role could be permitted to: • Set a master key • Import keys **Note:** If you employ introduction of keys in parts (Key_Part_Import or Master_Key_Process verbs; see "Using cryptographic keys" on page 452 for more information), the first-part and second-part permissions should be assigned to SO1 and SO2, respectively.
Default	You must have a Default role. When a host thread is not logged on, requests from such a thread are performed based on the permissions set in the default role. You should enable only those commands necessary for normal operations. At a maximum, only those functions specifically required should be enabled. All sensitive or unusual requirements should be processed following a logon to an appropriate profile (and thus its role).
Application user *n*	As required, *n* application-specific roles and associated profiles should be established for processing portions of applications with security requirements different from those permitted under the Default role. For example, enabling any of the key export verbs could release keys to an adversary. Such operations are candidates for selective enablement under control of a specific role.

In all cases, enable only the commands needed to accommodate the permitted applications.

Using cryptographic keys

Cryptographic keys are typically passed across the CCA interface as encrypted objects in key-token data structures. Rogue processes on your host system might be able to capture a copy of such keys, or the contents of the key-storage data set might be copied. You must rely on your operating system security, system-operational security, and physical security to counter any threat from an encrypted-key copy. Do not allow a rogue process to make use of the encrypted key. Your environmental security policy should consider how rogue processes could make use of a copy of an encrypted key. You must consider the handling of any clear keys.

Keys are further discussed under these topics:
- CCA asymmetric DES keys
- Clear key parts
- Key export
- Key unwrapping
- Clear key operations
- DES replicated keys
- RSA keys

CCA asymmetric DES keys

With CCA, you can often make use of a unique capability afforded through the CCA control vector and command architecture. CCA enables DES keys to have asymmetric properties. Using MAC or MACVER, ENCIPHER/DECIPHER, IMPORTER/EXPORTER, PINGEN/PINVER, and IPINENC/OPINENC key types, you can separate which systems and processes can reverse various cryptographic functions.

MAC and MACVER

A node that has a MAC-class key can both generate and verify a DES MAC value. A CCA node having only the key with MACVER properties is unable to create an authentication code or MAC with the key. Thus, data recipients who receive only a MACVER key can be enabled to validate data, but are prevented from producing a MAC on data potentially altered to their advantage.

Also, a DES MAC is computed by enciphering the cleartext data. You need to ensure that an adversary is denied access to enciphering processes with the key used in the MAC computations. For this reason, consider using the MAC and MACVER keys rather than the DATA-class keys.

By default, DATA-class keys perform in both encipher and MAC generation and verification operations.

ENCIPHER and DECIPHER

You can separate the ability to reverse a DES ciphering process. Use an ENCIPHER key at the data-encryption node, and only supply the data-decryption node with the DECIPHER form of the ciphering key.

You might also find uses for enciphering data where you want to disallow the possibility that the data is ever deciphered. You can determine the equivalence of two copies of source data by comparing their enciphered value. Thus, you can store an enciphered copy of data and determine later that other data is not equivalent without revealing the clear value of the original data. A hash process can give the same effect, but in some environments, encryption might be faster than hashing.

Consider the use of the CIPHER-class keys rather than the more general DATA-class keys because the CIPHER-class keys have reduced capabilities and thus offer fewer opportunities for misuse.

IMPORTER and EXPORTER

You use a key in these key classes to set up a one-way key-distribution channel. In fact, it is generally considered inappropriate to have the same key-value encrypted as both an IMPORTER and as an EXPORTER on the same CCA node. You can use the functionality of the Key_Generate verb and the one-way key-distribution channel to distribute CCA asymmetric DES keys to node pairs.

For example, a data originator can encipher data and be sure that no one can decipher the data on his node using an ENCIPHER-class key. The DECIPHER-class copy of the key, probably with the CCA export-allowed control-vector bit turned off, can be sent over the one-way key-distribution channel to another node. Only there can the data be deciphered.

As another example, a key-distribution center can originate and distribute a no-export-allowed MAC key to one node and the matching MACVER key, also with the no-export-allowed attribute, can be sent to another node. In this scenario (and if the CCA master keys are managed and audited in a secure manner), the MAC verification node has no means of producing a valid MAC on altered data.

PINGEN and PINVER

You can segregate the ability to create a PIN value from the ability to validate a PIN value (and PIN offsets, PVV values, and so on).

OPINENC and IPINENC

As with one-way key-distribution channels, you can set up one-way encrypted PIN-block distribution channels. This can enable you to further segregate which nodes in your network can perform various forms of PIN processing.

Clear-key parts

Typically, two or more users each install a key part to instantiate a cryptographic key. The key parts are exclusive-ORed together to form the final key. CCA supports this option with the Key_Part_Import and Master_Key_Process verbs. You can force the separation of key-part installation into two groups by enabling the first-part capability and the key-part-combine capability in different roles. You can also use different roles for processing master keys versus other key types.

Release 2.41 added options to the Key_Part_Import verb to provide additional separation in functional capability. The preexisting Combine Key Parts command (X'001C', invoked with the **MIDDLE** and **LAST** keywords) processes key-part information and completes a key by turning off the key-part control-vector bit. Two new options are added:

- Use the Add Key Part command (X'0278'), invoked with the **ADD-PART** keyword, to exclusive-OR additional information into an incomplete key.
- Use the Complete Key Part command (X'0279'), invoked with the **COMPLETE** keyword, to turn off the key-part control-vector bit.

These commands enable you to designate an individual with the authority to accept a key without providing that individual with the possibility of modifying the value of the key.

Pre-exclusive-OR
Additional information, such as a variant or a control-vector value, can be exclusive-ORed into a key-encrypting key so other keys processed with the modified key-encrypting key can be assigned alternative control vectors and alternative functionality. This pre-exclusive-OR technique can be valuable when exchanging keys with non-CCA systems. However, an adversary, if permitted to employ the technique, could change the control vector of a key to his advantage. To counter this threat, separate key-part loading responsibilities so that an individual can oversee the operations and complete a key without having the authority to add information to the key.

Clear key-part interception
Key-part information flows in the clear through your host system. If you view this as an unacceptable risk, consider the following alternatives:

Random generation of master keys
If you need to backup the master key or have the same master key in an additional coprocessor, use master-key cloning to securely transfer the value of the master key to additional coprocessors.

Random key-generation and RSA-based key-distribution
Distribute RSA-encrypted, randomly generated DES data or DES key-encrypting keys to the node where the key should be instantiated. With CCA and this strategy, you do not need key parts or secrecy. Continue to use two-channel distribution techniques to ensure integrity of the public-key distribution. This is true even when certificates are in use; you must provide integrity for the top-level public key.

Key export

Be careful when allowing the export of keys from your system, especially when enabling the PKA_Symmetric_Key_Generate verb and the other key-export verbs:

- Data_Key_Export
- Key_Export
- PKA_Symmetric_Key_Export

In particular, the verbs PKA_Symmetric_Key_Export and PKA_Symmetric_Key_Generate permit the export of selected classes of keys under any public key. Ensure that the target nodes are legitimate and that only appropriate processes have use of these verbs, EXPORTER keys, and public keys.

Consider taking advantage of the export-allowed control-vector bit. By switching this bit off, you can prevent a key from being exported.

Note: Master-key-encrypted RSA private keys or retained RSA private keys cannot be exported from a CCA node.

Key unwrapping

An RSA private key that is used in a symmetric-key exchange to unwrap or decrypt a DES key should not typically be allowed to perform digital signature signing. This is because the Digital_Signature_Generate verb could be used with the **ZERO-PAD** keyword to reveal the DES key. To prevent this from happening, generate RSA keys used for symmetric key-exchange with their key-usage flag bits set to KM-ONLY. The key-usage flag bits should **not** be set to KEY-MGMT, as this allows the key to be used for both symmetric key exchange and digital signature signing.

Clear-key operations

Remember that the following CCA verbs operate with keys in the clear. Carefully consider using them.

CSNDSYX PKA_Symmetric_Key_Export	A clear, unprotected public key is entered under which DATA keys can be enciphered. This operation can be disallowed through the access-control system. This is a potentially insecure operation. Any DATA key with the XPORT-OK bit on in the control vector (bit 17) can be exported to the owner of the associated private key.
CSNDSYG PKA_Symmetric_Key_Generate	A clear, unprotected public key is entered under which a freshly generated KEK or DATA key can be created. This operation can be disallowed through the access-control system. This is a potentially insecure operation if you set up a key-distribution channel with an inappropriate public key. Be sure that you know who has access to the associated private key.
CSNBCKI Clear_Key_Import CSNBCKM Multiple_Clear_Key_Import	Either 8 or 16 bytes of clear information can be accepted to be returned as an encrypted DATA key. This operation can be disallowed through the access-control system. The clear-key information could be intercepted as it is transmitted to the coprocessor. Consider freshly generating a key using Key_Generate.
CSNBKPI Key_Part_Import	This verb requires use of two commands (using the **FIRST**, with **MIDDLE** and **LAST** keywords), or three commands (using the **FIRST**, with **ADD-PART** and **COMPLETE** keywords) to complete the establishment of a productive key of any type. The key information is passed in the clear. These operations can be disallowed through the access-control system. The access controls can enforce a dual-control policy, but the key components still pass across the general interface in the clear. As an alternative, consider using PKA_Symmetric_Key_Import and Key_Import to receive keys from another source. An adversary might be able to change the value of a key by employing the **MIDDLE** or **ADD-PART** keywords. If the key were for an **IMPORTER** or **EXPORTER**, this could be used later to alter the control vector of an imported or exported key. This pre-exclusive OR technique is sometimes viewed as a legitimate means for altering control vectors. (See “Pre-exclusive-OR” on page 454 and “Clear key-part interception” on page 454 for more information.)

CSNBMKP Master_Key_Process	Use of the **FIRST**, **MIDDLE**, and **LAST** keywords employs clear data to establish the value of a master key. These operations can be disallowed through the access-control system. The preferred means to establish a master key is through random generation (**RANDOM** keyword) or through the master-key cloning process. (See "Clear key-part interception" on page 454 for more information.)
CSNDDSG Digital_Signature_Generate	The **ZERO-PAD** option allows a hash to be specified which can be as long as the modulus of the key. If the hash is in fact a public-key-encrypted key, the key can be recovered in the clear. RSA keys intended for key management should be restricted to that usage by specifying **KM-ONLY** in PKA_Key_Token_Build and PKA_Key_Generate.

DES replicated keys

A *replicated key* is defined as a double-length DES key having equal left and right halves. Such a key performs as a single-length key. Because CCA always uses double-length key-encrypting keys and PIN-processing keys, it is sometimes advantageous to generate or install replicated keys to inter-operate with non-CCA systems that are using single-length DES keys. Be careful in permitting the generation and use of replicated keys as overcoming the work factor to attack single-length DES keys might be within the capability of certain adversaries. You can block the generation of replicated DES keys in the Key_Generate and the Diversified_Key_Generate verbs by not enabling optional commands.

Beginning with Release 2.41, the Key_Import, Key_Export, Data_Key_Import, and Data_Key_Export verbs no longer permit use of a replicated key-encrypting key to import or export a non-replicated key unless special permission is granted using the unrestrict commands, X'0276', X'0277', X'027B', and X'027C'.

RSA keys

When generating RSA key-pairs, you must consider these issues:
- Regeneration data
- Attributes for signature and key management
- Retained keys

Regeneration data

When an application calls the PKA_Key_Generate verb, it can specify regeneration_data that seeds the random number generator. If the same regeneration data is supplied in the future, the same RSA key-pair is generated. This can be useful in a development environment, but is inappropriate in a production environment.

Starting with Release 2.54, the access control system only allows use of regeneration data when you enable command X'027D', Permit Regeneration Data.

Attributes for signature and key management

The skeleton key-token that an application provides when calling the PKA_Key_Generate verb assigns usage attributes to the generated private key. You can assign attributes that only permit the key to perform in digital signature generation or only in unwrapping or importing symmetric keys, or you can assign attributes that enable the key to generate both digital signatures and unwrap keys.

If you use the Digital_Signature_Generate verb with the **ZERO-PAD** option and supply a wrapped symmetric key, the verb decrypts the wrapped key and an application can then recover the symmetric key in the clear. This most likely is undesirable. Generally, private keys intended for use in key-management applications should be generated with the **KM-ONLY** attribute to block the use of the key in the Digital_Signature_Generate verb. For private keys used to generate digital signatures, consider applying the **SIG-ONLY** attribute.

Retained keys

In some applications you must not release an RSA private key in any form from the device in which it is generated. Using the **RETAIN** keyword with the PKA_Key_Generate verb prevents the coprocessor from outputting the private key. Alternative choices permit the key to be returned to the application wrapped under the master key or a transport key, or returned in the clear with access-control permission.

You can confirm that a key has been generated and retained in the coprocessor using the Retained_Key_List verb and the verification of a digital signature made with the retained key. Because no means exist to import an RSA private key into retained-key storage, the list of key labels returned from the Retained_Key_List verb demonstrates the existence of a particular named key. You can confirm that the same label does not exist in host-system key-storage using the PKA_Key_Record_List verb.

Using PIN data

A personal identification number (PIN) is generally passed across the CCA interface as an encrypted object in an encrypted PIN-block. Typically, verbs protect PIN values using encryption. The exceptions are described in the following table:

Table 83. PIN data exceptions

CSNBCPE Clear_PIN_Encrypt	Encrypts a clear-PIN value and returns the result under an OPINENC class key. This operation can be disallowed through the access-control system. Unrestricted usage can permit the construction of a dictionary of encrypted PIN values.
CSNBPGN Clear_PIN_Generate	Generates the PIN for a given account number. This operation can be disallowed through the access-control system. Unrestricted usage permits the generation of PIN numbers for the specified account numbers, using information that can be known to an adversary.

Using status data

Status is returned from the CCA application through the Access_Control_Maintenance and Cryptographic_Facility_Query verbs. An adversary with access to the computing system could alter coprocessor status responses.

Certain status information can be obtained from the Miniboot component of the coprocessor through the coprocessor load utility (CLU). This response is signed and can be validated using the CLU utility.

Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department VM9A, MG39/201
8501 IBM Drive
Charlotte, NC 28262-8563
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM License Agreement for Non-Warranted Programs.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM agreement for licensed internal code

Read Before Using

IMPORTANT

YOU ACCEPT THE TERMS OF THIS IBM LICENSE AGREEMENT FOR MACHINE CODE BY YOUR USE OF THE HARDWARE PRODUCT OR MACHINE CODE. PLEASE READ THE AGREEMENT CONTAINED IN THIS BOOK BEFORE USING THE HARDWARE PRODUCT.

You accept the terms of this Agreement[14] by your initial use of a machine that contains IBM Licensed Internal Code (called “Code”). These terms apply to Code used by certain machines IBM or your reseller specifies (called “Specific Machines”). International Business Machines Corporation or one of its subsidiaries (“IBM”) owns copyrights in Code or has the right to license Code. IBM or a third party owns all copies of Code, including all copies made from them.

If you are the rightful possessor of a Specific Machine, IBM grants you a license to use the Code (or any replacement IBM provides) on, or in conjunction with, only the Specific Machine for which the Code is provided. IBM licenses the Code to only one rightful possessor at a time.

Under each license, IBM authorizes you to do only the following:

1. execute the Code to enable the Specific Machine to function according to its Official Published Specifications (called “Specifications”);

14. Form Z125-4144

2. make a backup or archival copy of the Code (unless IBM makes one available for your use), provided you reproduce the copyright notice and any other legend of ownership on the copy. You may use the copy only to replace the original, when necessary; and
3. execute and display the Code as necessary to maintain the Specific Machine.

You agree to acquire any replacement for, or additional copy of, Code directly from IBM in accordance with IBM's standard policies and practices. You also agree to use that Code under these terms.

You may transfer possession of the Code to another party only with the transfer of the Specific Machine. If you do so, you must 1) destroy all your copies of the Code that were not provided by IBM, 2) either give the other party all your IBM-provided copies of the Code or destroy them, and 3) notify the other party of these terms. IBM licenses the other party when it accepts these terms. These terms apply to all Code you acquire from any source.

Your license terminates when you no longer rightfully possess the Specific Machine.

Actions you must not take

You agree to use the Code only as authorized above. You must not do, for example, any of the following:

1. Otherwise copy, display, transfer, adapt, modify, or distribute the Code (electronically or otherwise), except as IBM may authorize in the Specific Machine's Specifications or in writing to you;
2. Reverse assemble, reverse compile, or otherwise translate the Code unless expressly permitted by applicable law without the possibility of contractual waiver;
3. Sublicense or assign the license for the Code; or
4. Lease the Code or any copy of it.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	eServer
eServer i5	i5/OS
IBM	iSeries
OS/400	pSeries
RACF	S/390
xSeries	z/OS
zSeries	

The following terms are the trademarks of other companies:

MasterCard	MasterCard International, Inc.
RSA	RSA Data Security, Inc.
Visa	Visa International Service Association
SET	SET Secure Electronic Transaction™ LLC
Windows	Microsoft Corporation

List of Abbreviations

ANSI	American National Standards Institute
AIX	Advanced Interactive Executive operating system
API	Application Programming Interface
ASCII	American National Standard Code for Information Interchange
ATC	Application Transaction Counter
ATM	Automated Teller Machine
CBC	Cipher-Block Chaining
CCA	Common Cryptographic Architecture
CDMF	Commercial Data Masking Facility
CKSN	Current Key Serial Number
CNM	Cryptographic Node Management (utility)
COBOL	Common Business-Oriented Language
CV	Control Vector
CVC	Card-Verification Code
CVV	Card-Verification Value
DEA	Data Encryption Algorithm
DES	Data Encryption Standard
DOW	Day of the week
DMA	Direct Memory Access
EBCDIC	Extended Binary Coded Decimal Interchange Code
ECB	Electronic Code Book
EEPROM	Electrically erasable, programmable read-only memory
EMV	Europay, MasterCard, VISA
FIPS	Federal Information Processing Standard
IBM	International Business Machines
ICSF	Integrated Cryptographic Service Facility
I/O	Input/Output
IPL	Initial Program Load
ISO	International Standards Organization
KEK	Key-Encrypting Key
KM	Master key
MB	Megabyte
MAC	Message authentication code
MD5	Message Digest 5 Hashing Algorithm
MDC	Modification detection code
MK	Master key
MKVP	Master-key verification pattern
NIST	National Institute of Science and Technology (USA).
OEM	Original Equipment Manufacturer
OS/400	Operating System/400®
PAN	Personal Account Number
PIN	Personal Identification Number
PKA	Public Key Algorithm
POST	Power-On Self Test
PROM	Programmable Read-Only Memory
RACF	Resource Access Control Facility
RAM	Random Access Memory
ROM	Read-Only Memory
RSA	Rivest, Shamir, and Adleman
SET	Secure Electronic Transaction
SHA	Secure Hashing Algorithm
SNA	Systems Network Architecture
SSL	Secure Socket Layer
TLV	Tag, Length, Value
TVV	Token-validation value
UKPT	Unique key per transaction

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 might 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.
- The *TotalStorage® Enterprise Storage Server®* documentation. Definitions of published parts of this vocabulary are identified by the symbol (E) after the definition.

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. A technique for moving data between main storage and input/output devices.

adapter. A printed circuit card that modifies the system unit to allow it to operate in a particular way.

address. In data communication, the unique code assigned to each device or workstation connected to a network. A character or group of characters that identifies a register, a particular part of storage, or some other data source or data destination. (A) To refer to a device or an item of data by its address. (A) (I)

Advanced Interactive Executive (AIX) operating system. IBM's implementation of the UNIX®[15] operating system.

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 was one of a family of general purpose midrange systems with a single operating system, Operating System/400, that provides application portability across all models. AS/400 is now referred to as IBM eServer iSeries.

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. A process used to verify the integrity of transmitted data, especially a message. (T) In computer security, a process used to verify the user of an information system or protected resources.

authorization. The right granted to a user to communicate with or make use of a computer system. (T) 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.

B

bus. In a processor, a physical facility along which data is transferred.

byte. A binary character operated on as a unit and usually shorter than a computer word. (A) A string that consists of a number of bits, treated as a unit, and representing a character. A group of eight adjacent binary digits that represents one EBCDIC character.

15. UNIX is a trademark of UNIX Systems Laboratories, Incorporated.

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)

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. 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 (CCA). The CCA API is the programming interface described in this document.

concatenation. An operation that joins two characters or strings in the order specified, forming one string whose length is equal to the sum of the lengths of its parts.

configuration. The manner in which the hardware and software of an information processing system are organized and interconnected. (T) The physical and logical arrangement of devices and programs that constitutes a data processing system.

control program. A computer program designed to schedule and to supervise the programs running in a computer system. (A) (I)

control vector (CV). In CCA, 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.

coprocessor. In this document, the IBM 4758 and IBM 4764 PCI Cryptographic Coprocessors, generally also when using the CCA Support Program.

Cryptographic Node Management utility (CNM). One of the utility programs supplied with the CCA Support Program. It enables you to initialize the coprocessor access controls and the cryptographic master keys.

cryptography. The transformation of data to conceal its meaning.

D

data. A representation of facts or instructions in a form suitable for communication, interpretation, or processing by human or automatic means. Data includes constants, variables, arrays, and character strings. Any representations such as characters or analog quantities to which meaning is or might be assigned. (A)

data-encrypting key. A key used to encipher, decipher, or authenticate data. 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.

decipher. To convert enciphered data into clear data. Synonym for *decrypt*. Contrast with *encipher*.

decode. To convert data by reversing the effect of some previous encoding. (A) (I) In the CCA products, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard (DES). Contrast with *encode* and *decipher*..

decrypt. To decipher or decode. Synonym for *decipher*. Contrast with *encrypt*.

device driver. A program that contains the code needed to attach and use a device.

device ID. In the IBM 4758 and IBM 4764 CCA implementations, a user-defined field in the configuration data that can be used for any purpose the user specifies. For example, it can be used to identify a particular device, by using a unique ID similar to a serial number.

diagnostic. Pertaining to the detection and isolation of errors in programs, and faults in equipment.

directory server. A server that manages key records in key storage by using an Indexed Sequential Access Method.

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.

encipher. To scramble data or to convert data to a secret code that masks the meaning of the data to unauthorized recipients. Synonym for *encrypt*. Contrast with *decipher*. See also *encode*.

enciphered data. Data whose meaning is concealed from unauthorized users or observers. See also *ciphertext*.

encode. To convert data by the use of a code in such a manner that reconversion to the original form is possible. (T) In the CCA implementation, decode and encode relate to the Electronic Code Book mode of the Data Encryption Standard. Contrast with *decode*. See also *encipher*.

encrypt. Synonym for *encipher*. (T) To convert clear text into ciphertext. Contrast with *decrypt*.

erasable programmable read-only memory (EPROM). (1) (2) A type of memory chip that can retain its contents without electricity. Unlike the programmable read-only memory (PROM), which can be programmed only once, the EPROM can be erased by ultraviolet light and then reprogrammed (E).

exit routine. In the CCA products, a user-provided routine that acts as an extension to the processing provided with calls to the security API.

EXPORTER key. In the CCA implementation, a type of DES Key-Encrypting Key that can encipher a key at a sending node. Contrast with *IMPORTER key*.

F

feature. A part of an IBM product that can be ordered separately.

Federal Information Processing Standard (FIPS). FIPS is a standard published by the US National Institute of Science and Technology.

financial PIN. A Personal Identification Number used to identify an individual in some financial transactions. To maintain the security of the PIN, processes and data structures have been adopted for creating, communicating, and verifying PINs used in financial transactions. See also *Personal Identification Number*.

Flash-Erasable Programmable Read-Only Memory. A memory that has to be erased before new data can be saved into the memory.

H

host. In this publication, same as host computer or host processor. The machine in which the coprocessor resides. In a computer network, the computer that usually performs network-control functions and provides end-users with services such as computation and database access. (T)

I

IMPORTER key. In the CCA implementation, a type of DES Key-Encrypting Key that can decipher a key at a receiving mode. Contrast with *EXPORTER key*.

initialize. In programming languages, to give a value to a data object at the beginning of its lifetime. (I) To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine. (A)

Integrated Cryptographic Service Facility (ICSF). ICSF is an IBM licensed program that supports the cryptographic hardware feature for the high-end System/390® processor running in a z/OS environment.

International Organization for Standardization (ISO). 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.

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). A KEK is a key used for the encryption and decryption of other keys. Contrast with *data-encrypting key*.

key half. In the CCA implementation, one of the two DES keys that make up a double-length key.

key identifier. In the CCA implementation, a 64-byte variable which is either a key label or a key token.

key label. In the CCA implementation, an identifier of a key-record in key storage. See “Key labels” on page 137 and “Key-label content” on page 242.

key storage. In the CCA implementation, a data file that contains cryptographic keys which are accessed by key label.

key token. In the CCA implementation, a data structure that can contain a cryptographic key, a control vector, and other information related to the key.

L

link. The logical connection between nodes including the end-to-end control procedures. The combination of physical media, protocols, and programming that connects devices on a network. In computer programming, the part of a program, in some cases a single instruction or an address, that passes control and parameters between separate portions of the computer program. (A) (I) To interconnect items of data or portions of one or more computer programs. (T) In SNA, the combination of the link connection and link stations joining network nodes.

M

make file. A composite file that contains either device configuration data or individual user profiles.

master key (MK, KM). In computer security, the top-level key in a hierarchy of key-encrypting keys.

Message Authentication Code (MAC). A number or value derived by processing data with an authentication algorithm, The cryptographic result of block cipher operations on text or data using a cipher block chaining (CBC) mode of operation, A digital signature code.

migrate. To move data from one hierarchy of storage to another. To move to a changed operating environment, usually to a new release or a new version of a system.

Modification Detection Code (MDC). In cryptography, 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.

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. A configuration of data-processing devices and software programs connected for information interchange. An arrangement of nodes and connecting branches. (T)

node. In a network, a point at which one-or-more functional units connect channels or data circuits. (I)

O

Operating System/400 (OS/400). OS/400 is an operating system for the IBM eServer iSeries, formerly known as Application System/400 computers.

P

panel. The complete set of information shown in a single image on a display station screen.

parameter. In the CCA security API, an address pointer passed to a verb to address a variable exchanged between an application program and the verb.

password. In computer security, a string of characters known to the computer system and a user; the user must specify it to gain full or limited access to a system and to the data stored within it.

Personal Identification Number (PIN). 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.

profile ID. In the CCA implementation, the value used to access a profile within the CCA access-control system.

plaintext. Data that has nor been altered by a cryptographic process. Synonym for *cleartext*. See also *ciphertext*.

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. In computer security, a key that is known only to the owner and used together with a public-key algorithm to decipher data. The data is enciphered using the related public key. Contrast with *public key*. See also *public-key algorithm*.

procedure call. In programming languages, a language construct for invoking execution of a procedure. (I) A procedure call usually includes an entry name and possible parameters.

profile. Data that describes the significant characteristics of a user, a group of users, or one-or-more computer resources.

protocol. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I) In SNA, the meanings of and the sequencing rules for requests and responses used to manage the network, transfer data, and synchronize the states of network components. A specification for the format and relative timing of information exchanged between communicating parties.

public key. In computer security, a key that is widely known, and used with a public-key algorithm to encrypt data. The encrypted data can be decrypted only with the related private key. Contrast with *private key*. See also *public-key algorithm*.

Public-Key Algorithm (PKA). In computer security, PKA is an asymmetric cryptographic process that uses a public key to encrypt data and a related private key to decrypt data. Contrast with *Data Encryption Algorithm* and *Data Encryption Standard algorithm*. See also *Rivest-Shamir-Adleman algorithm*.

public-key hardware. That portion of the security module in an IBM 4758 and IBM 4764 that performs modulus-exponentiation arithmetic.

R

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. A value that provides a specific result as opposed to a general result. Contrast with *return code*.

replicated key-half. In the CCA implementation, a double-length DES key where the two halves of the clear-key value are equal.

Resource Access Control Facility (RACF). RACF is 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. A code used to influence the execution of succeeding instructions. (A) A value returned to a program to indicate the results of an operation requested by that program. In the CCA implementation, a value that provides a general result as opposed to a specific result. Contrast with *reason code*.

Rivest-Shamir-Adleman (RSA) algorithm. 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 server. In the CCA implementation, the functions provided through calls made to the security API.

server. On a Local Area Network, a data station that provides facilities to other data stations; for example, a file server, a print server, a mail server. (A)

session. In network architecture, for the purpose of data communication between functional units, all the activities that take place during the establishment, maintenance, and release of the connection. (T) 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.

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)

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

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

throughput. A measure of the amount of work performed by a computer system over a given period of time; for example, number of jobs per day. (A) (I) A measure of the amount of information transmitted over a network in a given period of time; for example, a network's data-transfer-rate is usually measured in bits per second.

TLV. A widely used construct, Tag, Length, Value, to render data self-identifying. For example, such constructs are used with EMV smart cards.

token. In a Local Area Network, the symbol of authority passed successively from one data station to another to indicate the station is temporarily in control of the transmission medium. (T) A string of characters treated as a single entity.

trace file. A file that contains a record of trace information for the selected processing.

U

Unique key per transaction (UKPT). UKPT is a cryptographic process that can be used to decipher PIN blocks in a transaction.

user-exit routine. A user-written routine that receives control at predefined user-exit points.

user ID. User identification.

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.

z/OS. An operating system for the IBM eServer product line that uses 64-bit real storage (E).

Numerics

4758. IBM 4758 PCI Cryptographic Coprocessor.

4764. IBM 4764 PCI-X Cryptographic Coprocessor.

Index