IBM

Encryption Facility for z/OS Version 1.10

- Principles of operations and options explained
- Examples of setup and use of all the features
- Expert considerations and recommendations

Patrick Kappeler
Peggy LaBelle
Pekka Hanninen
Lennie Dymoke-Bradshaw
Stanley Jon
Greg Boyd

Redbooks

ibm.com/redbooks

International Technical Support Organization

Encryption Facility for z/OS Version 1.10

January 2007

SG24-7318-00

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

First Edition (January 2007)

This edition applies to Version 1, Release 1.0 of IBM Encryption Facility for z/OS (product number 5655-P97).

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

CICS®	ibm.com®	Redbooks™
DB2®	IBM®	System z9™
DFSMSdfp™	IMS™	System z™
DFSMSdss™	iSeries™	VTAM®
DFSMShsm™	MVS™	z/Architecture®
ESCON®	OS/390®	z/OS®
eServer™	RACF®	z9™
FICON®	Redbooks (logo) ™	zSeries®

The following terms are trademarks of other companies:

Java, JVM, Sun, Sun Microsystems, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbook introduces IBM Encryption Facility for z/OS®, Program Product 5655-P97, from the description of its principles of operation to information related to its setup intended for system programmers and security officers.

We include recommendations, based on our own experience, in the chapters addressing these topics, including warnings and explanations about problems we discovered during our own trials.

This redbook provides also several examples of the use of the following Encryption Facility for z/OS features:

- Encryption Services
- Encryption Facility for z/OS Java™ Client
- DFSMSdss™ Encryption feature

This book also addresses in detail the many options proposed by the product and provides you with considerations and recommendations regarding proper selection of these options in the planned context of use.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world at the IBM Washington System Center, on behalf of the International Technical Support Organization, Poughkeepsie Center.

Patrick Kappeler led this IBM Redbook project. For the past 35 years, he has held many international specialist and management positions in IBM, all dealing with mainframe technical support. He is now part of the European Products and Solutions Support Center, located in Montpellier, France, where his domain of expertise is e-business security on System z™ technology. He has authored many IBM Redbooks™ and still extensively writes and presents about this topic.

Peggy LaBelle is an Advisory Software Engineer with IBM. She has worked on various components of z/OS for her entire career as developer, team leader, and designer. She recently joined the z/OS ICSF project as one of the designers. Prior to that she worked on RACF®.

Lennie Dymoke-Bradshaw has been an IT Specialist at IBM for 10 years. He has worked in mainframe computing since 1975, when he started applications programming in PL/I. He currently works in Integrated Technology Services in the U.K. delivering System z services and consultancy. During his career, he has maintained an interest in programming. In recent years, he has concentrated on engagements in cryptography. He also has expertise in RACF and JES3.

Pekka Hanninen is an IT Specialist working with the Integrated Technology Services team in Finland. He has 35 years of experience in IBM Large Systems software. He has worked at IBM for 10 years, and his areas of expertise include RACF, cryptography, and security administration. He holds certificates for CISSP, CISA, and CISM.

Stanley Jon is an Advisory IT Specialist with IBM Canada. He has been with IBM and working with the mainframe platform for fewer than 10 years. He is part of the Canadian team that supports z/OS defect and Q&A questions from Canadian customers, as well as z/OS Q&A questions from American customers. He has been supporting ICSF since 1999 and System SSL since 2003.

Greg Boyd is a certified IT Specialist in the IBM Advanced Technical Support organization. He provides technical marketing support for IBM eServer™ zSeries® cryptographic hardware and software, and assists customers with installation and technical questions about the products. He recently obtained his CISSP and has focused on cryptographic support for the last two years. He has more than 28 years of experience in the computer industry, both as a systems programmer and in application development, and he has technical leadership experience in project management and in design and implementation of data processing solutions. Greg has experience with large z/OS operating systems and related products including JES2, VTAM®, CICS®, and RACF. He is also experienced in performance measurement and tuning.

Figure 1 Stanley, Patrick, Peggy, Pekka, Lennie, and Greg

Thanks to the following people for their contributions to this project:

Chris Rayns
International Technical Support Organization, Poughkeepsie Center

Michael B. Schwartz
International Technical Support Organization, Poughkeepsie Center

Michael Kelly
z/OS Cryptographic Software Design

Ernie Nachtigall
System z Security Americas ATS

Marilyn Allmond
WSC primary for Americas System z Crypto and ICSF support and security team lead

Victoria Mara
Development DFSMS Program Manager

Jason Katonica
z/OS Development

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

1

What the Encryption Facility for z/OS is and is not

This chapter introduces IBM Encryption Facility for z/OS V1.10, Licensed Program Product 5655-P97. We describe at a high level the product features, the environments in which the features execute, what functions they provide, and how the business needs that led IBM to market this product are met by the host-based Encryption Facility for z/OS.

We provide further detailed information about hardware and software prerequisites in Chapter 3, “Principles of operation and hardware and software requirements for the Encryption Facility” on page 19.

1.1 Why the Encryption Facility for z/OS?

The Encryption Facility for z/OS (Encryption Facility) processes data at rest and is intended for encryption of media whose contents must be securely transported, that is, physically moved, for example, shipped in a truck, or electronically sent over non-secure links.

The “security” of the movement here covers both network eavesdropping and unauthorized reading of physical media containing sensitive information. Consequences of such an unauthorized disclosure of information can be severe, as illustrated by many examples in the press, where companies’ finances and image were affected after loosing track of physical media with sensitive contents known to be easily readable once one has access to the media itself.

IBM Encryption Facility for z/OS exploits the existing strengths of the mainframe and the IBM z/OS operating system. It is a host-based facility that leverages existing centralized key management in z/OS and the hardware encryption capabilities of IBM mainframes.

When sharing encrypted data between z/OS systems, the Encryption Facility for z/OS software can also use the mainframe’s hardware to compress the data, before the data is encrypted and written to the tape or DASD media.

A natural extension to the encryption of the moved data is the encryption of archived data, thus the added encryption and decryption capabilities to the DFSMSdss DUMP and RESTORE functions. as part of the features offered in the Encryption Facility for z/OS.

As of the writing of this book, the Encryption Facility for z/OS is at the V1.R10 level and is delivered as Program Product 5655-P97.

1.2 What Program Product 5655-P97 contains

There are three components (or features) delivered under the generic name of “Encryption Facility for z/OS”:

- Encryption Services feature
  This program executes exclusively on z/OS (z/OS V1R4 and later) and exploits the System z integrated hardware cryptography to encrypt and decrypt data files or data sets. The Encryption Services feature also provides ways to protect the key used to encrypt the data with a secret password or by using public key cryptography with the RSA algorithm. See Appendix B, “Some encryption basics” on page 155 for an introduction to RSA public key cryptography and see Chapter 3, “Principles of operation and hardware and

software requirements for the Encryption Facility" on page 19 for a detailed description of the encryption and decryption processes used by the Encryption Facility for z/OS.
The Encryption Services feature supports as input data to be encrypted or decrypted data in physical sequential data sets in members of partitioned data sets (PDS) and partitioned data sets extended (PDSE) and in z/OS UNIX® files (HFS or zFS).

Additionally, it can also use the large block interface for output files written to tape in order to optimize both performance and media space.

- Encryption Facility for z/OS Client
  This feature is a Web deliverable, separately licensed program (see "Client license agreement and additional assistance" on page 5) that provides two components in a single package:
  - The Encryption Facility for z/OS Client itself, which is a Java technology-based code that enables client systems to decrypt and encrypt data files in an interoperable way with the other features of the Encryption Facility for z/OS. As with the Encryption Services feature, the Java Client provides data encryption key protection through password or RSA public key cryptography.
    Note that in the rest of the book, we call this feature the "Java Client."
  - The Decryption Client for z/OS is a native z/OS load module that can be installed and used for, as the name implies, decrypting files that were originally encrypted using the Encryption Services (note that we do not mention here that files encrypted using the Java Client can be decrypted using the Decryption Client. Look at "Client license agreement and additional assistance" on page 5 for explanations on the conditions of use of the client).
- DFSMSdss Encryption feature
  This feature enables encryption of DUMP data sets created by DFSMSdss and supports decryption during RESTORE.

  Note that DFSMShsm™ exploits the encryption support provided by DFSMSdss in the DFSMShsm full-volume dump function and the associated restore functions, including both full-volume and data set-level restore.

The Encryption Services and the DFSMSdss Encryption features are priced features of Program Product 5655-P97.

Download Encryption Facility for z/OS Java Client from:

http://www.ibm.com/servers/eserver/zseries/zos/downloads/#asis

The Decryption Client for z/OS, as already mentioned, is also included in this download and is SMP/E installable.

Optional data compression

Note also that the Encryption Services, the DFSMSdss Encryption features, and the Decryption Client, that is all the System z “native” programs, can optionally compress data and to do so exploit the hardware-accelerated compression available on the System z platform.

Functional restrictions of the Encryption Facility Java Client

The Encryption Facility Java Client does not support data compression and decompression.

It does not support either receiving data encrypted with a secure Triple-DES (T-DES) key (option ENCTDES in 5.2.1, “Control statement keywords for encryption (CSDFILEN)” on page 57).

Figure 1-1 summarizes the possible exchanges between systems hosting the different features of the Encryption Facility for z/OS.

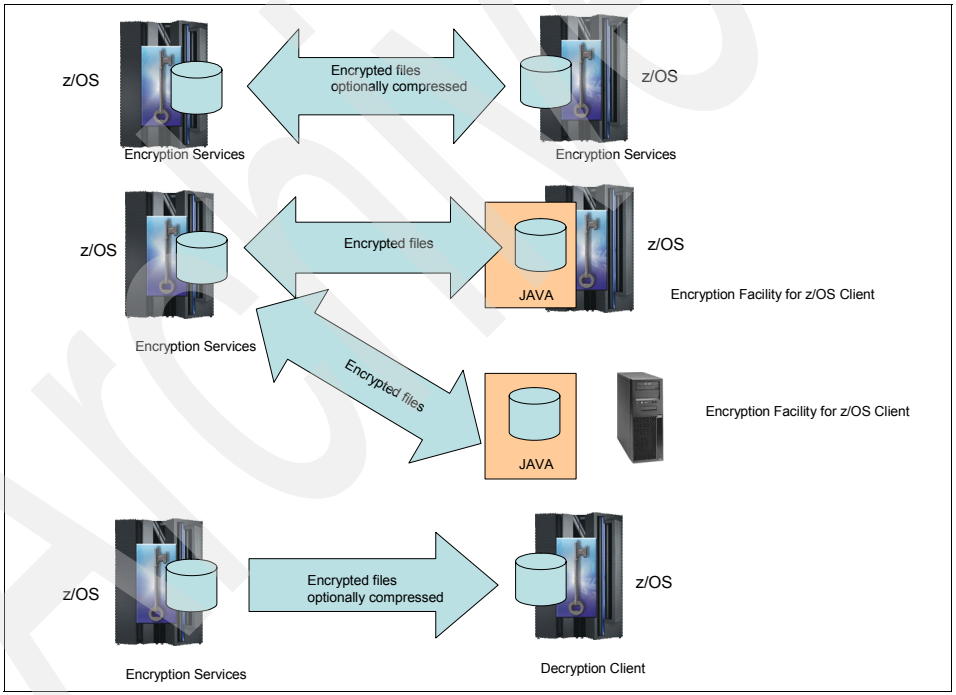

Figure 1-1 Encryption services and clients

Notice that the Java Client can run on a z/OS system or any other system that supports Java. However, we do not see here a Java Client exchanging files with another Java Client. This is contrary to the terms and conditions of the Java

Client licenses. Client license agreement and additional assistance are two items covered in the next section.

1.3 Client license agreement and additional assistance

Although it is technically possible to have an Encryption Facility for z/OS Java Client exchanging data with another Java Client, the terms and conditions of the License Agreement for Non-Warranted Programs that is downloaded as part of the Java Encryption Client and the z/OS Decryption Client, states that: “You may only use the Product for decrypting information or data that was encrypted by IBM's Encryption Facility for z/OS, or for encrypting information or data to be decrypted by IBM's Encryption Facility for z/OS.”

The downloadable IBM Encryption Facility for z/OS Client, and the included IBM Decryption Client, are programs provided “as is”, that is without warranty and without product support. However, IBM does intend to review and analyze problems reported, and in order to do so, provides e-mail addresses to report problems to in the Web pages. At the discretion of IBM, an updated Encryption Facility for z/OS Client Web download (which includes the Java Client and the Decryption Client for z/OS) may be provided.

1.4 Considerations about data encryption

In this section, we discuss general considerations about the use of data encryption and the consequences it might have on the relevant processes and organizations.

Encryption and decryption of data

Although the concept of secure data exchange using encryption and decryption is quite clear to people familiar with the data processing technology, confusion arises as to where the process of encryption and decryption needs to occur:

- Encryption and decryption can be performed against data “on the wire,” that is being performed as part of the data transfer protocol. This is typically the case for TCP/IP communications when they are protected by the Secure Sockets Layer (SSL)/Transport Layer Security (TLS) or IP Security Architecture (IPSec) protocols. In this case, the encryption protocol operates on the data presented at the sending endpoint, and decryption occurs at the receiving endpoint, implying that data is “in clear” when not being handled by the transport protocol. Note that “in clear” can have a relative meaning here because an IPSec virtual private network (VPN) can transfer data already encrypted by other means, the point being that the VPN-specific encryption

covers data only while it is “on the wire,” that is between the two endpoints of the VPN.

- Encryption and decryption is performed on data stored in a file or data set. In this situation there are still two cases to consider:
  - The data is “alive”, meaning that it can be used at any time by an application and can therefore be rendered to its clear value on the fly in order to be processed and be re-encrypted when returned to the file or data set. This is typically what happens with IBM Data Encryption Tool for IMS™ and DB2® Databases.
  - The data is put “at rest” before being encrypted. The data is explicitly made unavailable to applications and then goes through the encryption process. It has to go through a decryption process first before being made available again to applications.

Important: The Encryption Facility for z/OS encrypts data “at rest” on DASD or tape medias. That is, the data is not intended to be used by applications until it is decrypted.

Note also that the Encryption Facility for z/OS does not provide any electronic means of transport for the encrypted data. It is up to the installation to select and implement any required transport, be it a physical media movement or a purely electronic one.

Compression and encryption

Data compression involves a process where a data pattern is replaced by a binary number of a fixed length. At decompression time, the original data pattern is retrieved using the binary number. It takes two conditions for the compression process to be efficient:

- The decompression process must know which data patterns correspond to which binary numbers provided as a result of the compression. One of the most popular compression algorithm, the Ziv-Lempel algorithm, produces a so called “compression dictionary” that provides this correspondence between the fixed length binary numbers (dictionary indexes) and the pattern for which they stand.
- The longer the data pattern the fixed length binary number represents, the better the compression ratio is. The compression algorithm works in such a way that the more repetitive a data pattern is, the higher the chance is to have it represented in its totality by a single binary number. In other words, input data repetitiveness is a mandatory condition to get efficient compression.

Compression must occur before encryption: The result of the encryption process is as close as possible, by design, to random data, carefully hiding any

repetitions that might occur in the clear source data. Therefore, encrypted data does not lend itself to efficient compression and might even, with the Ziv-Lempel algorithm, yield an inverted compression ratio (compressed data are bigger than the original uncompressed input[1]).

The Encryption Services and the DFSMSdss Encryption feature can optionally invoke System z hardware compression prior to encrypting the data. The System z platform uses the Ziv-Lempel algorithm with a static decompression dictionary that is then encrypted and stored along with the encrypted data.

Important: If you compress encrypted data, as is most of the time when you record data on a tape media, be prepared to see the volume of encrypted, then compressed, data on the tape to exceed the volume of the same data when it is compressed only.

1.5 Sizing and capacity planning considerations

We discuss sizing and capacity planning in Appendix C, “Encryption Facility for z/OS: Performance and sizing” on page 165.

1.6 What it means to use cryptography

In this section, we provide some general considerations about the use of cryptography. These considerations complement the functional technical details provided in Appendix B, “Some encryption basics” on page 155.

Encryption and archival of data

Encrypting data as part of the archival process also brings specific considerations, which we briefly discuss here.

What you need to encrypt and to decrypt

The encryption process involves:

- Input clear data that will be transformed and delivered as an encrypted output.

[1] Expansion of data occurs when the fixed length binary number replaces data shorter than the binary number itself.

- An encryption algorithm. This is a formal description of all the transformations the input data must go through before being delivered as an encrypted output. Note that the algorithms used by the Encryption Facility for z/OS are commonly used in the Industry and happen to be public algorithms. In other words, the algorithm logical flows of operations is known; however, how the transformations are performed inside the algorithm (sequential order of transformations, iterative rounds of the process, and so on) is driven by a secret value, the "key."
- The key is a binary pattern that, as implied earlier, is the secret that needs to be preserved when exchanging encrypted data.

Encryption keys, algorithms, and performance

Appendix B, "Some encryption basics" on page 155 explains in more detail the algorithms and types of keys that we use with the Encryption Facility. For the moment, we stress three basic principles:

- The value of the key cannot, by any means, be guessed or predicted, thus the absolute requirement in the world of cryptography of quasi-perfect randomness in numbers generation.
- It is true for all algorithms that the longer the key, the harder it is to guess its value starting from an encrypted output ("harder" meaning usually in the world of cryptography a few centuries of CPU time worth of computations, based on computers technologies currently in use).
- If the secret key has been thrown away, there is no way of recovering the clear value of encrypted data. This is particularly relevant to encrypted archives of which the secret key might have to be preserved and remain secret for perhaps decades.

Note that data encryption can also serve another purpose other than ensuring the privacy of the data. By encrypting data with a secret key, you also demonstrate that you possess the key, and this is used in many secure protocols as a means of authenticating the party with which you are communicating.

The intrinsic performance of the encryption/decryption process depends on two factors:

- The encryption/decryption algorithm selected. Some algorithms are more demanding than others on computing resources.
- The length of the key.

Note however that the actual throughput of a specific cryptographic algorithm implementation usually exhibits a wide swing in encryption/decryption performance figures, depending on the length of the bursts of data submitted to the algorithm implementation. The highest throughput being reached when processing a long block of data, for example, one megabyte-long record, for each

encryption call, and the poorest throughput is usually met with very short blocks data, for example, 64-byte long records, per encryption call.

Cryptographic coprocessors and key management

Encryption can be performed entirely by software. This can be an adequate solution for a small-to-moderate cryptographic workload that uses secret keys with a rather short lifetime in order to mitigate the key discovery exposure. As soon as the workload grows, the software implementation becomes less and less adequate in regards of performance. It quickly reaches a level of MIPS consumption deemed unacceptable by many installations.

Because secret keys must be made available to the encryption program in storage, they are becoming more vulnerable to compromise as their lifetime increases. In today's common practices, a key lifetime, depending on the length of the key, of one or two days is no longer considered short.

Hardware cryptographic coprocessors are designed to handle efficiently the performance and security problems met with software implementations:

- These are processors specialized for high speed cryptographic computations, and thus provide excellent response time and, above all, save the CPU MIPS for what they are intended: to run applications.
- They can implement techniques to protect the secret keys and make them highly secure. The technique used in the IBM cryptographic secure coprocessors is the encryption of the secret keys whenever they stay outside of the coprocessor, be it in system storage or on disks. The keys are encrypted by a "master key," which is another secret key kept and protected inside the coprocessor itself. The encryption key is decrypted by the master key inside the coprocessor's secure physical enclosure at use time only. This technique and its implementation in the IBM cryptographic coprocessors is further explained in Chapter 2, "System z integrated cryptography: A reminder" on page 11.

> **Note:** Whenever we refer to a "clear key" in the rest of this book, we are designating keys that are not kept encrypted with the master key. Conversely, we call "secure keys" keys that are protected by an encryption with the master key.

It is not unusual in installations heavily using cryptographic services to have to deal with hundreds, if not thousands, of long-lifetime secret keys. It is then mandatory to use a key management infrastructure that provides facilities for at least:

- The automated generation of keys with quasi-perfect randomness.

- The retrieval of keys using labels so that they can be easily and unambiguously identified to operators and applications (which, incidentally, also brings the possible issues of naming conventions).
- The disposal of keys that are no longer needed. Remember, however, that for many usages, the keys might have to remain active and of course protected for several years.
- Maintain a constant level of protection for secure keys over long period of time.

2

System z integrated cryptography: A reminder

The System z platform offers standard and optional hardware cryptographic devices along with proper software components in the different operating systems to provide applications with APIs to invoke the system's hardware cryptography and with key repository management facilities. In this chapter, we focus on the support z/OS provides for applications using hardware cryptography.

The z/OS base component that provides the cryptographic APIs that eventually invoke the hardware coprocessor is the Integrated Cryptographic Service Facility (ICSF). The ICSF APIs implement the IBM Common Cryptographic Architecture (CCA) and are available for programs written in assembler or high-level languages. Note that the IBM CCA supports a hierarchical structure of keys where keys can be encrypted by other keys (key-encrypting keys, KEKs), the master key being at the top of the hierarchy.

ICSF provides cryptographic coprocessors administration facilities for those coprocessors that require a master key to be set.

ICSF also provides key repositories in the form of two VSAM data sets were keys can be kept in "key tokens" in clear value or encrypted under a KEK or under the coprocessors master keys. The VSAM data sets are the Cryptographic Key Data Set (CKDS) and the Public Key Data Set (PKDS). The key tokens in the CKDS

and the PKDS are given a user- or system-defined label that is used for their retrieval and maintenance.

Figure 2-1 is a schematic view of the hardware cryptography implementation in the System z environment. Note that the hardware cryptography technology shown here is the one available on the IBM System z9™ and eServer zSeries 990 and 890 platforms. The zSeries 800 and 900 host other, although functionally compatible, types of cryptographic coprocessors. We provide a reminder of the complete range of hardware cryptographic coprocessors available in the System z environment in Chapter 3, “Principles of operation and hardware and software requirements for the Encryption Facility” on page 19.

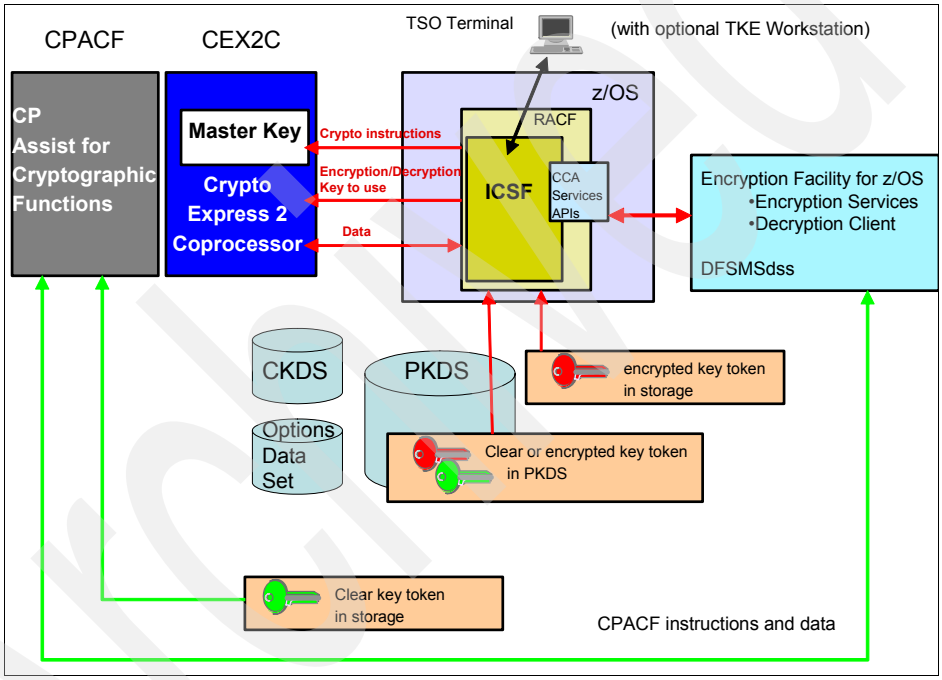

Figure 2-1 Hardware cryptography use by the Encryption Facility for z/OS

The interactions between the functional blocks shown in Figure 2-1 are as follows:

- ICSF is a z/OS started task that offers cryptographic APIs to the Encryption Facility features and drives the requests to the Crypto Express2 Coprocessors (CEX2C).

- Note however that the CEX2C is a “secure” coprocessor in that it contains a master key used to encrypt keys to be kept in storage or in the PKDS data set. The master key resides in the coprocessor hardware only and is used to decrypt, internally to the coprocessor, the “secure” keys provided so that they can be used to encrypt or decrypt data.
- Installing and maintaining the secret master key is a task that security officers can perform from TSO/E terminals or from an optional Trusted Key Entry (TKE) workstation, the latter for a very high security level of the interactions between the security officers and the CEX2C.

  If there is more than one secure coprocessor to which ICSF has access, all coprocessors must have been set with the same master key value.
- ICSF needs other data sets to operate; the CKDS, which is not used in the context of the Encryption Facility for z/OS, use of cryptographic hardware, and an options data set that contains the ICSF started task startup parameters.
- The Encryption Facility for z/OS also uses the CPACF facility that resides in each processing unit of the z9, z990, and z890.

  The CPACF is directly called by instructions in the Encryption Facility load modules.
- The CPACF operates only with clear keys.

Note: For the rest of the discussion, we focus on the use of ICSF as done by the Encryption Facility for z/OS that does not involve using the ICSF CKDS data sets. We, therefore, restrain ourselves to considerations pertaining to the PKDS only in the rest of the book.

2.1 Enablement of cryptographic coprocessors

The cryptographic hardware functions required by the Encryption Facility for z/OS are available when Feature Code 3863 has been installed. These are the CPACF only when using the password option of the Encryption Facility for z/OS, and at least one CEX2C feature when using the RSA option (these options are described in Chapter 5, “Encryption Services to Encryption Services or the Decryption Client” on page 51).

Order FC 3863 with the system or after receiving the system. In the latter case, FC 3863 comes as a Licensed Internal Code (LIC) module on a CD, to be installed through the system HMC. The enablement of the cryptographic services is not disruptive to the system’s operations.

2.2 RACF involvement in ICSF operations

RACF, or an equivalent product, is called by ICSF to verify that requestors have permission to access the requested cryptographic service and the designated key. This access control is enforced by the definition of RACF profiles in the CSFSERV and CSFKEYS classes of resources with proper access lists.

When relevant profiles are not defined, access is freely granted to the service and the key.

2.3 Audit trails

ICSF provides SMF records with administrative audit data related to the ICSF operations themselves and the security officer access to the CEX2C coprocessors. These are the SMF type 82 records.

RACF provides auditing data and violation reports on the CSFSERV and CSFKEYS classes of resources as specified by the RACF auditors. The audit data are provided in SMF records types 80, 81, and 83.

2.4 Performance and utilization data

ICSF collects performance and utilization data for the Crypto Express2 Coprocessors. They are delivered in the SMF types 30, 70, 72, and 82 subtypes 18, 19, and 20.

There is no SMF data collection for the utilization of the CPACF.

2.5 Logical partitioning and System z hardware cryptography exploitation

The functional drawing in Figure 2-1 on page 12 describes cryptographic operations at the logical partition level. Up to 16 different logical partitions can share the same CEX2C card, with two CEX2C cards available in each CEX2C feature plugged in the system.

> **Note:** Each logical partition sharing a physical CEX2C coprocessor is granted a “domain” in the coprocessor. A “domain” is a set of physical resources that are dedicated to this very logical partition. Among the resources provided in a domain are registers intended to securely hold the master keys installed for the ICSF instance that runs in the logical partition.
>
> Domains are part of the CEX2C coprocessor FIPS 140-2 evaluation at level 4 and therefore provide complete isolation and secrecy of individual master keys between the sharing logical partitions.
>
> Domains are manually allocated in the logical partitions image profiles and in the ICSF options data sets.

There is no notion of logical partition sharing with the CPACF because the facility is directly available to any logical partition dispatched on any processing unit.

2.6 Sysplex and System z hardware cryptography

Several instances of ICSF can coexist in a sysplex, with one ICSF instance per sysplex member, and they can share the CKDS and PKDS data sets, that is, share the keys stored in these data sets. Note however that:

- CKDS or PKDS sharing implies having set up the same master keys in the coprocessors domains used by the sharing ICSF instances.
- There is as of now no automatic PKDS cache consistency support between the sharing ICSF. When an ICSF changes the PKDS DASD contents, the change is not automatically propagated to the other ICSF instances that share the CKDS in the sysplex.

  However, it is expected that such changes are manageable manually because they occur at low frequency and manual processes do exist in ICSF to re-synchronize local PKDS caches to the latest PKDS DASD change.

2.7 More about key management using ICSF

ICSF provides, along with the PKDS secure keys repository, built-in processes for the management of those keys:

- The keys stored in the PKDS are RSA public and private keys (see Appendix B, “Some encryption basics” on page 155 for an explanation about the RSA algorithm and keys). The public keys are kept unencrypted, because a public key is intended to be public knowledge, while the private keys are kept encrypted by the master key installed in the Crypto Express2 coprocessors domain.
- Key pair records are given a label by the user at their creation that is used further for management of these key records.
- The PKDS has an affinity with the master key value that has been installed for the using instance or instances of ICSF.
- Note that a PKDS backup can be created with a VSAM backup utility. This is strictly a copy of the PKDS contents; the secrets are preserved as the keys are copied exactly as they stay on the PKDS, that is, in clear or encrypted form. However, a PKDS backup copy is usable only by an ICSF instance that uses the same master key value as the one used with the original PKDS.
- Keys can remain in the PKDS for long periods of times and stored private keys might outlast the planned life for a given master key value; in other words, the protected key’s life is longer than the life of the protecting key. For such occurrences, ICSF provides for the ICSF administrator the Set New Master Key and PKDS re-encipher functions, allowing a change of the master key without affecting and disclosing the value of the RSA keys kept in the PKDS.

  During a PKDS re-encipher, encrypted keys are brought from the PKDS into the CEX2C coprocessor and, securely inside the secure physical boundary of the coprocessor, they are decrypted with the current master key and re-encrypted with the new master key.

We provide further information about setting the master key in “Installation of the secure coprocessors master key” on page 36.

2.8 ICSF bibliography

The following guides are available with z/OS:

- *z/OS ICSF Overview*, SA22-7519
- *z/OS ICSF System Programmer's Guide*, SA22-7520
- *z/OS ICSF Application Programmer's Guide*, SA22-7522

- *z/OS ICSF Administrator's Guide*, SA22-7521
- *z/OS ICSF Messages*, SA22-7523
- *z/OS Trusted Key Entry Workstation User's Guide 2000*, SA22-7524

3

Principles of operation and hardware and software requirements for the Encryption Facility

After you decide to use the Encryption Facility for z/OS, the detailed planning begins. This chapter describes at a high level how the Encryption Facility for z/OS operates, the encryptions options, and the relevant hardware and software requirements.

3.1 Principles of operation: High-level view

Figure 3-1 on page 21 provides a high-level description of the Encryption Facility for z/OS principles of operations:

1. The data to be encrypted are provided as input to the symmetric encryption process (T-DES or AES encryption).
2. The data encryption key is generated in one of these ways:
   - It is derived from a password value that is a shared secret between the issuer of the encrypted data and the recipient.
   - It is generated dynamically as an ICSF secure key token (T-DES key only). The value of the key is never to appear in system storage.
   - It is dynamically generated and is kept in its clear form to proceed with the encryption of data (T-DES or AES key).
3. The dynamically generated data encryption key is then encrypted with the recipient's RSA public key, (or the RSA archival key) that the issuer of the data must have received and installed earlier in the PKDS. The encrypted data encrypting key is kept along with the encrypted data to form the output file of the Encryption Facility for z/OS.

Note: For those of you not familiar with the symmetric and asymmetric cryptographic processes, see Appendix B, "Some encryption basics" on page 155 for more explanations pertaining to these processes.

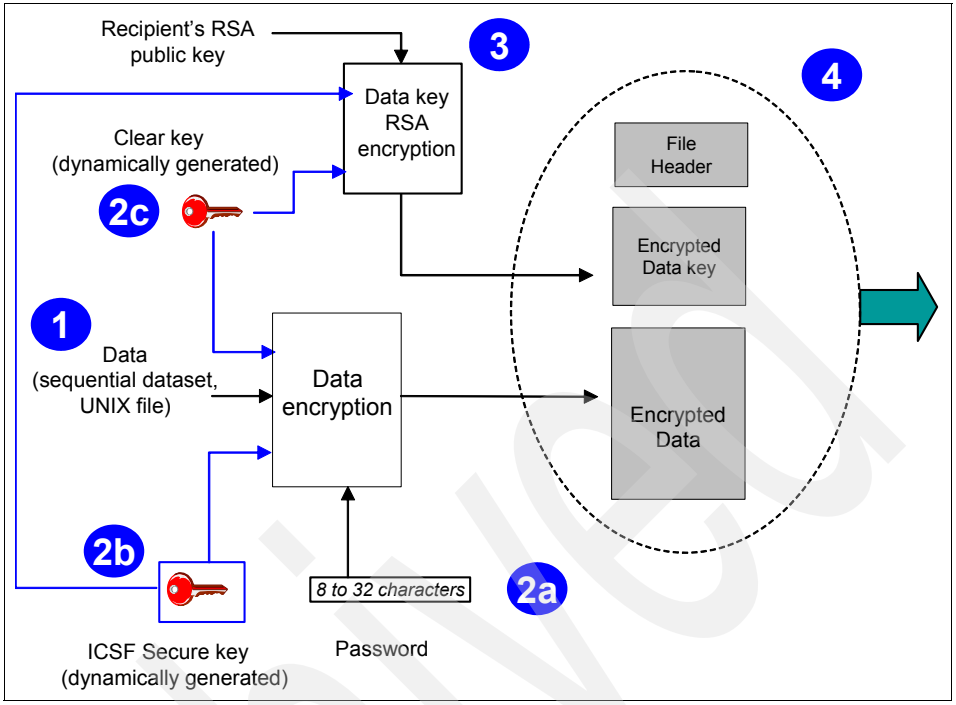

Figure 3-1 Principles of operation: Data encryption overview

The data decryption process is shown in Figure 3-2 on page 22:

1. If the password option has been chosen at encryption time, the decryption key is derived from the password entered at the decryption site.
   Or if the RSA protection of the data encrypting key has been selected, the encrypted data key is extracted from the file and decrypted using the recipient's RSA private key.
2. The data are decrypted using the recovered data encrypting key.

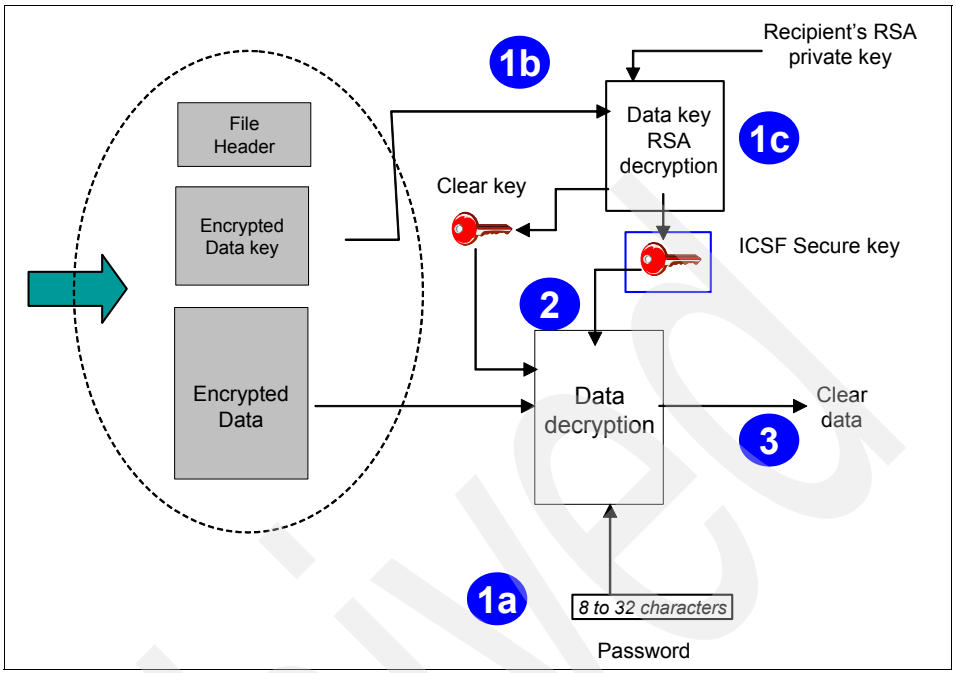

Figure 3-2 Principles of operation: Data decryption overview

3.2 Summary of hardware and software prerequisites

This section addresses the hardware and software prerequisites that are required by each one of the Encryption Facility for z/OS features.

3.2.1 The Encryption Services feature

For those of you not aware of the set of cryptographic devices offered on the zSeries and System z platforms, see 3.3, "Exploitation of the z/OS integrated cryptography" on page 26 for more information.

See Table 3-1 for the Encryption Services feature prerequisites.

Table 3-1 Encryption Services feature prerequisites

	Systems, coprocessors, and z/OS version and release
Hardware	► zSeries 800 and 900 with CCF and optional PCICC (limitations apply when there is no PCICC on the system; see “The PCI Cryptographic Coprocessor (PCICC)” on page 30): The CCF has to be enabled with Feature Code 0875 and the PCICC needs the Function Control Vector (FCV) FC 0865 diskette to be imported at the HMC. ► zSeries 890 and 990: The Feature Code 3863 must be installed and at least one PCIXCC or CEX2C is available. ► System z9: The Feature Code 3863 must be installed and at least one CEX2C is available.
Operating systems	z/OS or z/OS.e V1R4, V1R5, V1R6, V1R7, and later
ICSF	ICSF HCR770A or HCR770B on z/OS V1R4 and V1R5 ICSF HCR7720, HCR7730, HCR7731 and later on z/OS V1R6 and later
Compression option	Yes
Encryption options	Data encryption key: ► Derived from an 8 to 32 characters password[a] ► T-DES key (168 bits[b]) ► 128-bit AES key ► Secure T-DES key (168 bits) The T-DES and AES data keys are encrypted and decrypted using: ► RSA public key encryption (512, 1024, 2048 bits) ► RSA private key decryption (512, 1024, 2048 bits)

a. T-DES or AES-128 keys are derived from the password value using the PKCS#12 file protection algorithm.

b. T-DES are effectively 168 bits long. However, for historical reasons, they come with one extra parity bit per byte, thus making the space used by a T-DES key a 24 byte long (192 bits) field.

3.2.2 IBM DFSMSdss Encryption feature

The DFSMSdss Encryption feature has requirements similar to the Encryption Services feature, as shown in Table 3-2.

Table 3-2 DFSMSdss Encryption feature prerequisites

	Systems, coprocessors, and z/OS version and release
Hardware	► zSeries 800 and 900 with CCF and optional PCICC (limitations apply when there is no PCICC on the system; see “The PCI Cryptographic Coprocessor (PCICC)” on page 30): The CCF has to be enabled with Feature Code 0875 and the PCICC needs the Function Control Vector (FCV) FC 0865 diskette to be imported at the HMC. ► zSeries 890 and 990: The Feature Code 3863 must be installed and at least one PCIXCC or CEX2C is available. ► System z9: The Feature Code 3863 must be installed and at least one CEX2C is available.
Operating systems	z/OS or z/OS.e V1R4, V1R5, V1R6, or V1R7
ICSF	ICSF HCR770A or HCR770B on z/OS V1R4 and V1R5 ICSF HCR7720, HCR7730, HCR7731 and later on z/OS V1R6 and later
Compression option	Yes
Encryption options	Data encryption key: ► Derived from an 8 to 32 characters password[a] ► T-DES key (168 bits[b]) ► 128-bit AES key ► Secure T-DES key (168 bits) The T-DES and AES data keys are encrypted and decrypted using: ► RSA public key encryption (512, 1024, 2048 bits) ► RSA private key decryption (512, 1024, 2048 bits)

a. T-DES or AES-128 keys are derived from the password value using the PKCS#12 file protection algorithm.
b. T-DES are effectively 168 bits long. However, for historical reasons, they come with one extra parity bit per byte, thus making the space used by a T-DES key a 24 byte long (192 bits) field.

3.2.3 The Decryption Client

The Decryption Client has the same set of requirements as the Encryption Services feature, as shown in Table 3-3.

Table 3-3 Decryption Client prerequisites

	Systems, coprocessors, and z/OS version and release
Hardware	► zSeries 800 and 900 with CCF and optional PCICC (limitations apply when there is no PCICC on the system; see “The PCI Cryptographic Coprocessor (PCICC)” on page 30): The CCF has to be enabled with Feature Code 0875 and the PCICC needs the Function Control Vector (FCV) FC 0865 diskette to be imported at the HMC. ► zSeries 890 and 990: The Feature Code 3863 must be installed and at least one PCIXCC or CEX2C is available. ► System z9: The Feature Code 3863 must be installed and at least one CEX2C is available.
Operating systems	z/OS or z/OS.e V1R4, V1R5, V1R6, or V1R7
ICSF	ICSF HCR770A or HCR770B on z/OS V1R4 and V1R5 ICSF HCR7720, HCR7730, HCR7731 and later on z/OS V1R6 and later
Decompression option	Yes
Decryption options	Data encryption key: ► Derived from an 8 to 32 characters password[a] ► T-DES key (168 bits[b]) ► 128-bit AES key ► Secure T-DES key (168 bits) The T-DES and AES data keys are encrypted and decrypted using: ► RSA public key encryption (512, 1024, 2048 bits) ► RSA private key decryption (512, 1024, 2048 bits)

a. T-DES or AES-128 keys are derived from the password value using the PKCS#12 file protection algorithm.
b. T-DES are effectively 168 bits long. However, for historical reasons, they come with one extra parity bit per byte, thus making the space used by a T-DES key a 24 byte long (192 bits) field.

3.2.4 The Java Client

Table 3-4 shows the Java Client requirements, as of the writing of this book.

Table 3-4 Java Client prerequisites

	Systems and JVM™ version or release
Hardware	Any platform that can host a Java Virtual Machine (JVM) at a proper level.
JVM	On z/OS: ► IBM SDK for z/OS, Java 2 Technology Edition, 5655-I56, at PTF UQ90449 or higher (SDK1.4.2) ► IBM Developer Kit for OS/390®, Java 2 Technology Edition, 5655-D35, at PTF UQ88094 or higher (SDK1.3.1) On other platforms than z/OS: ► Sun™ SDK 5.0 ► IBM JVM at SDK1.4.2 ► JVM with a JCE cryptographic provider installed that supports all the required algorithms There is a special set of information, including PTF requirements for IBM eServer iSeries™ and other platforms, provided in the *readme* file in the downloadable package of the Encryption Facility for z/OS Java Client. Download the Java Client from the following Web site: `http://www.ibm.com/servers/eserver/zseries/zos/downloads/#asis`
Compression option	No
Encryption options	Data encryption key: ► Derived from an 8 to 32 characters password[a] ► T-DES key (168 bits[b]) ► 128-bit AES key The T-DES and AES data keys are encrypted and decrypted using: ► RSA public key encryption (512, 1024, 2048 bits) ► RSA private key decryption (512, 1024, 2048 bits)

a. T-DES or AES-128 keys are derived from the password value using the PKCS#12 file protection algorithm.
b. T-DES are effectively 168 bits long. However, for historical reasons, they come with one extra parity bit per byte, thus making the space used by a T-DES key a 24 byte long (192 bits) field.

3.3 Exploitation of the z/OS integrated cryptography

We provide an overview of the z/OS integrated cryptography infrastructure in Chapter 2, “System z integrated cryptography: A reminder” on page 11. In this section, we provide more details about the elements of the infrastructure that are required for the Encryption Facility for z/OS to operate.

Note: The Java Client, as provided, does not use the IBM classes that invoke an IBM hardware cryptographic coprocessor if available on the Java host system. We are, therefore, implicitly excluding the Java Client from all discussions in the book that pertain to the use of hardware cryptography.

Table 3-5 summarizes the exploitation of the system cryptographic devices by the Encryption Facility for z/OS. The next sections provide detailed information about these devices and on which cryptographic services are called.

Table 3-5 Minimum cryptographic hardware requirements

System model	Password option	RSA option	
		Up to 1024-bit key	2048-bit key
z900	CCF	CCF	PCICC
z800	CCF	CCF	PCICC
z990	CPACF	CEX2C/PCIXCC	CEX2C/PCIXCC
z890	CPACF	CEX2C/PCIXCC	CEX2C/PCIXCC
z9	CPACF	CEX2C/PCIXCC	CEX2C/PCIXCC

3.3.1 The hardware cryptographic facilities in z9, z990, and z890

In this section, we describe the hardware cryptographic devices that are available for these systems.

CP Assist for Cryptographic Functions (CPACF)

Each system central processor (CP) has an assist processor on the chip in support of cryptographic operations. The CP Assist for Cryptographic Function offers a set of cryptographic functions that supports symmetric algorithms using clear keys, that is, keys that are not kept encrypted with a master key.

The CPACF cryptographic services include encryption and decryption with DES, T-DES, and AES-128 (on z9 only), MAC message authentication, and SHA-1 and SHA-256 (on z9 only) hashing. These functions are directly available to application programs, because they can be called by problem state z/Architecture® instructions. Alternatively, these functions can also be called though the Integrated Cryptographic Service Facility (ICSF) component of z/OS by an ICSF-aware application, at the cost of additional instructions path length, with, however, the potential advantage of exploiting some ICSF built-in capabilities that might be relevant to the specific use case.

Out of the five problem-state instructions that System z software can use to invoke CPACF services, the following instructions are used by the Encryption Facility for z/OS:

- KMC: Cipher Message with Chaining
  The KMC instruction performs encryption or decryption of data using the DES, T-DES or AES-128 algorithms.

  Important: In order to provide the encryption and decryption services, the CPACF must be “enabled” by installing the Feature Code 3863 (firmware only) on the system.

- KIMD and KLMD: Compute Intermediate Message Digest and Compute Last Message Digest
  These instructions, in the context of the Encryption Facility for z/OS, invoke the SHA-1 algorithm. SHA-1 is involved in the transformation of the optional password into a key.

Crypto Express2 Coprocessor (CEX2C)

The optional Crypto Express2 Coprocessor (CEX2C) comes as a Peripheral Component Interconnect Extended (PCI-X) pluggable feature that provides a high performance and secure cryptographic environment. The following functions of the CEX2C feature are exploited by the Encryption Facility for z/OS:

- Data encryption and decryption with the T-DES algorithm.

  Important:

  - Contrary to the CPACF, the CEX2C performs T-DES encryption and decryption using secure keys only, that is, keys encrypted with the master key.
  - The CEX2C requires the installation of Feature Code 3863 to provide cryptographic services.

- Pseudo Random Number Generation (PRNG)
  This is used for the generation of data encryption keys when the Encryption Facility for z/OS password option has not been chosen.
- RSA asymmetric algorithm support with:
  - RSA key pair generation, with up to 2048-bit long RSA keys
  - Signature generation, with up to 2048-bit long RSA keys
  - Encryption and decryption of symmetric keys with an RSA key up to 2048-bit long

Note that the two first services manage the RSA keys in the PKDS with the ICSF utility functions; strictly speaking, the Encryption Facility for z/OS uses the last service only.

> **Note:** The CEX2C "feature," the book that is plugged into the system, contains two coprocessors or "cards." Beware of this somehow confusing terminology in the IBM documentation: By plugging in a "feature" in the system, you are plugging in two "cards" at once.

The CEX2C feature is designed for FIPS 140-2 Level 4 compliance rating for secure cryptographic hardware modules. This rating ensures that the security-relevant portion of the cryptographic functions is performed inside the secure physical boundary of a tamper-resistant card and that master keys and other security-relevant information, also maintained inside this secure boundary, are subject to "zeroization" in case of tampering attempt detection.

PCIX Cryptographic Coprocessor (PCIXCC)

The PCIXCC is the coprocessor of the CEX2C. It was made available on early z890 and z990 platforms as a feature containing only one coprocessor. It is functionally equivalent to the CEX2C.

The hardware cryptographic facilities in z900 and z800

In this section, we describe the hardware cryptographic devices that are available for these systems.

The Cryptographic Coprocessor Facility (CCF)

The CCF comes as a standard feature of the systems, as one or two coprocessors depending on the system model. A CCF provides, in the context of use by the Encryption Facility for z/OS, the following services:

- Data encryption and decryption with the T-DES algorithm

> **Important:**
>
> - Contrary to the CPACF, the CCF performs T-DES encryption and decryption using secure keys only, that is, keys encrypted with the master key.
> - The CCF requires Feature Code 0875 to be installed in the system to provide cryptographic services. It is important to note that IBM provides firmware "feature codes" that have to be installed to make the cryptographic devices enabled on the system. We explain this in more detail in "Enablement of the cryptographic functions" on page 36.

- Pseudo Random Number Generation (PRNG)
  This is used for the generation of data encryption keys when the Encryption Facility for z/OS password option has not been chosen.
- SHA-1
- RSA asymmetric algorithm support with:
  - Signature generation with up to 1024-bit long RSA key
  - Import and export of DES keys encrypted with an RSA key up to 1024-bit long

  Note that the first service is used for the purpose of managing the RSA keys in the PKDS with ICSF utility functions; strictly speaking, the Encryption Facility for z/OS uses the last service only.

The CCF is designed for FIPS 140-1 Level 4 compliance rating for secure cryptographic hardware modules. This rating ensures that the security-relevant portion of the cryptographic functions is performed inside the secure physical boundary of a tamper-resistant module and that master keys and other security-relevant information, also maintained inside this secure boundary, are subject to “zeroization” in case of tampering attempt detection.

The PCI Cryptographic Coprocessor (PCICC)

The optional PCICC feature comes as a Peripheral Component Interconnect (PCI) pluggable feature that provides a high performance and secure cryptographic environment. The Encryption Facility for z/OS exploits the PCICC function of RSA algorithm support with:

- RSA key pair generation with up to 2048-bit long keys
  Note that this function is not available on the CCF.
- Signature generation with up to 2048-bit long RSA key
  Note that this function is limited to a 1024-bit long key with the CCF.
- Import and export of DES keys encrypted with an RSA key, up to 2048-bit long
  Note that this function is limited to a 1024-bit long key with the CCF.

Note that the two first services manage the RSA keys in the PKDS with ICSF utility functions; strictly speaking, the Encryption Facility for z/OS uses the last service only.

Notes:

- The PCICC "feature," the book that is plugged into the system, contains two coprocessors or "card." Beware of this somehow confusing terminology in the IBM documentation: By plugging in a "feature," you are plugging in two "cards" at once.
- The PCICC feature must be enabled by installing a Function Control Vector (FCV) diskette. See 3.3.3, "Cryptographic coprocessors administration considerations" on page 35.

The PCICC feature is designed for FIPS 140-1 Level 4 compliance rating for secure cryptographic hardware modules. This rating ensures that the security-relevant portion of the cryptographic functions is performed inside the secure physical boundary of a tamper-resistant module and that master keys and other security-relevant information, also maintained inside this secure boundary, are subject to "zeroization" in case of tampering attempt detection.

3.3.2 Application programming interfaces exploited by the Encryption Facility for z/OS

The z/OS integrated cryptographic functions are made available to applications through:

- The ICSF API for all types of coprocessors. The *z/OS Cryptographic Services Integrated Cryptographic Service Facility Application Programmer's Guide*, SA22-7522, describes all the ICSF cryptographic services that can be called by applications. Refer to this book for any detailed information about the ICSF services that we mention in this chapter.
- "System z Assembler instructions for the CPACF" (z9, z990, and z890 only). These are problem state instructions that any application can use and are described in *z/Architecture Principles of Operation*, SA22-7832, as part of the z/Architecture Message-Security Assist (MSA).

Note: What follows can be considered as somehow advanced information about the ICSF services exploited by the Encryption Facility for z/OS. At a higher level, this information can also be used to decide what profiles in the RACF CSFSERV class of resources have to be defined and protected in order to control access to the cryptographic services that the Encryption Facility calls.

Cryptographic services exploited on z9, z990, and z980

The Encryption Facility for z/OS invokes the following services:

- The ICSF callable service CSNBRNG (Random Number Generate) in order to get a high-quality random number for a dynamically generated symmetric key and a corresponding initialization vector.
  The service is provided by a CEX2C/PCIXCC.
- The ICSF callable service CSNBOWH (One Way Hash Generate) is called with the SHA-1 keyword for the Generation of a T-DES or AES-128 clear key derived from the optional password.
  ICSF, in turn, uses the KIMD and KLMD instructions to invoke the CPACF.
- The ICSF callable service CSNBCKM (Multiple Clear Key Import).
  Some of the ICSF callable services that the Encryption Facility for z/OS exploits operate only on T-DES keys already encrypted with the coprocessor's master key. CSNBCKM is a service that takes a clear T-DES data key value and returns it encrypted under the master key.
  The service is provided by the CEX2C/PCIXCC.
- The ICSF callable service CSNDSYX (Symmetric Key Export) to get a T-DES key encrypted with a designated RSA public key.
  The T-DES key is presented to the service already encrypted under the coprocessor master key.
  The service is provided by the CEX2C/PCIXCC.
- The ICSF callable service CSNDSYI (Symmetric Key Import) to likewise decrypt with a designated RSA private key a T-DES key that was encrypted with the corresponding RSA public key.
  The T-DES key is provided by the service encrypted with the coprocessor master key.
  The service is provided by the CEX2C/PCIXCC.
- The ICSF callable service CSNDPKE (PKA Encrypt). This service is called to encrypt a clear AES-128 key with a designated RSA public key.
  The service is provided by the CEX2C/PCIXCC.
- The ICSF callable service CSNDPKD (PKA Decrypt). This service is called to decrypt with a designated RSA private key a clear AES-128 key encrypted with the corresponding RSA public key.
  The service is provided by the CEX2C/PCIXCC.

- The ICSF callable service CSNDSYG (Symmetric Key Generate). This service is used to produce directly inside the coprocessor a T-DES key that comes in two different output formats: One form is the T-DES key encrypted with the coprocessor master key and the other form is the same T-DES key encrypted with a designated RSA public key. This is the service used for the ENCTDES option, because the T-DES key is never seen in clear outside of the coprocessor.
  The service is provided by the CEX2C/PCIXCC.
- The ICSF callable service CSNBENC (Encipher) and CSNBDEC (Decipher) to proceed with the encryption or decryption of data using the ENCTDES option (the T-DES data key is always used encrypted with the coprocessor master key).
  The service is provided by the CEX2C/PCIXCC.
- Encryption and decryption of data using the T-DES algorithm with a clear key. The CPACF is directly called by the Encryption Facility for z/OS using the KMC instruction.
- Encryption and decryption of data using the AES-128 algorithm with a clear key on the System z9 environment.
  The CPACF is directly called by the Encryption Facility for z/OS using the KMC instruction.
- The ICSF callable services CSNBSYE and CSNBSYD (Symmetric Key Encipher and Decipher) to proceed with the encryption and decryption of data using the AES-128 algorithm with a clear key on the z990 and z890environments. These two services are provided as software services only by ICSF.

The following ICSF callable services are solely used for the management of the cryptographic keys in the PKDS, either through the RACF RACDCERT command or the ICSF PKDS Key Management panel (we explain these facilities in Chapter 4, “Planning for the Encryption Facility for z/OS” on page 39). All these services are provided by the CEX2C.

- CSNDPKG (PKA Key Generate): To generate an RSA key pair
- CSNDPKB (PKA Key Token Build): To prepare a formatted key token to receive a new RSA key pair
- CSNDKRC (PKDS Record Create)
- CSNDKRR (PKDS Record Read)
- CSNDKRD (PKDS Record Delete)
- CSNDPKX (PKA Public Key Extract)
- CSNDDSG (Digital Signature Generate)

Cryptographic services exploited on z900 and z800

The Encryption Facility for z/OS invokes the following services:

- The ICSF callable service CSNBRNG (Random Number Generate) in order to get a high quality random number for a dynamically generated symmetric key and a corresponding initialization vector.
  The service is provided by the CCF.
- The ICSF callable service CSNBOWH (One Way Hash Generate) is called with the SHA-1 keyword for the generation of a T-DES or AES-128 clear key derived from the optional password
  The service is provided by the CCF.
- The ICSF callable service CSNBCKM (Multiple Clear Key Import).
  Some of the ICSF callable services that the Encryption Facility for z/OS exploits operate only on T-DES keys already encrypted with the coprocessor's master key. CSNBCKM is a service that takes a clear T-DES data key value and returns it encrypted under the master key.
  The service is provided by the CCF.
- The ICSF callable service CSNDSYX (Symmetric Key Export) to get a T-DES key encrypted with a designated RSA public key.
  The T-DES key is presented to the service already encrypted under the coprocessor master key.
  The service is provided by the CCF or the PCICC.
- The ICSF callable service CSNDSYI (Symmetric Key Import) to likewise decrypt with a designated RSA private key a T-DES key that was encrypted with the corresponding RSA public key.
  The T-DES key is provided by the service encrypted with the coprocessor master key.
  The service is provided by the CCF or the PCICC.
- The ICSF callable service CSNDPKE (PKA Encrypt). This service is called to encrypt a clear AES-128 key with a designated RSA public key.
  The service is provided by the CCF or the PCICC.
- The ICSF callable service CSNDPKD (PKA Decrypt). This service is called to decrypt with a designated RSA private key a clear AES-128 key encrypted with the corresponding RSA public key.
  The service is provided by the CCF or the PCIXCC.

- The ICSF callable service CSNDSYG (Symmetric Key Generate). This service is used to produce directly inside the coprocessor a T-DES key that comes as two different output formats: One form is the T-DES key encrypted with the coprocessor master key and the other form is the same T-DES key encrypted with a designated RSA public key. This is the service used for the ENCTDES option, because the T-DES key is never seen in clear outside of the coprocessor.
  The service is provided by the CCF or PCICC.
- The ICSF callable service CSNBENC (Encipher) and CSNBDEC (Decipher) to proceed with the encryption or decryption of data using either secure T-DES key (ENCTDES option) or clear T-DES key.
  The service is provided by the CCF.
- The ICSF callable services CSNBSYE and CSNBSYD (Symmetric Key Encipher and Decipher) to proceed with the encryption and decryption of data using the AES-128 algorithm with a clear key. These two services are provided as software services only by ICSF.

The following ICSF callable services are solely used for the management of the cryptographic keys in the PKDS, either through the RACF RACDCERT command or the ICSF PKDS Key Management panel (we explain these facilities in Chapter 4, “Planning for the Encryption Facility for z/OS” on page 39). All these services are provided by the CCF, except for the CSNDPKG service, which is only provided by the PCICC.

- CSNDPKG (PKA Key Generate): To generate an RSA key pair. This service is provided by the PCICC only.
- CSNDPKB (PKA Key Token Build): To prepare a formatted key token to receive a new RSA key pair.
- CSNDKRC (PKDS Record Create).
- CSNDKRR (PKDS Record Read).
- CSNDKRD (PKDS Record Delete).
- CSNDPKX (PKA Public Key Extract).
- CSNDDSG (Digital Signature Generate).

3.3.3 Cryptographic coprocessors administration considerations

In this section, still in the context of use of the Encryption Facility for z/OS, we consider that the System z and zSeries coprocessor administration tasks fall in two categories:

- Enabling the cryptographic functions
- Administrating the coprocessors master key

Enablement of the cryptographic functions

National regulations about the exportation of cryptographic devices and their country of use lead the manufacturers of such devices to provide the devices initially disabled, that is, in a state where the device cannot provide any services that involve usage of a secret value. The vendor also provides, under international regulations control, ways of enabling the device at the customer's site.

IBM provides firmware "feature codes" that have to be installed to make the cryptographic devices enabled on the system.

Cryptographic functions enablement on System z9 and zSeries 890 and 990

Enable the cryptographic functions by installing Feature Code 3863. FC 3863 pertains both to the CPACF and CEX2C cryptographic functions.

Enabling the cryptographic functions is not disruptive to the system's operations.

Cryptographic functions enablement on zSeries 900 and 800

The total enablement comes in two steps: A first enablement step has to be performed to enable the CCF coprocessors, and a second step is necessary for the enablement of the PCICC cards (if installed in the system):

- CCF enablement

  Feature Code 0875 must be ordered and received as a firmware diskette to be loaded by the HMC.

  > **Important:** The installation of FC 0875 must be followed by a power-on reset, thus making the first step of cryptographic functions enablement disruptive to the system's operations.

  Note that the installation of FC 0875 is also a prerequisite to the enablement of the PCICC cards.

- PCICC enablement

  The PCICC enablement is achieved by installing the FC 0865 firmware diskette, which must also be ordered to and received from IBM. FC 0865 is also called the PCICC Function Control Vector (FCV), and contrary to FC 0875, its installation is not disruptive to the system's operations.

Installation of the secure coprocessors master key

The secure coprocessors in use on the System z platform are the CCF, PCICC, PCIXCC, and CEX2C.

Note that for the sake of simplicity, we refer to the master key as a single entity in the coprocessor hardware. There are several master keys in each one of these coprocessors: a master key that protects the symmetric keys, that is, keys used by the DES and T-DES algorithms and their derivatives, and a master key to protect the asymmetric keys, that is, the RSA keys. We do not describe in further detail in this book the implementation of the master keys in these coprocessors. For more information, see *z/OS Cryptographic Services Integrated Cryptographic Service Facility Overview*, SA22-7519.

z/OS Cryptographic Services Integrated Cryptographic Service Facility Administrator's Guide, SA22-7521, describes the relevant operations to install and manage master keys, including the PPINIT facility addressed later.

We just want to point out here a useful and easy function to initialize the master keys for the first time in a secure coprocessor: the Pass Phrase Initialization (PPINIT) function. Be aware that Pass Phrase Initialization is intrinsically a non-secure way of establishing a new master key value, because anybody who knows the passphrase knows the master key. However, due to its simplicity, it fits most of the testing environment requirements very well and does not require specific coprocessor administration skills.

3.4 Problem determination and IVP

The main source for problem determination is the SYSPRINT output for the Encryption Facility for z/OS, where messages provide information about the potential cause of a problem. Be aware however that the return and reason codes indicated in the messages might not match the ICSF return and reason codes as specified in *z/OS Cryptographic Services Integrated Cryptographic Service Facility Application Programmer's Guide*, SA22-7522.
These reason and return codes are generated internally to the Encryption Facility and are not documented.

Note: A typical error is to have not started ICSF before invoking the Encryption Facility for z/OS. These are the error messages that the Encryption Services issues in that case:

- If the password option is selected:
  `**ERROR** CSNBOWH  08  000000`
- If the RSA option is selected:
  `**ERROR**CSNBRNG  0C  000000`

Similarly, the DFSMSdss Encryption feature issues:

- `ADR512E (R/I)-RI03 (01), THE ICSF SERVICE CSNBOWH FAILED WITH RETURN CODE 0000000C REASON CODE 00000000`
- or `ADR511E (R/I)-RI03 (01), RSA ENCRYPTION NOT SUPPORTED ON THIS SYSTEM`

Appendix E, “A simple installation verification program” on page 181 provides a sample installation verification program (IVP).

4

Planning for the Encryption Facility for z/OS

This chapter describes and discusses the capabilities of encryption facilities available on System z hardware and software and shows how these can be used with the Encryption Facility.

In this chapter, we provide:

- Some reminders about the IBM Common Cryptographic Architecture (CCA) and its implementation in z/OS
- More detailed explanations about how the Encryption Facility for z/OS operates and its encryption options
- Information about the combinations of valid encryption and decryption methods
- An explanation about how to exploit RSA keys for use with the Encryption Facility for z/OS

4.1 z/OS implementation of IBM CCA overview

In this section, we provide an overview of the implementation of the IBM Common Cryptographic Architecture services in z/OS. We assume that you already have some knowledge about this implementation.

4.1.1 Keys and key labels

ICSF uses T-DES keys held within a 64-byte key token that is structured to contain the key and associated information to control its use.

Asymmetric keys, that is RSA keys, are held within larger tokens (up to 2500 bytes), but are referenced by a 64-byte token label when they are stored in the PKDS.

The Encryption Facility for z/OS calls the ICSF services for data encryption with the symmetric key provided in an application-prepared 64-byte token for a T-DES key; an AES-128 key is provided as a 16-byte clear value.

4.1.2 Key stores: CKDS and PKDS

These two data sets are VSAM data sets and contain the operational keys used within the z/OS environment. All keys held in CKDS and PKDS are encrypted under the master key with the exception of those designated “clear keys.” These keys can be used with the CPACF instructions or by ICSF services that support clear keys.

The CKDS and the PKDS have to go through a semi-automated initialization process before being used to store keys. This initialization process creates an affinity between the data sets and the symmetric and asymmetric keys master keys installed in the secure coprocessor or coprocessors at the time of the initialization, that is, the keys in the data sets are usable only by coprocessors with these master keys set in the relevant domain.

Note: The Encryption Facility for z/OS does not use the CKDS.

4.2 Encryption options for Encryption Facility for z/OS

In this section, we describe the options available to Encryption Facility for z/OS users. Chapter 5, “Encryption Services to Encryption Services or the Decryption Client” on page 51 provides examples of use.

4.2.1 Encryption Facility for z/OS uses

The Encryption Facility encrypts data that is “at rest.” This might mean that backups or archive data sets are encrypted, or it might mean that we want to encrypt data that is being physically transported to another site, or to a client or business partner.

It might also be relevant to encrypt data that is sent electronically over unsecured public networks, or in cases where the recipient is required to take an overt action to recover the clear text.

4.2.2 How Encryption Services works

The Encryption Services consists of two programs:

- CSDFILEN for encrypting data
- CSDFILDE for decrypting data

CSDFILEN overview

CSDFILEN takes the encryptions options, specified by the user, and the DCB information from the initial file (SYSUT1) and places this in a header record in the output data set (SYSUT2). The header record is then followed by the encrypted data. The output data set is structured independently of the input data set, and information needed to reconstitute the original data set is held in the header record.

The header record contains:

- A user-specified description of the file
- The salt and iteration values to be used with the password option
- Length and ICSF labels of the RSA public keys used to encrypt the data key along with an Encryption Facility-generated “RSA tag”
- Initialization chaining vector to use for data encryption
- The type of data key:
  - Clear triple DES key
  - Clear AES-128 key
  - Secure triple DES key
- Input file format:
  - Record format
  - Logical record length
  - Block size

- Output file format:
  - Record format
  - Logical record length
  - Block size
- RSA encrypted data-encrypting key in multiple encrypted form if multiple RSA recipients' keys are specified

CSDFILDE overview

CSDFILDE designates the encrypted file as SYSUT1 and reconstitutes the original data set to SYSUT2. Under some circumstances, the output BLKSIZE can be altered from the original data set, as further explained in the examples in Chapter 5, "Encryption Services to Encryption Services or the Decryption Client" on page 51.

Flow of data through the two programs

Figure 4-1 shows the data flow through the two programs. The intermediate file will be packaged in a format that is independent of the original data set. It contains information in the header that is sufficient to reconstitute the file using CSDFILDE.

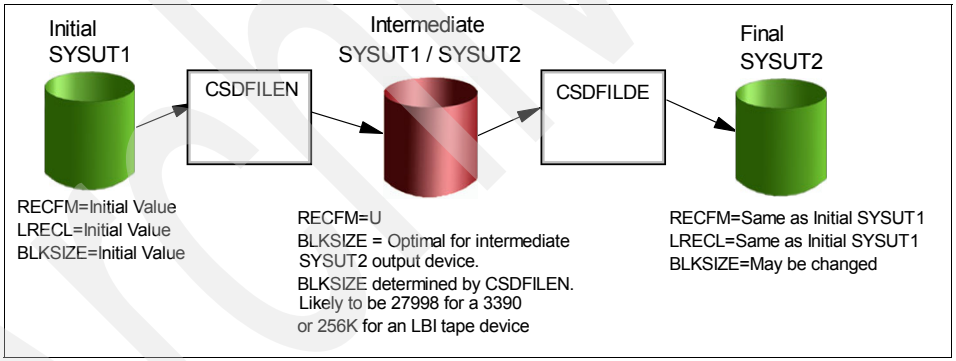

Figure 4-1 Encryption Facility for z/OS programs flow

What Encryption Services options are available

We demonstrate the use of the Encryption Services parameters in far greater detail in Chapter 5, "Encryption Services to Encryption Services or the Decryption Client" on page 51. They include:

- PASSWORD= or RSA=
- ICOUNT=
- DESC=
- CLRTDES or CLRAES128 or ENCTDES
- COMPRESSION=

Some combinations of keywords are invalid. For example, ENCTDES can only be used with RSA, not with PASSWORD.

The decryption program is provided only with the password value or, in some cases, the RSA key label to use. All other information is retrieved from the file header.

The Encryption Services options

There are five valid combinations of keywords for specifying data and data key encryption, as shown in Table 4-1.

Table 4-1 Encryption Services keyword combinations

Method	Description
CLRTDES PASSWORD=	A triple DES key is generated from a password and an iteration count fixed by the user and a random salt value. The triple DES key is used to encrypt the data. The random salt value and the iteration count are written in the header of the file. The password and the iteration count are agreed upon between the sender and receiver. The encryption is performed using clear keys, that is, the triple DES key appears in clear value during the execution of the CSDFILEN program.
CLRTDES RSA=	A random triple DES key is generated and used to protect the data. This triple DES key is encrypted using an RSA public key and is then stored in the file header. The public RSA key belongs to the recipient of the encrypted data and can be between 512 and 2048 bits in length. The RSA private key that corresponds to the public key is known only to the recipient of the file. The data encryption is performed using the clear triple DES key by CSDFILEN.
CLRAES128 PASSWORD=	A 128-bit AES key is generated using a password, a random salt, and an iteration count. The AES key is used to encrypt the data. The random salt value and the iteration count are written to the header of the file. The password and the iteration count must be agreed upon between the sender and receiver. The data encryption is performed using the clear AES-128 key by CSDFILEN.
CLRAES128 RSA=	A random 128-bit AES key is generated and used to encrypt the data. The AES key is then encrypted using an RSA public key and stored in the file header. The RSA public key belongs to the recipient of the encrypted data and can be between 512 and 2048 bits in length. The RSA private key that corresponds to the public key is known only to the recipient of the file. The encryption is performed using the clear AES-128 key.

Method	Description
ENCTDES RSA=	A random triple DES key is generated directly as an ICSF secure key token in storage; that is, the triple DES key is encrypted with the coprocessors symmetric master key. ICSF is called to perform a triple DES encryption using a secure key. The triple DES key is also encrypted using an RSA public key and is stored in the file header. The RSA public key belongs to the recipient of the encrypted data and can be between 512 and 2048 bits in length. The RSA private key that corresponds to the public key is known only to the recipient of the file. The value of the data encryption triple DES key is never exposed in storage during the execution of CSDFILEN. Note: Data encrypted with this method are to be decrypted using ICSF, that is, by the Encryption Services or the Decryption Client features. As for encryption, the triple DES key is used encrypted with the receiving system coprocessors' symmetric master key. Note that the key is kept in its form encrypted with the coprocessor master key the time it takes to encrypt or decrypt the data and is then erased.

4.2.3 Encryption Facility Decryption Client

The Decryption Client options are the same as the CSDFILDE options.

4.2.4 Encryption Facility Java Client

The Java Client provides the same encryption options as the Encryption Services except for the secure triple DES key, which is not supported.

Note however that there is a specific syntax for specifying the encryption options to the Java Client, which we discuss further in Chapter 9, "Using the Encryption Facility Java Client on z/OS" on page 125. Note also that the Java Client does not support data compression.

4.2.5 DFSMSdss DUMP/RESTORE command options

The DFSMSdss DUMP command uses options similar to those proposed by the Encryption Services. We discuss the DFSMSdss DUMP and RESTORE commands in detail in Chapter 8, "The DFSMSdss Encryption feature" on page 103.

4.3 Interoperability between the features of Encryption Facility for z/OS

Table 4-2 shows the valid combinations of interactions between the various features of the Encryption Facility for z/OS.

Note that those combinations marked “Not authorized” refer to the combinations that are not allowed in the Java Client licensing terms and conditions, as mentioned in 1.3, “Client license agreement and additional assistance” on page 5.

Table 4-2 Interoperability matrix

	DFSMSdss Encryption feature	**Encryption Services**	**z/OS Decryption Client**	**Java Client on z/OS**	**Java Client on other (e.g., Microsoft® Windows® XP)**
DFSMSdss Encryption	Possible	Not possible	Not possible	Not possible	Not possible
Encryption Services	Not possible	Possible	Possible	Possible	Possible
Java Client on z/OS	Not possible	Possible	Not authorized	Not authorized	Not authorized
Java Client on other (e.g., Windows XP)	Not possible	Possible	Not authorized	Not authorized	Not authorized

Refer to Appendix A, “Encryption Facility for z/OS: Features compatibility” on page 151 for more details about the interoperability of the Encryption Facility for z/OS features.

4.4 Using a PKDS RSA key to protect the data key

This section discusses the Encryption Services, Decryption Client, and DFSMSdss Encryption features of the Encryption Facility that use the PKDS as a key repository for the RSA public keys of the recipients of the encrypted data.

The Java Client does not support the PKDS as a key store. We explain the use of the RSA protection of the data key by the Java Client in Chapter 9, “Using the Encryption Facility Java Client on z/OS” on page 125.

Most of public keys users today transfer them according to the standardized X.509 V3 digital certificate format. The public key is wrapped in the certificate file with other information that X.509 V3 enabled applications require. The X.509 V3 format is not directly compatible with the RSA key tokens that ICSF keeps in the PKDS, and in z/OS, it takes a manual intervention to extract the public key out of the digital certificate and to insert it in a PKDS record.

In this section, we explain how to use the two facilities available in z/OS, namely the RACDCERT RACF command and the ICSF PKDS Key Management Utility, for setting up RSA keys in the PKDS for use by the Encryption Facility for z/OS.

Important: If RSA keys are not used for securing the encryption keys of data in transit, the password mechanism is used. This is inherently less secure because the password must be coded in the JCL and so can be exposed at either the encryption site or the decryption site or both. The use of RSA keys avoids this exposure because the RSA private key is kept stored in the PKDS of the receiving side encrypted under the coprocessors' asymmetric master key.

4.4.1 X.509 V3 digital certificates

Digital certificates are files that contain, among other information, a public key value and the identity of the owner of the key (referred to as the "certificate subject"). The X.509 V3 standard specifies the format and contents of the digital certificate information fields. A digital certificate is also digitally signed, usually by a certificate authority (referred to as the "certificate issuer") unless the certificate is self-signed, that is, signed by its subject itself.

Note that the trust one can have in a digital certificate (the trust in that the public key inside the certificate actually belongs to the subject of whose identity is also specified in the certificate) is the trust that one can have in the certificate signer (or "issuer"). Using a proper public key infrastructure to distribute certificates provides this level of trust.

Although self-signed certificates are not best to establish this level of trust, they can still be used as a standardized "container" to transport a public key. However, it takes then two individuals to establish an adequately trusted path to communicate the keys this way. This might be the case if the self-signed certificate can be passed physically from one individual to the other on some agreed upon storage medium or transmitted in a way that ensures the veracity of the communicating parties identities.

If you use the Encryption Facility with RSA keys, you will find that X.509 V3 certificates are a very useful method of communicating public keys, although

there might still be situations where one party, or the two parties, involved in the exchange of encrypted data cannot or do not want to use digital certificates.

Data archiving

If the data to be encrypted is for archive purposes, we assume that it will be decrypted using the same system, having access to the same PKDS, as was used to encrypt it. If this is the case, if RSA keys are used, they do not need to be transported. Therefore, the use of certificates is not necessary to set up the RSA key for use by the Encryption Facility.

4.4.2 How do we use certificates?

Let us consider three distinct scenarios with the Encryption Services where RSA protection of the data key is used:

- Data archiving
- Exchange of data between parties who are not currently intending to use digital certificates
- Exchange of data between parties who are using digital certificates

How you choose to use X.509 V3 certificates depends on which of these situations fits your circumstances.

> **Important:** In this section, we assume that APAR OA13030 (ICSF PKDS enhanced key management) has been installed or that your z/OS system has ICSF FMID HCR7731 installed.

Figure 4-2 shows an example of the transfer of an RSA public key using a digital certificate and PKDS key labels to designate the proper RSA key. The certificates are handled by the ICSF PKDS Key Management Utility. Note the following steps:

1. The PKDS Key Management Utility of ICSF generates a certificate that contains the RSA public key of Enterprise B. The public and private keys already reside in the PKDS with Label LABEL_B, where they could have been generated by the utility as well in a prior step.
2. The certificate is sent to Enterprise A, where the public key is extracted and stored in Enterprise A's PKDS, still using the PKDS Key Management Utility, with the label LABEL_AB.
3. Enterprise A encrypts a data set to be sent to Enterprise B, using RSA protection of the data key, and specifies the RSA=LABEL_AB parameter with the CSDFILEN program.

4. Enterprise B receives the encrypted data and executes the CSDFILDE program with the RSA=LABEL_B parameter.

   Note that if Enterprise A and Enterprise B had agreed on using the same PKDS label, the RSA parameter specification would not be needed when executing CSDFILDE.

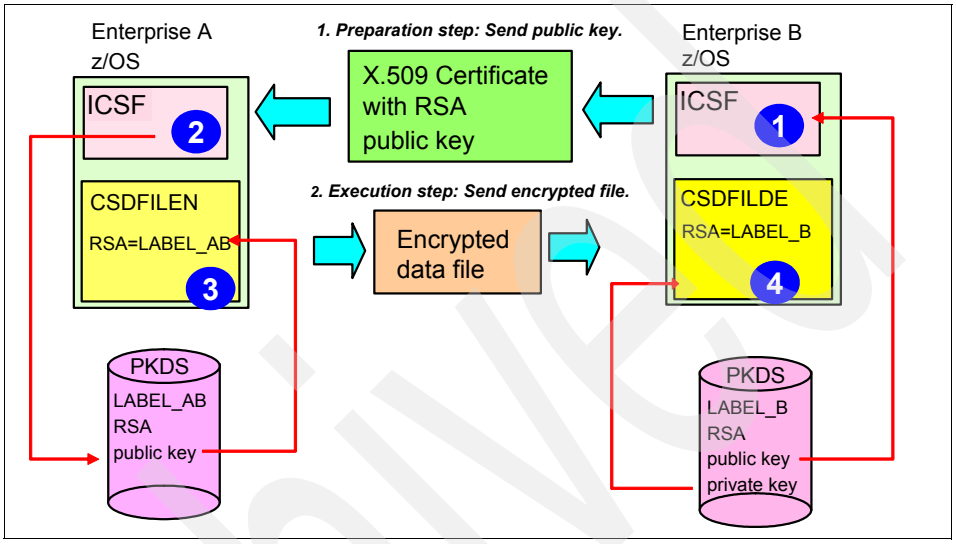

Figure 4-2 Exchanging and using X.509 V3 digital certificates

Let us see what the issues are for the three situations.

Data archiving

As explained earlier, this situation applies when an enterprise uses the Encryption Facility to archive data or store data in some way. The data needs to be protected while it is “at rest,” but when it is used, it is planned that it will be used from the same system that initially backed up the data or an identical system.

Therefore, if the encryption system is the same as the decryption system, we assume that the same PKDS is available. Thus the PKDS label will reference the same public and private key pair at decryption time as was used at encryption time. In this case, there is no need to transport RSA public keys to any other system.

The requirement is simply to have a pair of RSA public and private keys available in the PKDS, with the private key encrypted with the coprocessor master key.

Data transfer without exchange of X.509 V3 certificates

In this situation, we are dealing with two parties that might be within the same organization, or that might be within different organizations or enterprises. The relationship might be supplier and customer or the two might be business partners.

Neither of these two organizations uses X.509 V3 certificates. Therefore, they can consider using a simple method of exchanging RSA public keys[1] that does not involve external trusted authorities.

Data transfer with exchange of X.509 V3 certificates

In this case, the two parties have some experience using X.509 V3 certificates, so they can make good use of the services existing in z/OS to handle X.509 V3 digital certificates.

[1] It might be that the data flows only in one direction. In this case, a single public key needs to flow in the other direction to set up the secure environment.

5

Encryption Services to Encryption Services or the Decryption Client

This chapter describes practical examples of use of the Encryption Services and the Decryption Client.

In this chapter, we discuss the following topics:

- The job control DD statements to use with the Encryption Services and the Decryption Client on z/OS
- The control statement keywords to be specified in SYSIN DD
- Encryption and decryption examples using a password to protect the encryption key
- Encryption and decryption examples using RSA for the encryption key protection
- Encryption and decryption examples using with data compression
- Use of z/OS UNIX System Services pipe examples

5.1 Job control DD statements used for encryption and decryption

The JCL used to invoke the Encryption Services must include the DD cards described in this section.

5.1.1 JCL DD cards to run the encryption program (CSDFILEN)

CSDFILEN uses following DD statements on the encryption step execution.

SYSPRINT

SYSPRINT specifies the name of the data set to which CSDFILEN writes encryption statistics and diagnostic information. It can be a sequential data set, but typically it is a SYSOUT data set, for example, SYSPRINT DD SYSOUT=A.

CSDFILEN sets the following values:

- RECFM=FBA
- LRECL=133

The system selects the al BLKSIZE if not defined. If defined, the BLKSIZE is a multiple of 133.

SYSIN

SYSIN specifies the name of the data set from which CSDFILEN reads control statements that direct encryption processing. It can be a sequential data set such as:

- A spooled system input data set, for example, SYSIN DD *
- A data set with the following characteristics:
  - DASD or tape data set
  - PDS or PSDE member

The following record format is required for the SYSIN data set:

- RECFM=F or FB
- LRECL=80

SYSUT1

SYSUT1 specifies the name of the data set that contains the data to be encrypted. It can be a sequential data such as:

- DASD or tape data set

- PDS or PDSE member
- z/OS UNIX file (HFS or zFS)

The input to CSDFILEN can also be a system input data set, that is, if coded * or DATA on the DD statement. CSDFILEN can read data sets created with BSAM, QSAM, BPAM, or EXCP. The VSAM file must be first unloaded by executing an appropriate utility in order to encrypt it.

For all types of input data sets, consider the following points for specifying record format and DCB information:

- The data set can have any record format (RECFM), although we tested on the following RECFM values: F, FB, V, VB, and U only.

  Important restriction: We could not recover the clear data of a data set with RECFM values VS or VBS. For more details, see 5.6, “Encrypting and decrypting VSAM data sets” on page 74.

- If the data set label (whether on DASD or tape) contains the DCB information, you do not have to code RECFM, LRECL, or BLKSIZE on the DD statement.
- If you plan to decrypt the file into a z/OS UNIX file or to send the encrypted data to a non-z/OS system, RECFM=U must be coded to get the maximum length of records for better efficiency.

If the record format is not available in the data set label and RECFM is not coded on the DD statement, CSDFILEN assumes RECFM=U (undefined format). If the block size is not available from the data set label and BLKSIZE is not coded on the DD statement, CSDFILEN assumes the maximum block size supported by the device. If this block size value is much larger than the sizes of the real blocks, the program might run more slowly than expected.

Note: If the CSDFILEN input data set uses a record format of fixed or variable length and the record length is not available from the data set label and LRECL is not coded on the DD statement, CSDFILEN fails.

CSDFILEN stores the RECFM, LRECL, and BLKSIZE information into the header record of the encrypted file so that they can be used by the decryption program.

If the data set that is the output of CSDFILEN is copied to another z/OS data set, the LRECL or RECFM DCB parameters must *not* be changed.

SYSUT2

SYSUT2 specifies the name of the data set that is to contain the encrypted data. It can be a sequential data set such as:

- DASD or tape data set
- PDS or PDSE member
- z/OS UNIX file (HFS or z/FS)

CSDFILEN forces RECFM=U. The maximum block size can be explicitly set using the BLKSIZE or BLKSZLIM keyword. If not specified, CSDFILEN uses al value for the device type.

To copy the file containing the encrypted data to another device, for instance, a tape data set is specified in SYSUT2 DD but is intended to be later copied to a DASD data set, code BLKSIZE to reflect a block size that is applicable to the ultimate device, for example, 23476 for a DASD data set.

The following list is a summary of default block sizes depending on the kind of specified device for the output data set:

- If the device is a DASD, the default block size is a half track.
- If the device is an IBM 3590 or a later tape device, the default block size is 256 KB.
- If the device is an IBM 3480 or 3490 Tape Subsystem or an IBM Virtual Tape Server, the default block size is 65535.

Important:

- Note that this data set cannot be an ANSI/ISO standard labelled tape or a tape for which OPTCD=Q is specified. In either case, the system requires the data to be character data and performs character conversion, which thereby destroys the encrypted data.
- Do *not* specify DISP=MOD for the SYSUT2 data set on CSDFILEN. An error might be encountered when trying to decrypt the data through CSDFILDE.

Figure 5-1 shows a JCL example we used to run encryption tests. We explain the control statements in 5.2, “Control statement keywords in the SYSIN data set” on page 57.

```
//JOBNMEN  JOB (ACCT,3C1),'PEKKA H',CLASS=A,
//         MSGCLASS=X,MSGLEVEL=(1,1)
//*
//ENRS     EXEC PGM=CSDFILEN
//SYSUT1   DD DSN=PEKKAH.EFCLEAR.INP,DISP=SHR
//SYSUT2   DD DSN=PEKKAH.EFDATA.ENCR.TDES,DISP=(,CATLG),
//         UNIT=SYSDA,SPACE=(1024,(20,5)),AVGREC=K
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
DESC='TEST ENCRYPTED DATA'
CLRTDES
PASSWORD=Day 28072006 was used to test EF
ICOUNT=10
/*
```

Figure 5-1 JCL used to run CSDFILEN tests

5.1.2 JCL DD cards to run the decryption program (CSDFILDE)

CSDFILDE uses the following DD statements in the decryption step.

SYSPRINT

SYSPRINT specifies the name of the data set to which CSDFILDE writes decryption statistics and diagnostic information. It can be a sequential data set, but typically, it is a SYSOUT data set, for example, SYSPRINT DD SYSOUT=A.

CSDFILDE sets the following values:

- RECFM=FBA
- LRECL=133

The system selects the al BLKSIZE if not defined. If defined, the BLKSIZE must be multiple of 133.

SYSIN

SYSIN specifies the name of the data set from which CSDFILDE reads the control statements that direct the decryption process. It can be a sequential data set such as:

- A spooled system input data set, for example, SYSIN DD *

- A data set with the following characteristics:
  - DASD or tape data set
  - PDS or PSDE member

The following record format is required for the SYSIN data set:

- RECFM=F or FB
- LRECL=80

SYSUT1

SYSUT1 specifies the name of the data set that contains the encrypted data to be decrypted. This data set must be the output of the CSDFILEN program or the Encryption Facility for z/OS Client. It can be a sequential data set such as:

- DASD or tape data set
- PDS or PDSE member
- z/OS UNIX file (HFS or z/FS)

If the file that is the output of CSDFILEN is copied to another z/OS file, the LRECL or RECFM DCB parameters must *not* be changed.

SYSUT2

SYSUT2 specifies the name of the output data set that contains the decrypted data. It can be a sequential data set such as:

- DASD or tape data set
- PDS or PDSE member
- z/OS UNIX file (HFS or z/FS)

The output from CSDFILDE can also be a SYSOUT data set (SYSOUT=x).

RECFM and LRECL: There is no need to code RECFM or LRECL for SYSUT2. CSDFILDE assumes that the original values of RECFM and LRECL will be used for the output data set. The RECFM and LRECL values are retrieved from the HEADER record of the encrypted file.

BLKSIZE and BLKSZLIM: There is no need to specify BLKSIZE for SYSUT2. If the value for RECFM (record format) for the original data set whose encrypted output is on SYSUT1 is not an undefined format (RECFM=U) or the BLKSIZE value is not given in the JCL, the system calculates an optimal BLKSIZE value for the device represented by the CSDFILDE output data set on SYSUT2. Although it is possible to specify a BLKSIZE for SYSUT2 to enforce a specific block size value, the specified value might be less than optimal.

If the RECFM value for the original data set whose encrypted output is on SYSUT1 is undefined format (RECFM=U) and the BLKSIZE is not coded on SYSUT2, CSDFILDE makes an assumption for the block size for the original clear data set. Although this value might not be optimal for the device, CSDFILDE does not then attempt to reblock the data.

Figure 5-2 shows a JCL example we used to run the decryption tests. We explain the control statements in 5.2, “Control statement keywords in the SYSIN data set” on page 57.

```
//JOBNMDE  JOB (ACCT,3C1),'PEKKA H',CLASS=A,
//         MSGCLASS=X,MSGLEVEL=(1,1)
//*
//DECR     EXEC PGM=CSDFILDE
//SYSUT1   DD DSN=PEKKAH.EFDATA.ENCR.TDES,DISP=SHR
//SYSUT2   DD DSN=PEKKAH.EFDATA.CLROUT,DISP=(,CATLG),
//            SPACE=(1024,(10,10))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
PASSWORD=Day 28072006 was used to test EF
/*
```

Figure 5-2 JCL used to run CSDFILDE tests

5.2 Control statement keywords in the SYSIN data set

The behavior of the Encryption Services is controlled by the control statements pointed at by the SYSIN DD card of the invoking JCL. In this section, we describe these control statements.

5.2.1 Control statement keywords for encryption (CSDFILEN)

The following options can be specified in the control statement data set (identified in SYSIN DD) to control encryption of the input file. All keywords must start in column 1. Continuation statements cannot be coded. The CSDFILEN program treats an asterisk (*) in column 1 as a comment. If the same keyword is specified multiple times, the program uses the last specification.

Description (optional)

DESC= specifies 1 to 64 EBCDIC characters of descriptive text to be placed into the header record. The information is used to assist in identifying the source of the encrypted data.

Text must be enclosed in single quotation marks. Imbedded blanks are allowed. All text must be included on one control statement line.

Encryption type (optional, default CLRTDES)

This specifies which type of data encryption key the Encryption Services will generate. One of the following types is allowed:

- CLRTDES

  Specifies that the input file is to be encrypted with a clear T-DES key.

- CLRAES128

  Specifies that the input file is to be encrypted with a clear 128-bit AES key.

- ENCTDES

  Specifies that the input file is to be encrypted with a secure T-DES key. This works only with RSA protection of the data encrypting key.

Data key value protection (one keyword is required)

This specifies the method to be used to protect the data encrypting key value. RSA and PASSWORD are mutually exclusive.

- RSA=

  Specifies the label of an existing RSA public key token in the ICSF PKDS, or the label of an RSA private key token in the PKDS that contains both a public and private key. The CSDFILEN program uses the RSA public key to encrypt the data-encrypting key. The corresponding RSA private key must be present at the recipient's site when the encrypted data is decrypted.

  The encrypted data set or file can be sent to up to 16 different recipients, each one of them getting a copy of the data encrypting key protected by his/her own RSA public key. This is achieved by specifying up to 16 RSA= keywords, one for each recipient's RSA key label. There can be, therefore, up to 16 RSA encrypted data keys in the file header.

  For more details about how to specify multiple RSA key labels, see 5.4, "Encryption and decryption using RSA protection of data-encrypting key" on page 66.

- PASSWORD=

  Specifies a 8-to-32 character password to be used to derive a clear T-DES key or a clear 128-bit AES key. Leading and trailing blanks and tab characters are removed. Imbedded blanks and tab characters are allowed.

  In order to minimize problems because of code page differences at the encrypting and decrypting sites, we suggest using only the uppercase and lowercase letters A through Z, numerals 0 – 9, and the underscore character (_).

Note: Passwords are case-sensitive.

Iteration count (optional, default 16)

When the PASSWORD keyword is specified, ICOUNT= specifies the number of iterations of the SHA-1 hash algorithm to be run to derive a data key value and the initial chaining vector (ICV) value to be used for encryption. ICOUNT has an integer value between 1 and 1000.

Compression option (optional, default NO)

COMPRESSION= specifies whether the clear input data is to be compressed before encryption of the data. If compression is selected, it is always performed before encryption. Valid values are YES or NO.

Examples of control statement keywords used for encryption

The first example, shown in Example 5-1, is a set of control statements to compress and then encrypt an input data set using a random AES 128-bit key. The key value is derived from a password.

Example 5-1 File compression and encryption using AES with 128-bit key

```
DESC='TEST ENCRYPTED BIG FILE FOR EF RESIDENCY (COMPRESSED)'
CLRAES128
PASSWORD=03082006 is THE Password
ICOUNT=10
COMPRESSION=YES
```

The second example, shown in Example 5-2, is a set of control statements to generate a random secure key to encrypt the data. This key is then protected using two different RSA public keys.

Example 5-2 File encryption key protected using two different RSA public keys

```
DESC='Lennies test case with rsa keys'
ENCTDES
RSA=LENNIE.TEST.RSA1024
RSA=LENNIE.TEST.RSA1024.COPY
```

5.2.2 Control statement keywords for decryption (CSDFILDE)

The following options can be specified in the control statement data set (identified by SYSIN DD) to control decryption of the input files. All keywords

must start in column 1. A continuation statement cannot be coded. The CSDFILDE program treats an asterisk (*) in column 1 as a comment.

All control statement input for CSDFILDE is optional, unless the following conditions apply:

- PASSWORD= was specified as input to CSDFILEN. In this case, the correct password must be supplied on the SYSIN control statement.
- The label of the RSA private key to be used to decrypt the data-encrypting key is different from the label of the RSA public key used by CSDFILEN. In this case, the correct label of the RSA private key must be supplied on the control statement for CSDFILDE.
- Multiple RSA keys were used at encryption time to protect the data encrypting key. Then, there must be one RSA keyword to point at the table of the specific RSA key to use for decryption.

Data key value protection (default RSA)

This specifies the method used to protect the data encrypting key value. RSA and PASSWORD, when specified, are mutually exclusive.

- RSA=

  Specifies the 64-byte *label* of an RSA private key that is in the ICSF PKDS. This is an RSA private key that corresponds to one of the public keys used to encrypt the data-encrypting key that is in the header record of the encrypted file.

  If only one key was used to encrypt the file, and no RSA keyword is specified in the CSDFILDE control statements, the RSA label to be used is the one specified in the encrypted file header. Specifying the RSA= keyword allows you to point to a label different from the one recorded at encryption time.

  If multiple RSA keywords have been specified for CSDFILEN, one RSA= keyword must be specified in CSDFILDE.

  CSDFILDE does not allow multiple RSA keywords. For more details about how to specify the RSA key label, see 5.4, “Encryption and decryption using RSA protection of data-encrypting key” on page 66.

- PASSWORD=

  Specifies the 8-to-32 character long password to be used to derive the clear T-DES key or the clear 128-bit AES key from, and to use this derived key to decrypt the data. This password *must* match the password used to encrypt the data.

  Note that the ICOUNT is retrieved from the encrypted file header.

  In order to minimize problems that can be induced by code page differences at the encrypting and decrypting sites, we suggest using only the uppercase

and lowercase letters A through Z, numerals 0 – 9, and the underscore character (_) in a password.

Note: Passwords are case-sensitive.

- INFO (optional)

  Specifies that file decryption is not to be performed, but CSDFILDE is to retrieve information from the encrypted file header, such as the DESC, and write them along with any parameters default values it is going to use in the SYSPRINT.

  This option is useful to determine, if needed, such information as the original clear text file DCB information, or to ensure that a specified RSA key label is present in the current PKDS.

Examples of control statement keywords used for decryption

Example 5-3 shows a set of control statements for decryption to determine the DCB information the original file. The encrypted file is not decrypted, only the file information is printed on the report.

Example 5-3 Control statements to determine the DCB information

```
PASSWORD=03082006
INFO
```

In the second example, shown in Example 5-4, the control statement indicates which RSA key label CSDFILDE has to use when recovering the data encryption key.

Example 5-4 Decryption using RSA key to protect the encryption key

```
RSA=LENNIE.TEST.RSA1024
```

5.3 Encryption and decryption report examples with the password option

Both encryption (CSDFILEN) and decryption (CSDFILDE) produces a statistics report file that contains information about the control statements used in the CSDFILEN or CSDFILDE batch jobs, the name of the data set or input file that is to be encrypted or decrypted, and performance statistics about the execution of the job.

Note: Save the statistics report issued during encryption, so it can be used to verify that the correct amount of data is recovered in the decryption and optional decompression processes.

We provide examples of statistics reports in 5.4, “Encryption and decryption using RSA protection of data-encrypting key” on page 66 and 5.5, “Encryption and decryption using data compression” on page 71.

Here we provide examples of statistics reports produced using the JCL that we described earlier in this chapter. There were two input files for the encryption trials: a small one that has only a few hundred records and a larger one with several hundreds of thousands records. The contents of the files are, in both cases, text strings.

5.3.1 Contents of the statistics report file for encryption

In the first example, shown in Figure 5-3 on page 63, a small file has been encrypted using the CLRTDES keyword on a z900.

The password 1 will never be shown in clear on the report. The encryption is done using the CCF processors 2, as shown in the report. The Encryption Services uses 27998 3 as a length for the output BLKSIZE on disks. In this test case, the amount of the output data, 19992 4 bytes, written to the file also contains the header record information and any padding required for the encryption process. The report also shows the throughput rates along with the longest and shortest cipher times and the data length associated with them are shown at 5.

Note: The T-DES algorithm requires the input data length to be a multiple of 8 bytes. The AES-128 algorithm requires the input data length to be a multiple of 16 bytes.

```
 CSDFILEN Encryption Utility 07/28/2006 (MM/DD/YYYY) 21:48:45 (HH:MM:SS)
INPUT:  DESC='TEST ENCRYPTED DATA'
INPUT:  CLRTDES
INPUT:  PASSWORD=******************************** 1
INPUT:  ICOUNT=10
 CSDFILEN:           :
INPUT:  LRECL    80 BLKSIZE    6160 RECFM FB
OUTPUT: BLKSIZE   27998 3
ENCRYPTION OF DATA: CLEAR     TDES KEY USING CCF 2
  RECORDS READ:           227  WRITTEN:            1
  BYTES READ:             18,160 6
  BYTES WRITTEN:          19,992 4 WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): 5 HIGH: 0.000754 DATA: 19528 LOW: 0.000754 DATA: 19528
 TOTAL CIPHER TIME (IN SECONDS):    0.000754  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.11
```

Figure 5-3 Statistics report with CLRTDES and PASSWORD encryption on z900

In the next example, shown in Figure 5-4, the same small file has been transferred from z900 to z9 to carry on more tests, and the BLKSIZE was also changed to better fit our requirements. The file was encrypted using CLRTDES keyword on the z9 mainframe. This time, the encryption uses the CPACF processors 1, as shown in the report. As expected, the reported cipher time for the block of data is shorter with the CPACF on z9 2 than with CCF on z900.

```
 CSDFILEN Encryption Utility 08/03/2006 (MM/DD/YYYY) 11:06:03 (HH:MM:SS)
INPUT:  DESC='TEST ENCRYPTED DATA'
INPUT:  CLRTDES
INPUT:  PASSWORD=********************************
INPUT:  ICOUNT=10
 CSDFILEN:           :
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR     TDES KEY USING CPACF 1
  RECORDS READ:           227  WRITTEN:            1
  BYTES READ:           18,160
  BYTES WRITTEN:          19,992  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): 2 HIGH: 0.000089 DATA: 19528 LOW: 0.000089 DATA: 19528
 TOTAL CIPHER TIME (IN SECONDS):    0.000089  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure 5-4 Statistics report with CLRTDES and PASSWORD encryption on z9

In the next example, shown in Figure 5-5, the same small file has been encrypted using the CLRAES128 1 keyword on a z9 platform. The encryption again uses the CPACF. This time, the amount of the output data, 20000 2 bytes, written to the file will be bigger because of the longer padding needed for the AES-128 algorithm.

```
 CSDFILEN Encryption Utility 08/03/2006 (MM/DD/YYYY) 11:23:18 (HH:MM:SS)
INPUT:  DESC='TEST ENCRYPTED DATA'
INPUT:  CLRAES128 1
INPUT:  PASSWORD=********************************
INPUT:  ICOUNT=10
 CSDFILEN:           :
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR    AES  KEY USING CPACF
  RECORDS READ:           227  WRITTEN:            1
  BYTES READ:           18,160
  BYTES WRITTEN:          20,000 2 WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): 3 HIGH: 0.000069 DATA: 19536 LOW: 0.000069 DATA: 19536
 TOTAL CIPHER TIME (IN SECONDS):    0.000069  CIPHERS:          1
  TOTAL ELAPSED TIME:   0:00:00.01
```

Figure 5-5 Statistics report with CLRAES128 and PASSWORD encryption on z9

5.3.2 Contents of the statistics report file for decryption

A statistics report issued by the decryption process is shown in Figure 5-6 on page 65. 1 is the description information provided at encryption time. The original input data set RECFM, LRECL, and BLKSIZE 2 are also displayed in the report. It is expected that the description data give proper information so that the original data set or file can be easily identified.

> **Note:** It is important to verify that the decryption process has recovered all the data that was encrypted. This can be done by comparing the bytes read value 6 in Figure 5-3 and the recovered bytes value 3 in Figure 5-6.

To decrypt the encrypted file, the correct password must be given as an input keyword. The given password 4 is not shown in the report. If the password is not the same as the one used at encryption time, an error message “INCORRECT PASSWORD ENTERED” is displayed in the report.

If the BLKSIZE value assigned to the decryption process output data set is different from the BLKSIZE of the encryption input data set, for a RECFM=FB, a warning message 5 is issued and processing continues.

```
 CSDFILDE Decryption Utility 07/28/2006 (MM/DD/YYYY) 21:48:45 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      1
 CSDFILDE:           :
INPUT: DESC = TEST ENCRYPTED DATA 1
INPUT: LRECL 80 BLKSIZE    6160 RECFM FB 2
INPUT:  PASSWORD=******************************** 4
**WARNING** NEW OUTPUT BLKSIZE. REQUESTED: 27920 5
RECORDS READ:             1  WRITTEN:           227
  BYTES READ:            20,032  BYTES RECOVERED:           18,160 3
CIPHER TIMES (IN SECONDS): HIGH: 0.000979 DATA: 27536 LOW: 0.000979 DATA: 27536
 TOTAL CIPHER TIME (IN SECONDS):    0.000979  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.12
```

Figure 5-6 Statistics report for PASSWORD and CLRTDES decryption on z900

In the second example, shown in Figure 5-7, the same small encrypted file is decrypted on the z9 platform. Here, the BLKSIZE specified in the CSDFILDE JCL is the same as the original clear data BKSIZE data set.

```
 CSDFILDE Decryption Utility 08/03/2006 (MM/DD/YYYY) 11:06:03 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      1
 CSDFILDE:           :
INPUT: DESC = TEST ENCRYPTED DATA
INPUT: LRECL     80 BLKSIZE   27920 RECFM FB 1
INPUT:  PASSWORD=******************************
RECORDS READ:             1  WRITTEN:           227
  BYTES READ:            20,078  BYTES RECOVERED:           18,160
CIPHER TIMES (IN SECONDS): HIGH: 0.000123 DATA: 27536 LOW: 0.000123 DATA: 27536
 TOTAL CIPHER TIME (IN SECONDS):    0.000123  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure 5-7 Statistics report with PASSWORD and CLRTDES decryption on z9

In the third decryption report example, shown in Figure 5-8, the original file was encrypted using CLRAES128. CSDFILDE retrieves this information from the encrypted file header record.

```
 CSDFILDE Decryption Utility 08/03/2006 (MM/DD/YYYY) 11:23:18 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      1
 CSDFILDE:           :
INPUT: DESC = TEST ENCRYPTED DATA
INPUT: LRECL     80 BLKSIZE   27920 RECFM FB
INPUT:  PASSWORD=******************************
RECORDS READ:             1  WRITTEN:           227
  BYTES READ:            20,000  BYTES RECOVERED:           18,160
CIPHER TIMES (IN SECONDS): HIGH: 0.000090 DATA: 27536 LOW: 0.000090 DATA: 27536
 TOTAL CIPHER TIME (IN SECONDS):    0.000090  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure 5-8 Statistics report with PASSWORD and CLRAES128 decryption on z9

5.4 Encryption and decryption using RSA protection of data-encrypting key

When the PASSWORD option is not used, the dynamically generated data encryption key is protected by RSA encryption. In this section, we provide examples of the use of RSA to protect the data key.

5.4.1 Contents of the statistics report for encryption and decryption

Several reports are shown in this section, demonstrating a selection of options for encryption and decryption using RSA key protection of the data-encrypting key. The following figures show the statistics report for encryption and decryption in the following cases:

- System z9, ENCTDES, RSA with 2048-bit key. See Figure 5-9 and Figure 5-10 on page 67.
- System z9, CLRAES128, RSA with 2048-bit key. See Figure 5-11 on page 67 and Figure 5-12 on page 67.
- zSeries 900, ENCTDES, RSA with 1024-bit key. See Figure 5-13 on page 68 and Figure 5-14 on page 68.
- zSeries 900, CLRDES, RSA with 1024-bit key. See Figure 5-15 on page 68 and Figure 5-16 on page 69.

```
 CSDFILEN Encryption Utility 08/08/2006 (MM/DD/YYYY) 10:29:22 (HH:MM:SS)
INPUT:  DESC='Lennies test case with rsa keys'
INPUT:  ENCTDES
INPUT:  RSA=ITSOLD.TEST.RSA2048
 CSDFILEN:   RSA-PUB : ITSOLD.TEST.RSA2048
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: ENCRYPTED TDES KEY USING CRYPTO COPROCESSOR
  RECORDS READ:          231  WRITTEN:            1
  BYTES READ:          18,480
  BYTES WRITTEN:          20,336  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000235 DATA:    19872 LOW:    0.000235
DATA:    19872
 TOTAL CIPHER TIME (IN SECONDS):    0.000235  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure 5-9 Encryption with RSA (2048) and ENCTDES on z9

```
 CSDFILDE Decryption Utility 08/08/2006 (MM/DD/YYYY) 10:29:22 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      1
 CSDFILDE:    RSA-PUB : ITSOLD.TEST.RSA2048
INPUT: DESC = Lennies test case with rsa keys
INPUT: LRECL     80 BLKSIZE   27920 RECFM FB
INPUT:  RSA=ITSOLD.TEST.RSA2048
RECORDS READ:             1  WRITTEN:           231
  BYTES READ:           20,410  BYTES RECOVERED:           18,480
CIPHER TIMES (IN SECONDS): HIGH:   0.000287 DATA:   27536 LOW:   0.000287 DATA:
27536
 TOTAL CIPHER TIME (IN SECONDS):     0.000287  CIPHERS:            1
  TOTAL ELAPSED TIME:   0:00:00.03
```

Figure 5-10 Decryption with RSA (2048) and ENCTDES on z9

```
 CSDFILEN Encryption Utility 08/08/2006 (MM/DD/YYYY) 10:29:22 (HH:MM:SS)
INPUT:  DESC='Lennies test case with rsa keys'
INPUT:  CLRAES128
INPUT:  RSA=ITSOLD.TEST.RSA2048
 CSDFILEN:    RSA-PUB : ITSOLD.TEST.RSA2048
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR     AES  KEY USING CPACF
  RECORDS READ:           231  WRITTEN:           1
  BYTES READ:           18,480
  BYTES WRITTEN:           20,336  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000071 DATA:    19872 LOW:    0.000071
DATA:    19872
 TOTAL CIPHER TIME (IN SECONDS):    0.000071  CIPHERS:            1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure 5-11 Encryption with RSA (2048) and CLRAES128 on z9

```
 CSDFILDE Decryption Utility 08/08/2006 (MM/DD/YYYY) 10:29:23 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      1
 CSDFILDE:    RSA-PUB : ITSOLD.TEST.RSA2048
INPUT: DESC = Lennies test case with rsa keys
INPUT: LRECL     80 BLKSIZE   27920 RECFM FB
INPUT:  RSA=ITSOLD.TEST.RSA2048
RECORDS READ:             1  WRITTEN:           231
  BYTES READ:           20,390  BYTES RECOVERED:           18,480
CIPHER TIMES (IN SECONDS): HIGH:   0.000091 DATA:   27536 LOW:   0.000091 DATA:
27536
 TOTAL CIPHER TIME (IN SECONDS):     0.000091  CIPHERS:            1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure 5-12 Decryption with RSA (2048) and CLRAES128 on z9

```
 CSDFILEN Encryption Utility 08/08/2006 (MM/DD/YYYY) 18:04:16 (HH:MM:SS)
INPUT:  DESC='Lennies test case with rsa keys'
INPUT:  ENCTDES
INPUT:  RSA=LENNIE.TEST.RSA1024
 CSDFILEN:   RSA-PUB : LENNIE.TEST.RSA1024
INPUT:  LRECL   256 BLKSIZE    6233 RECFM VB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: ENCRYPTED TDES KEY USING CRYPTO COPROCESSOR
  RECORDS READ:            74  WRITTEN:            1
  BYTES READ:           18,776
  BYTES WRITTEN:           19,400  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000794 DATA:    18936 LOW:    0.000794
DATA:    18936
 TOTAL CIPHER TIME (IN SECONDS):    0.000794  CIPHERS:          1
  TOTAL ELAPSED TIME:   0:00:00.27
```

Figure 5-13 Encryption with RSA (1024) and ENCTDES on z900

```
 CSDFILDE Decryption Utility 08/08/2006 (MM/DD/YYYY) 18:04:16 (HH:MM:SS)
 CSDFILDE:   RSA-PUB : LENNIE.TEST.RSA1024
INPUT: DESC = Lennies test case with rsa keys
INPUT: LRECL    256 BLKSIZE    6233 RECFM VB
INPUT:  RSA=LENNIE.TEST.RSA1024
RECORDS READ:              1  WRITTEN:            74
  BYTES READ:           19,400  BYTES RECOVERED:           18,776
CIPHER TIMES (IN SECONDS): HIGH:   0.001102 DATA:   27536 LOW:   0.001102 DATA:
27536
 TOTAL CIPHER TIME (IN SECONDS):    0.001102  CIPHERS:            1
  TOTAL ELAPSED TIME:   0:00:00.11
```

Figure 5-14 Decryption with RSA (1024) and ENCTDES on z900

```
 CSDFILEN Encryption Utility 08/08/2006 (MM/DD/YYYY) 18:04:16 (HH:MM:SS)
INPUT:  DESC='Lennies test case with rsa keys'
INPUT:  CLRTDES
INPUT:  RSA=LENNIE.TEST.RSA1024
 CSDFILEN:   RSA-PUB : LENNIE.TEST.RSA1024
INPUT:  LRECL   256 BLKSIZE    6233 RECFM VB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR    TDES KEY USING CCF
  RECORDS READ:            74  WRITTEN:            1
  BYTES READ:           18,776
  BYTES WRITTEN:           19,400  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000846 DATA:    18936 LOW:    0.000846
DATA:    18936
 TOTAL CIPHER TIME (IN SECONDS):    0.000846  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.14
```

Figure 5-15 Encryption with RSA (1024) and CLRTDES on z900

```
 CSDFILDE Decryption Utility 08/08/2006 (MM/DD/YYYY) 18:04:17 (HH:MM:SS)
 CSDFILDE:   RSA-PUB : LENNIE.TEST.RSA1024
INPUT: DESC = Lennies test case with rsa keys
INPUT: LRECL   256 BLKSIZE   6233 RECFM VB
INPUT:  RSA=LENNIE.TEST.RSA1024
RECORDS READ:            1  WRITTEN:           74
  BYTES READ:          19,400  BYTES RECOVERED:          18,776
CIPHER TIMES (IN SECONDS): HIGH:   0.001093 DATA:   27536 LOW:   0.001093 DATA:
27536
 TOTAL CIPHER TIME (IN SECONDS):    0.001093  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.09
```

Figure 5-16 Decryption with RSA (1024) and CLRTDES on z900

5.4.2 Multiple RSA key support

With the use of RSA keys for the protection of the data-encrypting key comes another facility. It is possible to specify up to 16 RSA keys to be used when the data is encrypted; that is, the same encrypted data can be sent to up to 16 different recipients. In this case, the data is still encrypted under the same data-encrypting key, but multiple copies of the data-encrypting key are encrypted, each one with one of the RSA public keys.

This increases the length of the header block, which now contains the multiple encrypted versions of the data key.

If more than one RSA key is specified when the data is encrypted, it is mandatory to specify one RSA key label at decryption time.

Table 5-1 shows the available options.

Table 5-1 Specification of the RSA key parameter

	CSDFILDE - Decrypt No RSA key specified	**CSDFILDE - Decrypt 1 RSA key specified**
CSDFILEN Encrypt 1 RSA key label specified	Uses the label used at encryption time (taken from header record) to reference the PKDS private key to use.	Uses the label specified at decryption time to reference the PKDS private key to use.
CSDFILDE Decrypt > 1 RSA key label specified at encryption	Fails with message: “**ERROR** EITHER THE PASSWORD= OR RSA= IS REQUIRED”	Uses the label specified at decryption time to reference the PKDS private key to use. Uses a specific logic to determine the header record entry to use. See second note.

	CSDFILDE - Decrypt No RSA key specified	CSDFILDE - Decrypt 1 RSA key specified

Notes

- Multiple RSA keys specified at decryption time results in the message ""**ERROR** SPECIFY ONE OF RSA/PASSWORD ONLY".
- When multiple RSA keys are specified for CSDFILEN, a table is built in the header record showing all the RSA public keys labels that have been used to encrypt the data key. Each entry in the table also contains the data encrypting key encrypted with the designated RSA public key.
  A tag is also stored in each entry of the table. It consists of a known string encrypted by the entry's RSA public key. At decryption time, CSDFILDE uses the public key specified by the RSA parameter to encrypt the same known string and compares it with each tag in the table. A match indicates which encrypted data encrypting key entry has to go under decryption using the corresponding RSA private key.

In Figure 5-17 and in Figure 5-18 on page 71 you can see the resulting statistics report. Note that the key label used in CSDFILDE does not exactly match the name of any key label used during the CSDFILEN execution. However, CSDFILDE has worked out which version of the RSA encrypted data key to use.

```
 CSDFILEN Encryption Utility 08/08/2006 (MM/DD/YYYY) 13:04:30 (HH:MM:SS)
INPUT:  DESC='Lennies test case with rsa keys'
INPUT:  ENCTDES
INPUT:  RSA=ITSOLD.TEST.RSA2048
INPUT:  RSA=ITSOLD.TEST.RSA2048.COPY1
INPUT:  RSA=ITSOLD.TEST.RSA2048.COPY2
 CSDFILEN:   RSA-PUB : ITSOLD.TEST.RSA2048
 CSDFILEN:   RSA-PUB : ITSOLD.TEST.RSA2048.COPY1
 CSDFILEN:   RSA-PUB : ITSOLD.TEST.RSA2048.COPY2
INPUT:  LRECL   80 BLKSIZE  27920 RECFM FB
OUTPUT: BLKSIZE  27998
ENCRYPTION OF DATA: ENCRYPTED TDES KEY USING CRYPTO COPROCESSOR
  RECORDS READ:          231  WRITTEN:           1
  BYTES READ:          18,480
  BYTES WRITTEN:         21,372  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000243 DATA:    19872 LOW:    0.000243
DATA:    19872
 TOTAL CIPHER TIME (IN SECONDS):    0.000243  CIPHERS:          1
  TOTAL ELAPSED TIME:   0:00:00.42
```

Figure 5-17 Encryption with RSA (2048) and ENCTDES, multiple RSA keys on z9

```
 CSDFILDE Decryption Utility 08/08/2006 (MM/DD/YYYY) 13:04:30 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      2
 CSDFILDE:    RSA-PUB : ITSOLD.TEST.RSA2048
 CSDFILDE:    RSA-PUB : ITSOLD.TEST.RSA2048.COPY1
 CSDFILDE:    RSA-PUB : ITSOLD.TEST.RSA2048.COPY2
INPUT: DESC = Lennies test case with rsa keys
INPUT: LRECL     80 BLKSIZE   27920 RECFM FB
INPUT:  RSA=ITSOLD.TEST.RSA2048.COPY
RECORDS READ:             1  WRITTEN:           231
  BYTES READ:          21,412  BYTES RECOVERED:          18,480
CIPHER TIMES (IN SECONDS): HIGH:   0.000279 DATA:   26504 LOW:   0.000279 DATA:
26504
 TOTAL CIPHER TIME (IN SECONDS):    0.000279  CIPHERS:           1
  TOTAL ELAPSED TIME:   0:00:00.03
```

Figure 5-18 Decryption with RSA (2048) and ENCTDES on z9

5.5 Encryption and decryption using data compression

When the data compression is requested with encryption using the COMPRESS=YES control keyword, the CSDFILEN program uses hardware compression before encrypting the data file. At decryption, the file is first decrypted and then decompressed.

> **Important:** CSDFILEN will not attempt to compress a file that is less than 64 KB.
>
> The original data pattern might not lend itself to an appropriate compression ratio. In that case, the Encryption Services internal program logic detects this condition and does not proceed with data compression.

Both encryption and decryption produce a statistics report file, as described in 5.3, “Encryption and decryption report examples with the password option” on page 61. The report also contains the compression statistics.

5.5.1 Statistics report file for encryption with compression

In the first example, shown in Figure 5-19 on page 72, a file has been first compressed and then encrypted using the CLRTDES keyword on a z900 platform.

In this example, the original file was written text, which produces a good compression result. The original file size of more than 50 MB has been compressed and encrypted for a resulting size of 4.5 MB, as shown in **1**. The compression dictionary is stored into the output file and is also encrypted along

with the original data. The report also shows the CPU time used for the compression 2.

```
 CSDFILEN Encryption Utility 08/03/2006 (MM/DD/YYYY) 20:40:48 (HH:MM:SS)
INPUT:  DESC='TEST ENCRYPTED BIG FILE FOR EF RESIDENCY (COMPRESSED)'
INPUT:  CLRTDES
INPUT:  PASSWORD=********************************
INPUT:  ICOUNT=10
INPUT:  COMPRESSION=YES
 CSDFILEN:           :
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR    TDES KEY USING CCF
  RECORDS READ:       661,932  WRITTEN:          161
  BYTES READ:       52,954,560
  BYTES WRITTEN: 1 4,497,880  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH: 0.003001 DATA: 89272 LOW: 0.000534 DATA: 14816
 TOTAL CIPHER TIME (IN SECONDS):    0.154812  CIPHERS:          186
  TOTAL COMPRESS TIME (IN SECONDS): 1.112872 TOTAL 2 COMPRESS BYTES: 4,431,125
   TOTAL ELAPSED TIME:   0:00:08.40
```

Figure 5-19 File compression and encryption on z900

In the second example, shown in Figure 5-20, the same file has been compressed and then encrypted using the CLRTDES keyword on a z9 platform. The report shows similar results as in the example earlier. The only difference that pertains to the performance of the compression 1 and encryption.

```
 CSDFILEN Encryption Utility 08/03/2006 (MM/DD/YYYY) 13:50:29 (HH:MM:SS)
INPUT:  DESC='TEST ENCRYPTED BIG FILE FOR EF RESIDENCY, COMPRESSED (Z9)'
INPUT:  CLRTDES
INPUT:  PASSWORD=********************************
INPUT:  ICOUNT=10
INPUT:  COMPRESSION=YES
 CSDFILEN:           :
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR    TDES KEY USING CPACF
  RECORDS READ:       661,932  WRITTEN:          161
   BYTES READ:        52,954,560
  BYTES WRITTEN:       4,497,880  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000376 DATA:    89272 LOW:    0.000060
DATA:    14816
 TOTAL CIPHER TIME (IN SECONDS):    0.018396 CIPHERS:          186
  TOTAL COMPRESS TIME (IN SECONDS): 0.456056 1 TOTAL COMPRESS BYTES: 4,431,125
   TOTAL ELAPSED TIME:   0:00:04.44
```

Figure 5-20 File compression and encryption on z9

In the third example, shown in Figure 5-21, the input file contains data that cannot be compressed with an appropriate compression ratio. The message in the report 1 indicates that the compression is not done and CSDFILEN proceeds with the encryption of the original data.

```
 CSDFILEN Encryption Utility 08/01/2006 (MM/DD/YYYY) 14:37:51 (HH:MM:SS)
INPUT:  DESC='TEST ENCRYPTED DATA FOR ENCRYPTION FACILITY RESIDENCY'
INPUT:  CLRTDES
INPUT:  PASSWORD=********************************
INPUT:  ICOUNT=5
INPUT:  COMPRESSION=YES
 CSDFILEN:          :
INPUT:  LRECL 27998 BLKSIZE   27998 RECFM U
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR    TDES KEY USING CPACF
**WARNING** MINOR ERROR OCCURRED BUILDING THE COMPRESSION DICTIONARY. 1
**WARNING** ENCRYPTION CONTINUING WITHOUT COMPRESSION.
  RECORDS READ:        2,034  WRITTEN:        2,035
  BYTES READ:       56,947,932
  BYTES WRITTEN:       56,960,608  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.001190 DATA:   280040 LOW:    0.000456
DATA:   112024
 TOTAL CIPHER TIME (IN SECONDS):    0.232829  CIPHERS:         204
  TOTAL ELAPSED TIME:   0:00:08.78
```

Figure 5-21 File compression failing

5.5.2 Statistics report for decryption with decompression

In the decryption report example, Figure 5-22, the compressed and encrypted file is first decrypted and then decompressed. The most important verification to do pertains to the number of the bytes recovered 1, which must match the number of bytes read at encryption time, 2 in Figure 5-20 on page 72.

```
 CSDFILDE Decryption Utility 08/03/2006 (MM/DD/YYYY) 13:50:34 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      1
 CSDFILDE:          :
INPUT: DESC = TEST ENCRYPTED BIG FILE FOR EF RESIDENCY, COMPRESSED (Z9)
INPUT: LRECL     80 BLKSIZE   27920 RECFM FB
INPUT:  PASSWORD=********************************
RECORDS READ:           161  WRITTEN:       661,932
  BYTES READ:         4,497,880  BYTES RECOVERED: 1 52,954,560
CIPHER TIMES (IN SECONDS): HIGH:   0.001236 DATA:  279512 LOW:   0.000116 DATA:
28000
 TOTAL CIPHER TIME (IN SECONDS):     0.018672  CIPHERS:            17
  TOTAL EXPAND TIME (IN SECONDS): 0.153153 TOTAL EXPANDED BYTES: 56,926,160
  TOTAL ELAPSED TIME:   0:00:04.90
```

Figure 5-22 File decryption and decompression on z9

5.6 Encrypting and decrypting VSAM data sets

The programs CSDFILEN and CSDFILDE do not directly support VSAM data sets. Such data sets need to be unloaded to a sequential file first and then processed through the Encryption Facility for z/OS.

We tried three methods to achieve this:

- Using IDCAMS EXPORT
- Using IDCAMS REPRO
- Using DFSMSdss

The results highlighted an issue of which users need to take special note. The first test using IDCAMS EXPORT appears to work. The data is encrypted and decrypted. However, the reload of data using IDCAMS IMPORT fails.

We performed analysis on this and discovered that the programs CSDFILEN and CSDFILDE do not handle any files that contain SPANNED records. In the output file written by CSDFILDE, the segment control bytes (SCBs) have been removed from the record descriptor words (RDWs).

Spanned records are used by IDCAMS EXPORT and also by IEBCOPY UNLOAD. SMF records are also spanned. Clients might have other files that are of these formats.

The guide for the Encryption Facility clearly states that all RECORD formats are supported, but this appears not to be a correct statement. We understand that, as of the writing of this book, updates might be made to the guide to reflect this.

Important: The programs CSDFILEN and CSDFILDE do *not* support any files with RECFM=VBS or RECFM=VS. Corruptions might occur in the output files of CSDFILDE but there will be *no warning*. There is *no warning* produced when files of this format are processed by CSDFILEN either.

Note: If you are in any doubt about the support of your particular record format, we strongly recommend testing all files produced by decrypting them and then testing that the resulting output files are correctly formatted.

We believe the method shown later for VSAM data sets in 5.6.3, “Method 3: Using DFSMSdss” on page 76 will work for any format supported by DFSMSdss. However, you confirm this by your own testing if you have any unusual format or are in any doubt.

5.6.1 Method 1: Using IDCAMS EXPORT and IMPORT

This is where we experienced the spanned record problem. Do *not* use IDCAMS EXPORT files with the Encryption Facility.

5.6.2 Method 2: Using IDCAMS REPRO

The JCL shown here is intended to be example JCL that you can rerun. This method works correctly.

Example 5-5 IDCAMS REPRO

```
//UNLOAD   EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *

 /* FIRST LETS ENSURE WE HAVE NO DUPLICATE DATASETS */
 DELETE 'ITSOLD.Z990DEC.REPROED'
 DELETE 'ITSOLD.Z990DEC.REPROED.ENCRYPTD'
 DELETE 'ITSOLD.Z990DEC.REPROED.DECRYPTD'
 DELETE 'ITSOLD.Z990DEC.COPY'
 SET MAXCC=0

 /* NOW REPRO THE VSAM TEST FILE */
 REPRO  INFILE(IN) -
 OUTFILE(OUT)

/*
//IN       DD DSN=ITSOLD.Z990DEC,DISP=SHR
//OUT      DD DSN=ITSOLD.Z990DEC.REPROED,
//         DCB=(LRECL=252,BLKSIZE=0,RECFM=FB),
//         DISP=(,CATLG),SPACE=(CYL,(10,10))
//*
//ENCRYPT  EXEC  PGM=CSDFILEN
//SYSPRINT  DD   SYSOUT=*
//SYSUT1    DD   DISP=SHR,DSN=ITSOLD.Z990DECS.REPROED
//SYSUT2    DD   DISP=(,CATLG),
//          DSN=ITSOLD.Z990DEC.REPROED.ENCRYPTD,
//          SPACE=(CYL,(10,10))
//SYSIN     DD   *
DESC='Lennies VSAM test case with rsa keys'
ENCTDES
RSA=ITSOLD.TEST.RSA2048
/*
//*
//DECRYPT  EXEC  PGM=CSDFILDE
```

```
//SYSPRINT  DD   SYSOUT=*
//SYSUT1    DD   DISP=SHR,DSN=ITSOLD.Z990DEC.REPROED.ENCRYPTD
//SYSUT2    DD   DISP=(,CATLG),
//          DSN=ITSOLD.Z990DEC.REPROED.DECRYPTD,
//          SPACE=(CYL,(10,10))
//SYSIN     DD   *
RSA=ITSOLD.TEST.RSA2048
/*
//RELOAD   EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *

 DEFINE -
   CL(NAME(ITSOLD.Z990DEC.COPY) -
      MODEL(ITSOLD.Z990DECS)) -
      DATA(NAME(ITSOLD.Z990DEC.COPY.DATA)) -
      INDEX(NAME(ITSOLD.Z990DEC.COPY.INDEX)) -
      )

 REPRO INFILE(IN) OUTDATASET(ITSOLD.Z990DEC.COPY)

/*
//IN       DD DSN=ITSOLD.Z990DEC.REPROED.DECRYPTD,DISP=SHR
```

5.6.3 Method 3: Using DFSMSdss

The third method of running VSAM encryption is to dump the VSAM data set using DFSMSdss. After you go to this level of detail, you might want to use the built-in facilities of DFSMSdss as explained in Chapter 8, “The DFSMSdss Encryption feature” on page 103.

Example 5-6 DFSMSdss DUMP

```
//UNLOAD   EXEC PGM=IDCAMS
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *

 /* FIRST LETS ENSURE WE HAVE NO DUPLICATE DATASETS */
 DELETE 'ITSOLD.Z990DEC.DFDSSED'
 DELETE 'ITSOLD.Z990DEC.DFDSSED.ENCRYPTD'
 DELETE 'ITSOLD.Z990DEC.DFDSSED.DECRYPTD'
 DELETE 'ITSOLD.Z990DEC.COPY'
 SET MAXCC=0

/*
```

```
//*
//DSSDUMP  EXEC PGM=ADRDSSU
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
 DUMP DS(INC( -
 ITSOLD.Z990DEC -
 )) OUTDD(VSAMDUMP
/*
//VSAMDUMP  DD   DSN=ITSOLD.Z990DEC.DFDSSED,
//          DISP=(,CATLG),SPACE=(CYL,(10,10))
//*
//ENCRYPT  EXEC  PGM=CSDFILEN
//SYSPRINT  DD   SYSOUT=*
//SYSUT1    DD   DISP=SHR,DSN=ITSOLD.Z990DEC.DFDSSED
//SYSUT2    DD   DISP=(,CATLG),
//          DSN=ITSOLD.Z990DEC.DFDSSED.ENCRYPTD,
//          SPACE=(CYL,(10,10))
//SYSIN     DD   *
DESC='Lennies test case with rsa keys'
ENCTDES
RSA=ITSOLD.TEST.RSA2048
* A=ITSOLD.TEST.COPY.RSAKEY
/*
//*
//DECRYPT  EXEC  PGM=CSDFILDE
//SYSPRINT  DD   SYSOUT=*
//SYSUT1    DD   DISP=SHR,DSN=ITSOLD.Z990DEC.DFDSSED.ENCRYPTD
//SYSUT2    DD   DISP=(,CATLG),
//          DSN=ITSOLD.Z990DEC.DFDSSED.DECRYPTD,
//          SPACE=(CYL,(10,10))
//SYSIN     DD   *
* NOTHING
/*
//DSSREST  EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
 RESTORE -
 INDD(IN) -
 DS(INC(**)) -
 RENUNC((ITSOLD.Z990DEC,ITSOLD.Z990DEC.DECRYPTD)) -
 OUTDD(OUT)
/*
//IN       DD DSN=ITSOLD.Z990DEC.DFDSSED.DECRYPTD,DISP=SHR
//OUT      DD DSN=ITSOLD.Z990DEC.DFDSSED.DECRYPTD,DISP=SHR
```

5.7 z/OS UNIX pipe examples

A *pipe* in UNIX terminology is a FIFO file that acts, as the name implies, as a queue that can be filled by a program and emptied by another one.

Using the z/OS UNIX pipe is an easy way of overlapping the processing of two batch jobs: One job feeds the pipe with data that completed processing in this job, while another gets the data by reading the pipe and proceeds with whatever additional processing the data might require. The UNIX pipe technology interlock the writing and reading of the pipe data such that if the job that is reading from the pipe is submitted first, it will wait for data to be written to the pipe by the other job, and then read records as they are written.

We experimented with the use of a UNIX pipe with the output of the IEBGENER utility being “piped” into CSDFILEN.

5.7.1 Setting up a pipe in z/OS UNIX

The pipe has to be explicitly created before it can be used, as described here.

We created the z/OS UNIX file path /u/itsold/tmp/ and we now create the “pipe” as shown in Figure 5-23 in this path. To create the pipe, select **3** for FIFO.

```
  File  Directory  Special_file  Tools  File_systems  Options  Setup  Help
s +-------------------------------------------------------------------------+
  |                          Create a New File                              |
  |                                                                         |
E | Pathname:                                                               |
  |                                                          More:     +    |
  |   /u/itsold/tmp/temppipe                                                |
  |   ______________________________________________________________        |
  |   ______________________________________________________________        |
  |   ______________________________________________________________        |
R |                                                                         |
  | Permissions  . . 600  (3 digits, each 0-7)                              |
  |                                                                         |
  | File type                          File source for regular file         |
  | 3   1.  Directory                  __  1.  Edit...                      |
  |     2.  Regular file                   2.  Copy file...                 |
  |     3.  FIFO                           3.  Copy data set...             |
E |     4.  Symbolic link...                                                |
  |     5.  Hard link...               View and set attributes  . . . N     |
  |                                                                         |
  |                                                                         |
  |                                                                         |
  +-------------------------------------------------------------------------+
Command ===> ____________________________________________________________
```

Figure 5-23 Creating a pipe

tmp is a directory for temporary files and temppipe is the pipe file in this directory.

5.7.2 Using the pipe with two batch jobs communicating

The job that reads from the pipe can be submitted first. Example 5-8 shows the JCL to write the pipe. It waits until something is written into the pipe (note, however, that the job is to wait at a maximum until the JWT time specified in the SMFPRMxx member of SYS1.PARMLIB is met).

The job that writes to the pipe can be started afterward. Example 5-7 shows the JCL of this job. The records written into the pipe are passed directly to the fist job for processing.

Example 5-7 JCL to write to the pipe

```
//STEP1    EXEC  PGM=IEBGENER
//SYSUT2   DD    PATH='/u/itsold/tmp/temppipe',
//         LRECL=80,BLKSIZE=3200,RECFM=FB,
//         DSNTYPE=PIPE,
//         FILEDATA=BINARY,
//         PATHOPTS=(OWRONLY),
//         PATHMODE=SIRWXU
//SYSUT1   DD    DISP=SHR,DSN=ITSOLD.TEST.DATA
//SYSPRINT DD    SYSOUT=*
//SYSIN    DD    DUMMY
```

Example 5-8 JCL to read from the pipe

```
//ENCRYPT  EXEC  PGM=CSDFILEN
//SYSPRINT  DD   SYSOUT=*
//SYSUT1   DD    PATH='/u/itsold/tmp/temppipe',
//         LRECL=80,BLKSIZE=3200,RECFM=FB,
//         DSNTYPE=PIPE,
//         FILEDATA=BINARY,
//         PATHOPTS=(ORDONLY),
//         PATHMODE=SIRWXU
//SYSUT2    DD   DISP=OLD,
//          DSN=ITSOLD.TEST.DATA.ENCRYPTD,
//          LIKE=ITSOLD.TEST.DATA
//SYSIN     DD   *
DESC='Lennies test case with rsa keys'
ENCTDES
RSA=ITSOLD.TEST.RSA2048
/*
```

Important: Be sure to get the PATHOPTS statements specified correctly: One needs to specify ORDONLY to read and the other OWRONLY to write.

5.8 z/OS Decryption Client

The IBM Decryption Client for z/OS enables the decryption of the data created through the Encryption Services feature or the Java Client. The Decryption Client is supported on z/OS systems only and is similar to the CSDFILDE program of the Encryption Services feature.

You can download the Decryption Client from the following Web site:

http://www.ibm.com/servers/eserver/zseries/zos/downloads/#asis

The *readme* file contains complete information about software requirements including any required PTFs.

6

Certificate services in z/OS and usage considerations

This chapter describes the facilities available in z/OS to manipulate digital certificates in the context of use of the Encryption Facility for z/OS. It also discusses options to protect the data-encrypting key value in order to help you in making a decision about which option to choose.

6.1 Basic certificate services in z/OS

Note: The z/OS services that we describe here provide the basic certificate handling functions that are needed to exchange Encryption Facility encrypted files with RSA protection of the data key.

The services provided to generate and sign certificate can be seen as a local and limited certificate authority capability. These services can also be used within a wider implementation of a public key infrastructure.

There are two facilities that can assist in the certificate services that we need for the Encryption Facility. The first is the RACF RACDCERT command. The second is the set of utility ISPF panels provided in ICSF.

Certificate services with RACDCERT

The first method of creating and managing certificates and RSA keys is to use the RACF command RACDCERT.

The RACDCERT command can create RSA public and private keys and store them in the RACF database along with X.509 V3 certificates containing the public keys. The command can also create certificates whose keys are stored in the ICSF PKDS data set. This latter option is of use to us in setting up RSA keys for the Encryption Facility.

The RACDCERT command has been enhanced by the APAR OA13030, titled “New Function in RACF to support the z/OS Encryption Facility.” If you want to use RACDCERT to generate certificates for use with the Encryption Facility, you will find the enhancement provided by this APAR very useful. RACDCERT provides the capability to generate RSA public and private key pairs and store them in the PKDS data set. This enhancement enables you to specify the PKDS key label to use. (Prior to this enhancement, the label name was automatically generated during the execution of the RACDCERT GENCERT or RACDCERT ADD command.)

Note: After installing APAR OA13030, see SYS1.SAMPLIB(IRR13030) for a detailed description of the utility enhancements.

In 6.2, “Using the RACF RACDCERT command” on page 84, we describe in detail the use of the RACF RACDCERT command in the context of the Encryption Facility for z/OS.

Certificate services with ICSF panels

The second method of generating asymmetric keys is to use an enhancement to the ICSF panels. This enhancement is supplied by APAR OA15156, “New Function - PKDS Key management capabilities are being added to the ICSF utility panels.”

The new facilities are available for the following levels of ICSF, as shown in Table 6-1.

Table 6-1 ICSF utilities APAR OA15156

ICSF FMID	PTF
HCR770A	UA25815
HCR770B	UA25816
HCR7720	UA25817
HCR7730	UA25818

For the latest level of ICSF, as of the writing of this book FMID HCR7731 (the ICSF level delivered in z/OS V1R8), the new functions are included in the base code.

The new panels provide the capability of creating a PKDS public and private key pair, which are stored under a user-specified label in the PKDS. If the RACF CSFKEYS profiles are used to protect the PKDS keys, the user must be permitted in RACF to the key label as defined with a profile in the CSFKEYS class of resources.

Other options on the panel allow you to delete keys, to export an asymmetric public key to a self-signed X.509 V3 certificate, and also to import a public key that is contained in an existing X.509 V3 certificate file.

Store certificates used in this panel in sequential data sets with RECFM=VB. The certificates are Distinguished Encoding Rule (DER) encoded and contain binary data. When transporting such certificates, they must not undergo any character translation; that is, the binary pattern must remain as it is since it has been generated. The files used by these panels are compatible with files used by the RACDCERT EXPORT command when the parameter CERTDER parameter is used.

For full details of the enhancement, see:

`ftp://ftp.software.ibm.com/eserver/zseries/zos/icsf/pdf/oa15156.pdf`

Note: One difference between the use of the PKDS Key Management Utility and the RACF RACDCERT command is that RACDCERT, when adding a certificate to the RACF database, performs a verification of the certificate regarding the validity of the issuer's signature. The PKDS Key Management Utility does not do any verification.

In that respect, the RACDCERT command is probably the facility intended to be used if the certificates are to be provided by a certificate authority.

6.2 Using the RACF RACDCERT command

RACDCERT is a RACF command. It is used as a TSO command and follows normal TSO command conventions. It can be invoked from within ISPF option 6 or can be invoked using the Terminal Monitor Program (TMP) using JCL similar to that in Example 6-1 on page 85.

Note that here we use the RACDCERT command enhancements in APAR OA13030.

6.2.1 What the RACDCERT command does

RACDCERT has many functions. These are specified in the first parameter of the command. Table 6-2 shows parameters of importance in the context of use of the Encryption Facility and "How we used the RACDCERT command" on page 85 demonstrates their use.

Table 6-2 The RACDCERT command functions

Function	Description
ADD	Add a certificate to the RACF database
ALTER	Change the trust status of a certificate
CHECKCERT	Check if a certificate has already been added to the RACF database
DELETE	Delete a certificate from the RACF database
EXPORT	Export a certificate as a PKCS#7 package or a PKCS#12 package, or as a binary DER or Base 64 encoded X.509 V3 certificate
GENCERT	Generate a key pair and certificate
LIST	List a certificate content

How we used the RACDCERT command

We executed the RACDCERT command as a batch job using the JCL shown in Example 6-1.

Example 6-1 JCL to execute the RACF RACDCERT command

```
//RESTORE  EXEC  PGM=IKJEFT01
//SYSTSPRT  DD   SYSOUT=*
//SYSTSIN   DD   *
   <RACDCERT commands go here>
/*
```

This enabled us to save the output from the commands easily.

Example 6-2 shows the sequence of the RACDCERT commands necessary to generate an RSA key pair in the RACF database, along with the relevant self-signed certificate, and to export the certificate from the RACF database into a file.

Example 6-2 From key pair creation to certificate export

```
RACDCERT +
   DELETE ( +
   LABEL('ITSOLD.TEST.RSAKEY'))

RACDCERT +
   GENCERT +
   SUBJECTSDN(CN('lennie dymoke-bradshaw')) +
   SIZE(1024) +
   PCICC(*) +
   WITHLABEL('ITSOLD.TEST.RSAKEY')

RACDCERT +
   EXPORT  (+
   LABEL('ITSOLD.TEST.RSAKEY') +
   ) +
   DSN('LENNIE.TEST.RSAKEY') +
   FORMAT(CERTDER)
```

The RACDCERT commands perform the following functions:

1. The first RACDCERT command ensures that no X.509 V3 certificate already exists in the RACF database with the label we are about to create.

2. The second RACDCERT command generates an RSA public/private key pair and stores it in the PKDS with the label as specified. The size of the key is 1024 bits (alternatively this can have specified values of 512, 768, and 2048 bits). See 6.2.2, “RSA key recovery” on page 87 for important considerations regarding the use of the ICSF or PCICC parameters with the RACDCERT GENCERT command.

   This command also generates an X.509 V3 certificate that is stored in the RACF database.

   Note the PCICC keyword, meaning that a hardware coprocessor with RSA key generation capability will be invoked, and the generated keys will be stored in the PKDS with the private key encrypted with the coprocessor master key. When running in a z890, z990, or z9 environment, the “PCICC” keyword invokes the PCIXCC or the CEX2C coprocessor.

3. The third RACDCERT command exports the X.509 V3 certificate into a data set without the private key, which remains in the PKDS. The certificate is exported in DER-encoded format to a data set with RECFM=VB, which is dynamically created during the execution of the command.

Note the following:

- The certificate generated here will be “self-signed.” No certification authority has been involved in the creation of the certificate, nor has anyone else validated the certificate beside its creator.

  Note that the RACDCERT command also provides the functions necessary to get the certificates signed by an external certificate authority.

- Despite the previous point, if the certificate has been produced by a “trusted person,” it might be reasonable to use it if there are ways of ensuring that the certificate has been received from this person or through a trusted delegate or delegates.

- The X.509 V3 certificate was generated with a size of 1024 bits. RACDCERT also supports the generation of RSA keys with 512 bits, 768 bits, and 2048 bits. However, to create 2048-bit long RSA keys, you need, as already mentioned, proper hardware coprocessor support.

- The X.509 V3 certificate contains the public key that was generated in step 2 only. The private key remains in the PKDS data set.

The data set LENNIE.TEST.RSAKEY that was produced by the RACDCERT EXPORT command was taken to another computer site and stored as ITSOLD.TEST.RSAKEY.

Note: We used FTP for this purpose. If you do this, be sure to specify that a BINARY transfer is to take place. The certificate generated by the CERTDER parameter is in binary format and will be corrupted if FTP attempts to translate text.

The certificate was then imported into the target RACF system using the RACDCERT commands shown in Example 6-3. We imported the certificate as a SITE certificate, but you can import its as a user ID certificate. As long as the certificate is self-signed, its trust level remains the same.

Example 6-3 Importing the generated certificate into the target system's PKDS

```
RACDCERT +
   SITE +
   ADD('ITSOLD.TEST.RSAKEY') +
   TRUST +
   WITHLABEL('ITSOLD.TEST.RSAKEY') +
   PCICC(*)
```

The PCICC(*) keyword here indicates that RACF stores the public key value transported in the certificate into a PKDS key token with the same label as indicated in the WITHLABEL keyword.

If you use a certificate associated with a user ID, use care if the user ID is for a specific individual. If that person leaves and the user ID is deleted, the certificate will remain in place. But if IRRRID00 is run to remove all traces of the user ID from RACF, the certificate will be deleted. Certificate deletion will also attempt to remove the corresponding RSA keys from the PKDS.

6.2.2 RSA key recovery

As already mentioned, issuing the RACDCERT GENCERT command with the ICSF or PCICC parameters results in storing the RSA private key, encrypted with the coprocessor master key, in the PKDS. Then, there is no function available, by design, in ICSF or the hardware coprocessors to retrieve the clear value of the RSA private key. Although availability of the RSA private key in the PKDS can be insured through a periodic backup copy of the PKDS, there might be cases where such a clear value is needed, for example, in the context of a key recovery policy. You must then proceed as we describe here.

Important: This procedure is not applicable to key lengths of 2048 bits because RACDCERT requires the PCICC keyword to generate those keys (the generation is performed by a hardware cryptographic coprocessor). A consequence of using the PCICC keyword is that the keys are directly stored in the PKDS, thus preventing the use of this procedure.

Perform the following steps:

1. Issue the RACDCERT GENCERT command without the ICSF and PCICC parameters. The key pair is then stored in the RACF database.
2. Export the key pair with the RACDCERT EXPORT command to a PKCS#12 data set. Access to the private key is protected with a password, and this data set can then be kept for recovery of the private key.
3. Re-import the PKCS#12 data set with a RACDCERT ADD command with the ICSF parameter. The private key is then stored, unrecoverable in clear value, in the PKDS.

In Example 6-4, the data set to be kept for RSA private key recovery purpose is LENNIE.SAVE.RSAKEY.

Example 6-4 RACDCERT commands sequence to establish RSA key recovery

```
RACDCERT +
   GENCERT +
   SUBJECTSDN(CN('lennie dymoke-bradshaw')) +
   SIZE(1024) +
   WITHLABEL('ITSOLD.TEST.RSAKEY')

RACDCERT +
   EXPORT  (+
   LABEL('ITSOLD.TEST.RSAKEY') +
   ) +
   DSN('LENNIE.TEST.RSAKEY') +
   FORMAT(CERTDER)


RACDCERT +
   EXPORT  (+
   LABEL('ITSOLD.TEST.RSAKEY'))
   DSN('LENNIE.SAVE.RSAKEY') +
   FORMAT(PKCS12DER) +
   PASSWORD(’pkcs12-password’)

RACDCERT +
```

```
ADD (‘LENNIE.SAVE.RSAKEY’) +
TRUST +
WITHLABEL('ITSOLD.TEST.RSAKEY') +
PASSWORD(’pkcs12-password’) +
PCICC(*)
```

6.3 Using the ICSF utility panels

Figure 6-1 shows the ICSF main panel.

```
HCR7730  -------------- Integrated Cryptographic Service Facility--------------
OPTION ===>
Enter the number of the desired option.

  1  COPROCESSOR MGMT -  Management of Cryptographic Coprocessors
  2  MASTER KEY       -  Master key set or change, CKDS/PKDS Processing
  3  OPSTAT           -  Installation options
  4  ADMINCNTL        -  Administrative Control Functions
  5  UTILITY          -  ICSF Utilities
  6  PPINIT           -  Pass Phrase Master Key/CKDS Initialization
  7  TKE              -  TKE Master and Operational Key processing
  8  KGUP             -  Key Generator Utility processes
  9  UDX MGMT         -  Management of User Defined Extensions

     Licensed Materials - Property of IBM
     5694-A01 (C) Copyright IBM Corp. 1989, 2004.  All rights reserved.
     US Government Users Restricted Rights - Use, duplication or
     disclosure restricted by GSA ADP Schedule Contract with IBM Corp.


Press ENTER to go to the selected option.
Press END   to exit to the previous menu.
```

Figure 6-1 ICSF main panel

Option **5** from this panel takes you to the Utilities panel, as shown in Figure 6-2.

```
------------------------------ ICSF - Utilities ------------------------------
OPTION ===>

Enter the number of of the desired option.

  1  ENCODE     -  Encode data
  2  DECODE     -  Decode data
  3  RANDOM     -  Generate a random number
  4  CHECKSUM   -  Generate a checksum and verification and
                   hash pattern
  5  PPKEYS     -  Generate master key values from a pass phrase
  6  PKDSKEYS   -  Manage keys in the PKDS









Press ENTER to go to the selected option.
Press END   to exit to the previous menu.
```

Figure 6-2 ICSF Utilities panel

Select option **6** for the new facilities for PKDS keys. This opens the panel shown in Figure 6-3.

```
------------------------------ ICSF - PKDS Keys ------------------------------
COMMAND ===>
   Enter the PKDS record's label for the actions below
    ==>

   Select one of the following actions then press ENTER to process:

   _  Generate a new PKDS key pair record
      Enter the key length  ===>         512, 1024, or 2048
      Enter Private Key Name (optional)
       ==>

   _  Delete the existing public key or key pair PKDS record

   _  Export the PKDS record's public key to a certificate data set
      Enter the DSN  ===>
      Enter desired subject's common name (optional)
       CN=

   _  Create a PKDS public key record from an input certificate.
      Enter the DSN  ===>
```

Figure 6-3 ICSF PKDS Keys management panel

As shown, you must always reference a PKDS label. Whichever of the following options you select, this label will be used.

These options enable you to:

1. Generate a new PKDS public/private key pair and store it in a PKDS record with the label specified.

   Notes:

   - This facility is not available on a z800 or z900 platform without a PCICC card.
   - This does not provide the capability of recovering the clear value of the RSA private key.

2. Delete the PKDS record corresponding to the label specified.
3. Export the public key referenced by the PKDS label specified, and wrap it into an X.509 V3, self-signed, BER-encoded certificate contained in a data set.
4. Create a PKDS record containing only a public key from an existing data set containing a BER-encoded X.509 V3 certificate.

Note the following:

- ICSF does *not* proceed with any verification regarding the validity of certificates of which public key is imported in the PKDS.
- Certificates generated by the export facility are self-signed and are valid from 1 July 2006 until 31 December 2040.
- The data set the imported certificate is put in will be reused if it exists. If it is of the wrong format, it will be deleted and re-created. The required format is RECFM=VB.

Differences between the two methods

The RACDCERT command focuses on the X.509 V3 certificate services using self-signed certificates or certificates signed by a “local” (with the signing key in RACF) certificate authority or certificates signed by an external certificate authority, as you would expect when using a real public key infrastructure. The main option of interest for us in the context of the Encryption Facility for z/OS is to store the keys associated with the certificate into the PKDS.

Conversely, the ICSF panels focuses on creating and handling the RSA public and private key pairs in the PKDS. The panel also includes the ability to export the public key wrapped into a self-signed X.509 V3 certificate.

The method you choose depends on your situation. However, you can use both methods, one at the encrypting end and the other at the decrypting end.

6.4 Which data key protection option to use

There are five distinct ways in which the Encryption Facility can be used (see Table 4-1 on page 43). You must always use the same options to decrypt as were used to encrypt. The method you choose depends on some of the answers to the following questions:

- For what are you using the Encryption Facility?
- What are your security requirements?
- What are the risks of data exposure?
- What are the consequences of data exposure?
- Do you have experience with X.509 V3 certificates on z/OS?
- What are your current cryptographic capabilities?

You might find it easier to decide by using a structured approach addressing the following questions:

- Should I use AES or triple DES data encryption keys?
- Should I use clear keys or encrypted keys?
- Should I use RSA keys or passwords to protect the data encrypting keys?
- If I use RSA protection of the data key, should I use RACDCERT or the ICSF panels?

The following discussion lists some of the issues that should help you make a decision. This section is organized in part by your type of use of the Encryption Facility. We distinguish three types of use of the Encryption Facility, as shown in Table 6-3.

Table 6-3 The Encryption Facility for z/OS types of use

Type	Description
Archiving	Files are encrypted for use within the same computer site, or at least within the same enterprise or organization. Decryption is expected to take place rarely and probably in the same, or identical, z/OS system.
Ad hoc file transfer	Files are encrypted for transportation over a potentially unsecured mechanism. This might be by physically transporting a tape using a third-party carrier, or by sending the file over a public network. The files are sent very infrequently.

Type	Description
Regular file transfer	Files are encrypted for transportation over a potentially unsecured mechanism. This might be by physically transporting a tape using a third-party carrier, or by sending the file over a public network. The files are transported frequently.

Clearly, you can define other uses for the Encryption Facility, but these three are sufficient for us to illustrate the decision making processes you might need to go through.

6.4.1 Should I use AES or DES keys?

AES keys in this context mean AES 128-bit keys. These keys are symmetric and 16 bytes in length.

T-DES keys in this context mean triple-DES 192-bit keys. These keys are symmetric and, although they are 24 bytes in length, the key value itself is only 168 bits in length, because extra bits are parity bits being part of the triple DES key itself for historical reasons.

If you need to use the secure (encrypted) key option of the Encryption Facility for z/OS, you do not have the option of using AES.

This decision affects only those who use clear keys. If you have already decided to use encrypted keys, you must use T-DES.

It might be that one or other of the parties concerned mandates the use of triple DES (or conceivably of AES128). In this case, you will have no choice.

If given the information you still have not decided which key algorithm is more appropriate for you, consider the documentation provided by the National Institute of Standards and Technology, which is a U.S. government agency, available at:

http://csrc.nist.gov/

In addition, consider the following information.

Triple DES has been an accepted standard for many years by many organizations and bodies both in industry and government. Its behavior is “well-known.” In general, it is a trusted algorithm. However, this also means that it has been subjected to a great deal of scrutiny, some of which has exposed its shortcomings.

AES was developed in the 1990s as a replacement for triple DES and operates with three different lengths of keys: 128 bit, 192 bit, and 256 bit. We discuss the 128-bit version.

AES is reported to be stronger than T-DES at all three key lengths, largely due to the known shortcut attack possible for T-DES. No such shortcut attack has been published for AES yet, but it has been available for a far shorter time and so it is possible that such an attack might be discovered.

AES is also known to be computationally faster than T-DES. However, *be aware* that choosing AES-128 for the encryption of data by the Encryption Services results in software-only encryption or decryption on non-z9 platforms and might, therefore, prove to be unacceptable from the performance standpoint.

Ultimately, this is a choice you have to make, based on purely technical considerations or driven by organization-specific security policies and standards.

6.4.2 Should I use clear keys or encrypted keys?

“Encrypted keys” refers here to the ENCTDES option of the Encryption Facility for z/OS.

This decision is primarily a balance between the security policy and performance.

Clear keys are exposed in the storage of the computer. Encrypted keys are always manipulated in storage or other data repository while encrypted under a master key held in the tamper-proof coprocessor hardware.

By nature, encrypted keys are better protected than clear keys, and this will be true if we are dealing with a file transfer where both sites are using z/OS with ICSF and appropriate hardware protection.

Note that this option is not supported by the Java Client, because the encrypted key mechanism is specific to the use of ICSF, and is not suported as well on systems with CPACF only, because CPACF operates with clear keys only. Therefore, we need both partners using z/OS on a system with at least one secure coprocessor when ENCTDES is chosen. Note also that the master key encrypted triple DES data key is also securely encrypted, in the coprocessor, with the data recipient’s RSA public key. This is the latter form of encrypted key that is shipped with the data.

The recipient side ICSF enables you to securely re-encrypt the RSA encrypted triple DES key with the local master key. Therefore, at no point when creating the remote-use chain does the encrypted T-DES data key appear in the clear.

Note: Using ENCTDES does not require the sender and the recipient or recipitents of the data to share the same ICSF master key.

The use of clear keys is inherently faster than encrypted keys on systems with CPACF capabilities. If the system is not CPACF capable, the Encryption Facility usse the same coprocessors to perform the encryption and the decryption using clear keys or encrypted keys. In that case, you can expect very little difference in performance.

For performance considerations, see Appendix C, “Encryption Facility for z/OS: Performance and sizing” on page 165.

6.4.3 Should I use RSA keys or passwords?

Note the following general considerations:

- The Encryption Facility features that use ICSF, and therefore RSA private keys stored in the PKDS, benefit from a technology that provides an extremely secure environment to preserve the secrecy of the keys. However, in practice, it is only as secure as your partner’s own ability to set up and exploit properly this secure environment. It is, therefore, a recommended practice to check with your partner if either of you is affecting in some ways the security standards that the other ones expects to be in effect.
- Whenever RSA keys are used, their access must be controlled with an appropriate mechanism such as RACF. RACF protects keys using the CSFKEYS class. If you do not provide this level of protection, the use of the RSA keys is essentially uncontrolled. However, the values of RSA private keys are always protected by the ICSF asymmetric master key if they are held in the ICSF environment.
- After using the RSA keys outside the z/OS ICSF environment, you have to rely on other mechanisms, if available, to ensure their protection. This is the case of the Java Client with keystores protected by a password.
- If you uses the Encryption Facility for multiple purposes, this might have a bearing on which method you choose.

Archiving

If you use the Encryption Facility for long-term archiving of data, password protection might not be appropriate. The passwords are specified in-stream in the jobs and so are exposed there. If the password was exposed, the data would also be exposed and might possibly need re-encrypting.

If you use a password scheme for archiving, some form of management might be required for passwords if many data sets are archived.

Using the RSA capabilities is probably more appropriate for data archiving. However, this might lead to the need for more advanced hardware capabilities to support RSA keys and a need to manage the RSA keys. Consider the protection of the RSA keys. Access to RSA keys within a z/OS system is controlled using RACF protection of the key label in the CSFKEYS class.

Ad hoc file transfer

If a file is transferred to another site and there is a need to protect the data en-route, a password protection scheme might provide adequate protection. This applies particularly if the same person uses the data at the encryption site as at the decryption site.

Making use of RSA keys in an ad hoc situation might be appropriate, but always consider which keys are being used. If the keys used are also being used for other purposes, this might not be an appropriate way of protecting the data-encrypting key. Access to RSA keys within a z/OS system is controlled using RACF protection of the key label in the CSFKEYS class.

Regular file transfer

Regular file transfers between one enterprise and another (business partners or service providers and clients, and so on) is probably best handled using RSA keys.

This method avoids the need to regularly code passwords in the JCL, which runs CSDFILEN and CSDFILDE. This helps keep the data secure.

Specific RSA keys are probably best used so that RACF access to those keys can be controlled. Access to RSA keys within a z/OS system is controlled using RACF protection of the key label in the CSFKEYS class.

If I use PASSWORD=, what else should I consider?

Consider the following points.

DES or AES

If you decide to use the PASSWORD option, you need to decide whether to use CLRTDES or CLRAES128. The decision here is much the same as documented 6.4.1, “Should I use AES or DES keys?” on page 93.

ICOUNT

When you use PASSWORD, it is also possible to specify the ICOUNT keyword. ICOUNT controls the process of key derivation in Password Based Encryption

(PBE) and is intended to compensate for possibly weak passwords. The default of 16 provides reasonable security and performance if the password is robust (that is, 32 random characters). If a weaker password is selected (shorter than 32 random characters), iteration counts of 1000 or higher are often normal.

Protection of the password

You also need to consider how you communicate the password to the other party. Obviously, this needs to be secured in some way.

However, you also need to be aware that the password is likely to be exposed:

- In the JCL deck or parameter file if this is stored on disk
- On the JES2 or JES3 spool during the running of the job
- In script to control the execution of the Java Client

You can choose to place the password in a separate data set:

```
//SYSIN DD DISP=SHR,DSN=Your.Parm.Library(member)
```

If so, this can be used as input to the CSDFILEN and CSDFILDE programs. This data set can be RACF-protected independently from other data sets, thus providing enhanced security.

Neither CSDFILEN nor CSDFILDE will disclose the password on the statistics report, and in this case, the spool will not contain the password.

However, you instead can place the PASSWORD value in an instream data set as follows:

```
//SYSIN DD *
```

In this case, take steps to ensure that the spool data sets for JES2 or JES3 spools are RACF protected, and that the individual spool files are protected by making use of the JESSPOOL RACF class.

6.4.4 If I use RSA, should I use the RACDCERT command or the ICSF panels?

If you decide to use RSA key protection for your symmetric keys, you need to decide how your RSA keys are to be defined and transported. Earlier we described the RACDCERT RACF command and the ISCF panels options.

Note the following general considerations:

- If you use the Encryption Facility for multiple purposes, this might take you down a specific path.
- It is probably a good idea to use the same method in all cases.

- The fact that you use a specific method does not prevent your partner from using another method.
- X.509 V3 certificates will almost certainly be the method for packaging RSA keys for distribution. It is possible to use other methods, but we do not discuss them here.
- If you already use X.509 V3 certificates on the z/OS platform, this should influence your decision in favor of using certificates in RACDCERT.
- If you send files to another enterprise or receive files from another enterprise, that enterprise might require you to use specific mechanisms to protect and manipulate RSA keys.

Archiving

If you use the Encryption Facility only for archiving, the ICSF panels provide a very simple method of defining RSA keys that does not involve deep X.509 V3 certificates management. If it is unlikely that you will need to use X.509 V3 certificates on z/OS in the future, this is probably a good choice for you.

Ad hoc file transfer

If this is truly ad hoc, the use of the ICSF panels with the generation of X.509 V3 certificates used purely for transportation of the RSA public key can be an adequate mechanism.

You always need to consider how the keys are transported, because X.509 V3 certificates generated by the ICSF panels are self-signed and thus the communication methods used for keys may be vulnerable to “man in the middle” attacks. However, such attacks are clearly less likely with ad hoc transfers.

Regular file transfers

Regular file transfers are probably best handled with RSA keys transported through properly signed and trusted X.509 V3 certificates.

However, if either site does not have any experience using X.509 V3 certificates in z/OS, the ICSF panels provide a means to import the RSA keys without getting involved in the wider issues of X.509 V3 certificate management.

7

RACF protection of Encryption Facility for z/OS resources

In this chapter, we discuss ways of protecting the system resources used by the Encryption Facility for z/OS with RACF.

7.1 Protecting the Encryption Facility

The availability of the Encryption Facility on your z/OS system can have some undesirable side effects.

The Encryption Facility makes it relatively easy for an employee to hide data that, for example, that employee is sharing without authorization.

In this situation, it is probable that the employee would use the password method of encryption rather than the RSA method. This is simply because the RSA method requires the keys to be available within the PKDS and it is unlikely that most users have authority to create such keys in the PKDS. Furthermore, if the keys are in the PKDS, the employer can easily retrieve them, thus making the use of RSA keys less attractive to the employee.

Conversely, the use of the password option means that only the employee has the knowledge to decrypt the data.

Therefore, consider protecting the Encryption Facility from use by such people.

There are several methods. We discuss each with their advantages and disadvantages, but the final decision will have to be yours.

7.1.1 RACF protection of the Encryption Facility load library

If access to the load library for the Encryption Facility is restricted, consider the following possible exposures:

- Do not place the Encryption Facility load library in the system Linklist or the system LPA list. The programs CSDFILEN and CSDFILDE can be accessed without forcing an open of the load library, and so the RACF protection is bypassed.
- The current installation of the Encryption Facility installs the programs in SYS1.LINKLIB, so SMP/E changes might be needed to ensure that maintenance to the programs is applied correctly.
- The Encryption Facility programs do not require APF authorization, so there is no system built-in discrimination on the source of the program or the library from which it is executed.
- If you also have the DFSMSdss Encryption feature installed, take steps to secure this as well. You might have DFSMSdss protected already using RACF FACILITY class profiles, but note that you cannot allow the use of DFSMSdss with these FACILITY class profiles and at the same time prevent the data encryption.

- Use the Encryption Facility Java Client on your system to provide similar services.
- Any programmer with sufficient knowledge can write programs to encrypt data using ICSF services. It is, of course, possible that this is already true before the Encryption Facility was installed.

7.1.2 RACF protection of the Encryption Facility load modules

You can protect the load modules CSDFILEN and CSDFILDE using the RACF PROGRAM class. If you take this approach, consider the following possible exposures:

- Programs CSDFILEN and CSDFILDE do not require APF authorization. Consequently, a copy of the programs can be introduced from some other source and executed from another non-APF authorized library.
- If you also have the DFSMSdss Encryption feature installed, the encryption features in that program also need protection in a similar way.
- Any programmer with sufficient knowledge can write programs to encrypt data using ICSF services. It is, of course, possible that this is already true before the Encryption Facility was installed.
- Use the Encryption Facility Java Client on your system to provide similar services.

7.2 RACF protection of the ICSF services used by the Encryption Facility

Both features of the Encryption Facility for z/OS (Encryption Services and DFSMSdss Encryption feature) make use of calls to ICSF. Therefore, it is possible to protect the calls to ICSF using RACF resources in the CSFSERV class. If you want to do this, note the following information:

- The CSFSERV resources might be needed for some other reason than running the Encryption Facility.
- The ICSF services invoked by the Encryption Facility vary according to the options chosen in CSDFILEN. Refer to “Cryptographic services exploited on z9, z990, and z980” on page 32 and “Cryptographic services exploited on z900 and z800” on page 34 for further details about the services used.
- Note that access to the CPACF encryption and decryption functions are not protected by RACF profiles, whether they are invoked by the CSNBSYE and CSNBSYD services in ICSF or whether the CPACF is directly invoked using assembler instructions.

- Any programmer with sufficient knowledge can write programs to encrypt data using ICSF services. It is, of course, possible that this is already true before the Encryption Facility was installed. However, as long as the ICSF services are being called from problem-program state, protection of the appropriate CSFSERV resources will prevent this from executing successfully.
- Use the Encryption Facility Java Client on your system to provide similar services. This does not currently make use of ICSF services.

8

The DFSMSdss Encryption feature

This chapter describes what is needed to install the DFSMSdss Encryption feature and provides some examples of its use.

8.1 Getting started

The Encryption Facility for z/OS DFSMSdss Encryption feature encrypts DSS dump data sets written to tape and DASD. The DFSMSdss Encryption feature uses T-DES keys or 128-bit AES keys to protect the data. There is an option to use RSA public/private keys to protect the T-DES and AES data keys.

The DFSMSdss Encryption feature is an optional priced feature. You can order this product through your IBM marketing representative or through your regular product ordering channel.

Because ICSF is required to be started and a preliminary setup might have to be done using the RACF RACDCERT command, we recommend the following three books to be used as reference documentation:

- *z/OS ICSF Administrator's Guide*, SA22-7521
- *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683
- *z/OS Security Server RACF Command Language Reference*, SA22-7687

This chapter is written as a stand-alone chapter apart from the rest of the book. We assume that you are just interested in the DFSMSdss Encryption feature. However, we also recommend that you review Appendix B, “Some encryption basics” on page 155 to get a high-level idea about how the cryptographic functions invoked by DFSMSdss work.

8.2 Hardware and software requirements

In this section, we explain the software and hardware prerequisites for the DFSMSdss DUMP and RESTORE commands when encrypting and decrypting data.

8.2.1 Hardware requirements

To use the DFSMSdss Encryption feature, you need to execute DFSMSdss on a zSeries or z9 platform that has hardware cryptographic devices installed and enabled. See Table 8-1 on page 105 for the minimum cryptographic hardware requirements for the KEYPASSWORD and RSA options. Satisfying the minimum requirements for KEYPASSWORD and RSA also satisfies the requirements for CLRTDES, ENCTDES, and CLRAES128.

Table 8-1 Minimum cryptographic hardware requirements

Processor	KEYPASSWORD	RSA	
		1024-bit key	2048-bit key
z900	CCF	CCF	PCICC
z800	CCF	CCF	PCICC
z990	CPACF	CEX2C/PCIXCC	CEX2C/PCIXCC
z890	CPACF	CEX2C/PCIXCC	CEX2C/PCIXCC
z9	CPACF	CEX2C/PCIXCC	CEX2C/PCIXCC

We also recommend that you have the most up to date MCL patch installed on your cryptographic coprocessors. For the PCICCs, you need to be using LIC from January 2005 or later.

8.2.2 Software requirements

You need to install two FMIDs and some PTFs in order have a fully functional DFSMSdss Encryption feature.

FMID HCF773D needs to be installed for the DFSMSdss Encryption feature to work.

For ICSF, you need to be at FMID HCR770B or later. Download the latest ICSF release at:

http://www.ibm.com/servers/eserver/zseries/zos/downloads/

Table 8-2 lists the additional maintenance needed.

Table 8-2 Additional maintenance

Component	Required APAR
DFSMSdss	OA13300
	OA15165
	OA16852
	OA14991
	OA14481

Component	Required APAR
ISMF (Interactive Storage Management Facility)	OA13316
	OA13904
HSM (hierarchical storage management)	OA13453
	OA13687

Finally, you need to review the Encryption Facility PSP bucket ZOSEFV1R1 for any APARs that were closed after this redbook was written. Look in subset EFDSS773D. Also check the DFSMS PSP bucket for any related maintenance.

8.3 Cryptography options

There are six new keywords that can be used to invoke the DFSMSdss Encryption feature.

The KEYPASSWORD keyword generates the data key used to encrypt the data, either a clear TDES triple-length key or a clear 128-bit AES key. The password has to be a minimum of eight characters long and a maximum of 32 characters long. Use a password that is as long as possible and contains random characters. However, do not consider this method properly secure in many cases because the password is in the clear within the JCL and the JOBLOG.

Note: “In the clear” is a common term in the cryptographic world used to describe data that is left exposed and unprotected by any form of encryption.

The ICOUNT (iteration count) keyword is used in conjunction with the KEYPASSWORD option. The ICOUNT keyword strengthens the derivation process used to generate the data key from the password. The higher the ICOUNT value, the more robust to attack the derived key is. A good ICOUNT value to use is 1000 or higher.

The RSA keyword specifies a key label of an RSA public or private key located in the PKDS. The PKDS is an ICSF data set that is used to store RSA public and private key pairs. The public key is used during the DUMP job to encrypt the randomly generated data key if the password option is not used. The private key is used during the RESTORE job to decrypt the data key. The RSA key label of the public key and the encrypted data key are stored in the dump data set.

On the recovery site, if the private key is stored in the PKDS under the same label as the public key, you do not need to code the RSA keyword for RESTORE.

RESTORE uses the key label that is written in the dump data set as the private key label. If the private key is stored under a different label, you need to use the RSA keyword to indicate under which key label the private key is stored.

There is a subtle difference between the CLRTDES and the ENCTDES options. The CLRTDES option generates a T-DES data key that is in the clear in the address space. The ENCTDES option generates the T-DES data key within the secure cryptographic coprocessor hardware, that is, the CCF or CEX2C. The T-DES data key will never be in the clear when outside of the coprocessor.

However, there is a performance advantage in using CLRTDES if you are running on a z990, z890, or z9 platform with the CPACF. The CPACF is a set of cryptographic functions imbedded in each processing unit (PU) of the system that perform hashing and clear T-DES and AES-128 cryptography (the latter in System z9 environments only. The DFSMSdss Encryption feature performs the CLRTDES function with the CPACF machine instructions. The machine instructions provide faster cryptographic services because the process does not incur the instructions path length of calling the ICSF APIs to pass the work to the secure cryptographic hardware coprocessor.

The CLRAES128 option is performed by CPACF instructions if the job is running on a z9 platform. On any other processor (z900, z800, z990, or z890), the AES cryptography is done by the ICSF software. There is no cryptographic hardware support for AES on those system models.

Note: When using CLRAES128 on a z9 platform, make sure that you have APAR OA16852 installed.

If you have trouble deciding whether to use AES or TDES, take a look in Chapter 6, “Certificate services in z/OS and usage considerations” on page 81 for some facts about both algorithms. We feel that both are equally secure algorithms.

In summary, you can have the following cryptographic options combinations:

- KEYPASSWORD and CLRAES128
- KEYPASSWORD and CLRTDES
- RSA and CLRAES128
- RSA and CLRTDES
- RSA and ENCTDES

The KEYPASSWORD options are less secure than the RSA options. The RSA options provide extra protection to the data key. The most secure combination of options is RSA with ENCTDES.

8.4 Compression

The DFSMSdss Encryption feature also adds a new compression option to the DFSMSdss DUMP command. This new compression option makes use of the the zSeries and System z CMPSC machine instruction. To use this compression option, you code the keyword HWCOMPRESS (hardware-assisted compression) on the DUMP command. HWCOMPRESS and COMPRESS are mutually exclusive keywords for DFSMSdss DUMP. Both compression methods work with the DFSMSdss Encryption feature.

The compression algorithm used by HWCOMPRESS is different from the compression algorithm used by the existing COMPRESS option. The HWCOMPRESS option compresses repeated patterns of bytes. The COMPRESS option compresses repeated sequential bytes. Base the choice of which compression method to use on the type of data being compressed.

See Table 8-3 for the sizes of data sets after compression. The encryption options you choose, that is, KEYPASSWORD, RSA, CLRAES128, CLRTDES, or ENCTDES, do not affect the size of the resulting data set.

All the data sets were dumped onto DASD.

Table 8-3 Size of data sets before and after compression

Data type	Number of lines of text	Size of data in CYLs			
		Before Dump	No Compression	COMPRESS	HWCOMPRESS
SYSLOG 1	44,434	9	9	6	4
SYSLOG 2	209,031	34	34	33	9
SYSLOG 3	151,663	20	23	18	12
EREP REPORT	113,378	18	19	6	5
SVC dump	N/A	464	464	199	198
Load module	N/A	4	4	4	4

Looking at the compression results, HWCOMPRESS seems to have a slight advantage over COMPRESS. HWCOMPRESS does very well on SYSLOGs. SYSLOG #1 was taken from a JES3 system during normal activity. SYSLOG #2 was taken from a JES2 system taken when there was a problem with the system. There were a lot of repeating messages in the SYSLOG, which is perhaps why

HWCOMPRESS worked so well. SYSLOG #3 is also from a JES2 system during a problem. Again, there are repeating messages in the SYSLOG.

For the best results on your system, you might want to perform some testing on your own. You might find that some of your data compress better with the COMPRESS option instead of the HWCOMPRESS.

8.5 RSA key management

There are two facilities to manage the RSA keys. You can use ICSF with the PKDS key management panel introduced by OA15156, or you can use the RACF RACDCERT command with OA13030 applied.

The simplest approach in the case of DUMP/RESTORE is p the ICSF RSA key management panel, shown in Figure 8-1.

```
------------------------------- ICSF - PKDS Keys ------------------------------
   Enter the PKDS record's label for the actions below
    ==>

   Select one of the following actions then press ENTER to process:

   _  Generate a new PKDS key pair record
      Enter the key length  ===>         512, 1024, or 2048
      Enter Private Key Name (optional)
       ==>

   _  Delete the existing public key or key pair PKDS record

   _  Export the PKDS record's public key to a certificate data set
      Enter the DSN  ===>
      Enter desired subject's common name (optional)
       CN=

   _  Create a PKDS public key record from an input certificate.
      Enter the DSN  ===>



 COMMAND ===>
```

Figure 8-1 RSA key

This panel provides you with four options:

- Create a RSA key pair
- Delete a key pair or a stand-alone public key
- Export a stand-alone public key into a certificate data set
- Add a stand-alone public key into the PKDS

For the third option, you export the public key as a self-signed certificate in a data set.

The alternative to ICSF is to use the RACF RACDCERT command, which can be used to store and maintain information about digital certificates. For the DFSMSdss Encryption feature, the certificate acts as a vessel to hold the RSA public keys.

One very useful feature of a certificate is its authentication process. A certificate is authenticated by a signer. A signer authenticates the certificate by digitally signing it with its private key. A certificate can be self-signed or signed by an external certificate authority (CA). CAs are often well-known commercial organizations, or they can be local or internal organizations. Caution is recommended when trusting self-signed certificates because anyone can create them. Certificates signed by a CA offer an extra level of trust because no one has the private key except the CA.

When deciding on which facility to use to manage your Encryption feature RSA keys, consider how the RSA keys will be used. If the RSA keys are kept in house and you are not familiar with the RACDCERT command, ICSF might be the best facility to use. If you are going to send and receive keys, using the RACDCERT command might be the best option because of the certificate authentication process that is performed when RACF adds a new certificate to the RACF database (and therefore a new public key to the PKDS). The public keys are sent through a certificate. For a high level of trust, these certificates must be signed by a recognized and trusted CA. Therefore, the receiver will recognize the certificate and the public key it contains as the correct one from the correct company.

8.6 DFSMShsm encryption

HSM makes use of the DSS encryption support to provide encryption for HSM full volume dumps. As noted by Table 8-2 on page 105, you need to apply OA13453 and OA13687 to use this feature. A detailed description about how to set it up is in the APAR of OA13453.

Note: There is currently no encryption support for aggregate backup and recovery support (ABARs).

8.7 Implementation scenarios

Use this section as a supplement to the examples in *Encryption Facility for z/OS: User's Guide Version 1 Release 1.0*, SA23-1349, and *z/OS DFSMSdss Storage*

Administration Guide, SC35-0423. The intent behind these examples is to highlight the DFSMSdss Encryption feature options.

In the following scenarios, everyone has the proper RACF authority to access the necessary resources on their systems to perform their tasks. These scenarios do not show the procedure to give the individuals access to the resources.

8.7.1 Scenario 1: DFSMSdss using KEYPASSWORD

Anne Summers of the fictitious ITSO Magic Box Company wants to send some critical data to Gwen Post of the fictitious Watchers Inc. Both companies have a policy that any data transferred to each other must be encrypted. Anne has a z/OS V1.7 system with ICSF FMID HCR7720 running on a z9 platform with only the CPACF available. Gwen has a z/OS V1.5 system with ICSF FMID HCR770B running on a z800 platform with two PCICC cards. Because Anne does not have a CEX2C, they cannot use the RSA option. Therefore, Gwen and Anne use the KEYPASSWORD option. Scenario 1 proceeds as follows:

1. The systems at Magic Box Company are back level in maintenance. Anne notices she needs to apply the PTF for OA15165 on her system before she can use the DFSMSdss Encryption feature.
2. Anne uses the JCL in Example 8-1 to encrypt the data sets she will send to Gwen at Watchers Inc. In her password, Anne incorporates uppercase and lowercase letters, the number zero, and the underscore character. She uses an ICOUNT value of 1003 to strengthen the key derivation process. The encrypted data will be in a data set called ANNE.WATCHERS.DATA.DUMP.

Example 8-1 Step 1: DUMP JCL

```
//DUMP     EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DSN=ANNE.WATCHERS.DATA.DUMP,
//            DISP=(NEW,CATLG),
//            SPACE=(CYL,(35,1),RLSE),
//            UNIT=SYSDA
//SYSIN    DD *
  DUMP DATASET (INCLUDE(ANNE.WATCHERS.**)) -
       OUTDDNAME(ENCFILE) -
       KEYPASSWORD(NOrmal_is_the_WatchwOrd) -
       ICOUNT(1003) -
       ENCRYPT(CLRTDES) -
       HWCOMPRESS
/*
```

3. Anne sends the data to Gwen. Because the data is already encrypted, there might not be a high level of security required for the transfer method.
4. Anne sends the KEYPASSWORD value to Gwen through a secure and trusted channel.
5. Gwen uses the JCL in Example 8-2 to decrypt the data. Gwen named Anne's data set GWEN.ENCRYPT.DATA.DUMP. All Gwen needs is the KEYPASSWORD value to restore the data sets.

Example 8-2 Step 4: RESTORE JCL

```
//RESTORE  EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DISP=SHR,DSN=GWEN.ENCRYPT.DATA.DUMP
//DECFILE  DD DISP=(NEW,CATLG),
//            UNIT=SYSDA,
//            SPACE=(CYL,(50,5),RLSE)
//SYSIN    DD *
  RESTORE DATASET (INCLUDE(**)) -
             INDD(ENCFILE) -
             OUTDDNAME(DECFILE) -
             CATALOG -
             KEYPASSWORD(NOrmal_is_the_WatchwOrd)
/*
```

8.7.2 Scenario 2: DFSMSdss with the RSA option on one system

Gwen Post needs to encrypt Anne's data more securely on her system. She has a z/OS V1.5 system with ICSF FMID HCR770B running on a z800 with two PCICC. She has all the requisite hardware and software to use a RSA key to protect the data key used to encrypt the data. She will use the ICSF panels to create the RSA public and private key pair. Then, she will use DFSMSdss to encrypt the data and store it on tape. This scenario proceeds as follows:

1. Gwen verifies that the PTF for the ICSF APAR OA15156 is applied on her system.

2. To create the RSA key pair, she first selects option **5** from the main ICSF panel, as shown in Figure 8-2.

```
HCR770B  -------------- Integrated Cryptographic Service Facility--------------
Enter the number of the desired option.

  1  COPROCESSOR MGMT -  Management of Cryptographic Coprocessors
  2  MASTER KEY       -  Master key set or change, CKDS/PKDS Processing
  3  OPSTAT           -  Installation options
  4  ADMINCNTL        -  Administrative Control Functions
  5  UTILITY          -  ICSF Utilities
  6  PPINIT           -  Pass Phrase Master Key/CKDS Initialization
  7  TKE              -  TKE Master and Operational Key processing
  8  KGUP             -  Key Generator Utility processes
  9  UDX MGMT         -  Management of User Defined Extensions

     Licensed Materials - Property of IBM
     5694-A01 (C) Copyright IBM Corp. 1989, 2003.  All rights reserved.
     US Government Users Restricted Rights - Use, duplication or
     disclosure restricted by GSA ADP Schedule Contract with IBM Corp.


Press ENTER to go to the selected option.
Press END   to exit to the previous menu.

OPTION ===> 5
```

Figure 8-2 Select the UTILITY option on the main ICSF panel

3. She then selects option **6 PKDSKEYS**, as shown in Figure 8-3.

```
 ------------------------------ ICSF - Utilities ------------------------------

 Enter the number of the desired option.

   1  ENCODE     -  Encode data
   2  DECODE     -  Decode data
   3  RANDOM     -  Generate a random number
   4  CHECKSUM   -  Generate a checksum and verification and
                    hash pattern
   5  PPKEYS     -  Generate master key values from a pass phrase
   6  PKDSKEYS   -  Manage keys in the PKDS









 Press ENTER to go to the selected option.
 Press END   to exit to the previous menu.

 OPTION ===> 6
```

Figure 8-3 Select the PKDSKEYS option

4. Gwen enters the key label GWEN.1024.KEY and creates a 1024-bit RSA key pair, as shown in Figure 8-4. Gwen receives confirmation that ICSF created the public and private keys successfully, as shown in Figure 8-5.

```
------------------------------- ICSF - PKDS Keys ----------------------------
   Enter the PKDS record's label for the actions below
    ==>  GWEN.1024.KEY

   Select one of the following actions then press ENTER to process:

   S  Generate a new PKDS key pair record
      Enter the key length  ===> 1024     512, 1024, or 2048
      Enter Private Key Name (optional)
       ==>

   _  Delete the existing public key or key pair PKDS record

   _  Export the PKDS record's public key to a certificate data set
      Enter the DSN  ===>
      Enter desired subject's common name (optional)
       CN=

   _  Create a PKDS public key record from an input certificate.
      Enter the DSN  ===>


COMMAND ===>
```

Figure 8-4 Create a 1024-bit RSA key pair

```
---------------------- ICSF - PKDS Key Request Successful ---------------------


  Label ==> GWEN.1024.KEY




  Key function completed successfully

  Press ENTER or END to return to the previous menu.





COMMAND ===>
```

Figure 8-5 ICSF confirms that the key pair has been successfully created

5. Gwen uses the JCL shown in Figure 8-6 to encrypt the data Anne sent. She specifies the key label GWEN.1024.KEY. Only the public key is used by the DUMP process.

```
//DSS      EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DSN=GWEN.RSA.WATCHERS.DATA01,
//            DISP=(NEW,CATLG),
//            UNIT=3590,VOL=SER=TAPE01
//SYSIN    DD *
  DUMP DATASET(INCLUDE(GWEN.ANNE.**)) -
       OUTDDNAME(ENCFILE) -
       RSA(GWEN.1024.KEY) -
       ENCRYPT(ENCTDES) -
       HWCOMPRESS
/*
```

Figure 8-6 JCL to encrypt data invoking the RSA private key

6. Gwen uses the JCL shown in Figure 8-7 to decrypt the data set she created in the previous step. The RSA key label she specifies in the JCL is the same key label she used in the previous encrypt job, because we assume here that we are using the same PKDS as we did for encryption, which holds the public and private RSA keys under the same label.

```
//RESTORE  EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DSN=GWEN.RSA.WATCHERS.DATA01,
//            DISP=SHR,
//            UNIT=3590,VOL=SER=TAPE01
//DECFILE  DD DISP=(NEW,CATLG),
//            UNIT=SYSDA,
//            SPACE=(CYL,(200,5),RLSE)
//SYSIN    DD *
  RESTORE DATASET (INCLUDE(**)) -
            INDD(ENCFILE) -
            OUTDDNAME(DECFILE) -
            CATALOG -
            RSA(GWEN.1024.KEY)
/*
```

Figure 8-7 JCL to decrypt the data invoking the RSA key

Note: Specifying the RSA key label again as shown in this example is actually not necessary here: The initial RSA key label was stored with the archived data, and because we are restoring on the same system with the same PKDS, DSMSdss has enough information to retrieve the private RSA key to use.

8.7.3 Scenario 3: DFSMSdss with the RSA option on two systems

The Magic Box Company has bought two CEX2C cards and installed them on the z9 platform. Anne and Gwen have agreed to use the RSA option when Anne sends the data to Gwen. Anne has a z/OS V1.7 system with ICSF FMID HCR7720 running on a z9 platform with two CEX2C cards. Gwen has a z/OS V1.5 system with ICSF FMID HCR770B running on a z800 with two PCICCs. Anne and Gwen will use the ICSF panels to add the RSA keys to their systems. Because Anne will be sending data to Gwen, Gwen has to send her public key to Anne. Anne will use Gwen's public key to encrypt the T-DES data key. Gwen will use her private key to decrypt the data key. The scenario proceeds as follows:

1. Gwen creates a new RSA key pair with the ICSF panels as in the previous scenario. The key size is 1024 bits. Gwen calls the new key pair WATCHERS.MAGICBOX.1024.KEY.
2. Gwen exports the public key into a certificate data set, as shown in Figure 8-8. ICSF confirms that the data set has been successfully created, as shown in Figure 8-9 on page 117.

```
------------------------------ ICSF - PKDS Keys ------------------------------
   Enter the PKDS record's label for the actions below
    ==>  WATCHERS.MAGICBOX.1024.KEY

   Select one of the following actions then press ENTER to process:

   _  Generate a new PKDS key pair record
      Enter the key length  ===>          512, 1024, or 2048
      Enter Private Key Name (optional)
       ==>

   _  Delete the existing public key or key pair PKDS record

   s  Export the PKDS record's public key to a certificate data set
      Enter the DSN  ===> 'GWEN.WATCHERS.MAGICBOX.KEY'
      Enter desired subject's common name (optional)
       CN=

   _  Create a PKDS public key record from an input certificate.
      Enter the DSN  ===>


COMMAND ===>
```

Figure 8-8 Create certificate data set

```
------------------- ICSF - PKDS Public Key Export Successful ------------------


  Label ==> WATCHERS.MAGICBOX.1024.KEY

  Output Data Set  ===> 'GWEN.WATCHERS.MAGICBOX.KEY'


  Export to certificate successful. Binary (DER) certificate created.

  Press ENTER or END to return to the previous menu.









COMMAND ===>
```

Figure 8-9 ICSF confirms the creation of the certificate data set

3. The certificate data set is in binary. Gwen uses FTP to download the file onto her PC in BINARY mode. She e-mails it to Anne. Anne uses FTP to upload the certificate to the host in BINARY mode into an MVS™ data set. Anne names the data set ANNE.WATCHERS.MAGICBOX.KEY. Anne imports Gwen's public key into her PKDS with the ICSF panels, as shown in Figure 8-10. The import is successfully, as shown Figure 8-11 on page 118.

```
------------------------------- ICSF - PKDS Keys --------------------------
COMMAND ===>
   Enter the PKDS record's label for the actions below
    ==>  WATCHERS.1024.PUBLIC.KEY

   Select one of the following actions then press ENTER to process:

   _  Generate a new PKDS key pair record
      Enter the key length  ===>          512, 1024, or 2048
      Enter Private Key Name (optional)
       ==>

   _  Delete the existing public key or key pair PKDS record

   _  Export the PKDS record's public key to a certificate data set
      Enter the DSN  ===>
      Enter desired subject's common name (optional)
       CN=

   S  Create a PKDS public key record from an input certificate.
      Enter the DSN  ===> 'ITSOSJ.WATCHERS.MAGICBOX.KEY'
```

Figure 8-10 Import the public key from a certificate data set

```
------------------- ICSF - PKDS Public Key Import Successful ------------------
COMMAND ===>


  Label ==> WATCHERS.1024.PUBLIC.KEY

  Input Data Set  ===> 'ANNE.WATCHERS.MAGICBOX.KEY'


  Import from certificate successful. Public key PKDS entry created.

  Press ENTER or END to return to the previous menu.
```

Figure 8-11 ICSF confirms that the public key has been added successfully

4. Anne encrypts the data she will be sending with Gwen's RSA public key. She chooses to use CLRTDES since she has a z9 platform. On the z9 platform, the DFSMSdss Encryption feature will use the CPACF machine instructions to perform the clear T-DES encryption. The JCL Anne uses is shown in Figure 8-12.

```
//DUMP     EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DSN=ANNE.WATCHERS.DATA.DUMP02,
//            DISP=(NEW,CATLG),
//            SPACE=(CYL,(40,5),RLSE),
//            UNIT=SYSDA
//SYSIN    DD *
  DUMP DATASET (INCLUDE(ANNE.WATCHERS.**)) -
       OUTDDNAME(ENCFILE) -
       RSA(WATCHERS.1024.PUBLIC.KEY) -
       ENCRYPT(CLRTDES) -
       HWCOMPRESS
/*
```

Figure 8-12 JCL to encrypt public key

5. Anne sends the encrypted data to Gwen. Gwen decrypts it with the JCL shown in Figure 8-13.

```
//RESTORE  EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DISP=SHR,DSN=GWEN.WATCHERS.DATA.DUMP02
//DECFILE  DD DISP=(NEW,CATLG),
//            UNIT=SYSDA,
//            SPACE=(CYL,(50,5),RLSE)
//SYSIN    DD *
  RESTORE DATASET (INCLUDE(**)) -
             INDD(ENCFILE) -
             OUTDDNAME(DECFILE) -
             CATALOG -
             RSA(WATCHERS.MAGICBOX.1024.KEY)
/*
```

Figure 8-13 JCL to decrypt, invoking the RSA private key

8.7.4 Scenario 4: DFSMSdss with RSA and RACDCERT

The Magic Box Company sends some sensitive data to the fictitious Hyena Corp. An RSA key protects the data key. Alexander Harris of Hyena Corp uses RACDCERT to generate the RSA key pair. Alexander has a z/OS V1.7 with HCR7720 system running on a z990 with two CEX2Cs. Anne Summers of the Magic Box a z/OS V1.7 system with ICSF FMID HCR7720 running on a z9 platform with two CEX2C cards.

All the preceding RACF RACDCERT commands were issued in batch mode with the JCL shown in Example 8-3.

Example 8-3 JCL used to issue RACDCERT commands

```
//STEP1 EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
   <put RACDCERT commands here>
/*
```

Tip: A common practice is to issue the RACDCERT commands in batch mode rather than in the TSO foreground interactive mode. The commands are easier to type in. The output from the command can also be viewed easily with your favorite JOBLOG viewing tool such as SDSF or EJES.

The scenario proceeds as follows:

1. Alexander creates an RSA key pair using RACF. He gets it signed by an external trusted certificate authority (CA):

   a. Alexander creates a self-signed certificate with a 2048-bit key using the RACDCERT command, as shown in Figure 8-14. The certificate is called RSA2048 Magic Box to Hyena. The RSA key label in the PKDS is RSA.2048.MAGICBOX.2.HYENA.

      The certificate labels are case-sensitive and can include blanks. The ICSF key labels are in uppercase and cannot contain any blanks.

```
RACDCERT ID(ALEX) -
          GENCERT  -
         SUBJECTSDN(CN('Magic Box to Hyena') -
                    O('Hyena') -
                    L('Sunnydale') -
                    SP('CA') -
                    C('US')) -
          WITHLABEL('RSA2048 Magic Box to Hyena') -
          SIZE(2048) -
          PCICC(RSA.2048.MAGICBOX.2.HYENA)
```

Figure 8-14 RACDCERT command to create self-signed certificate

   b. Alexander creates a certificate request to send to an external CA. The request is in a data set called ALEX.CERT.REQ.ARM. The RACDCERT command Alexander uses is shown in Figure 8-15.

```
RACDCERT ID(ALEX) -
        GENREQ (LABEL('RSA2048 Magic Box to Hyena')) -
        DSN('ALEX.CERT.REQ.ARM')
```

Figure 8-15 RACDCERT command to generate a certificate request

   c. Alexander sends the certificate request in the data set ALEX.CERT.REQ.ARM to the CA. The certificate request is in Base 64-encoded text. Alexander sends the certificate request in an e-mail to the CA as a text attachment. The CA validates the certificate and signs it. The CA returns it to Alexander. Alexander transfers the certificate back to the host into the data set ALEX.SIGNED.CER.

d. Alexander replaces the self-signed certificate with the certificate signed by the CA. The RACDCERT command Alexander uses is shown in Figure 8-16.

```
RACDCERT ID(ALEX) -
         ADD('ALEX.SIGNED.CER') -
         WITHLABEL('RSA2048 Magic Box to Hyena')
```

Figure 8-16 RACDCERT command to import signed certificate into RACF

> **Important:** We assume here that the certificate authority's certificate has been already installed in the RACF database as a CERTAUTH certificate. RACF then verifies the newly added certificate for the validity of the issuer's digital signature. If the certificate authority's certificate is not already installed in the RACF database, you must manually put the added certificate in the TRUST status in the RACF database.

e. Alexander lists the certificate with the RACDCERT command, as shown in Figure 8-17, to make sure that the certificate is good.

```
RACDCERT ID(ALEX) -
         LIST (LABEL('RSA2048 Magic Box to Hyena'))
```

Figure 8-17 RACDCERT command to list the certificate

2. Alexander exports the public key to a data set to send to Anne. Figure 8-18 shows the RACDCERT command. The certificate data set is in PKCS#7 format encoded in Base 64. The data set is called ALEX.MAGIC.PUBLIC.CER.

```
RACDCERT ID(ALEX) EXPORT -
         EXPORT (LABEL('RSA2048 Magic Box to Hyena')) -
         DSN('ALEX.MAGIC.PUBLIC.CER') -
         FORMAT(PKCS7B64)
```

Figure 8-18 RACDCERT command to export the public key into a certificate

3. Anne receives the certificate as an e-mail attachment from Alexander. She uses FTP to upload the certificate to the host. Because the certificate is Base 64 encoded, she uploads the certificate in ASCII mode.

4. Anne imports Alexander's certificate as a SITE certificate. The RACDCERT command is shown in Figure 8-19. The certificate data set is called ANNE.HYENA.PUBLIC.CER. The certificate label is Hyena Corp 2048 Public Key. The key label in the PKDS is HYENA.2048.PUB.KEY.

```
RACDCERT SITE ADD -
         ('ANNE.HYENA.PUBLIC.CER') -
         WITHLABEL('Hyena Corp 2048 Public Key') -
         PCICC('HYENA.2048.PUB.KEY')
```

Figure 8-19 RACDCERT command to import the public key into the PKDS

Important: We do not recommend that you associate the certificate with a user ID for a specific individual. From a logical standpoint, a certificate and the public key used by Encryption Facility is not a personal certificate. Making it a SITE certificate makes more sense. From a practical standpoint, if the certificate is associated with an individual and the person leaves the company, you have to be careful in the method you use to delete the person's resources. If you delete the user ID, the certificate will remain in place. If IRRRID00 is run to remove all traces of the user ID from RACF, the certificate will be deleted. Certificate deletion will attempt to remove the corresponding RSA keys from the PKDS.

5. Anne encrypts the data with the JCL shown in Figure 8-20. Anne sends the data over to Alexander.

```
//DUMP     EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DSN=ANNE.HYENA.DATA.DUMP01,
//            DISP=(NEW,CATLG),
//            SPACE=(CYL,(40,5),RLSE),
//            UNIT=SYSDA
//SYSIN    DD *
  DUMP DATASET (INCLUDE(ANNE.HYENA.**)) -
       OUTDDNAME(ENCFILE) -
       RSA(HYENA.2048.PUB.KEY) -
       ENCRYPT(ENCTDES) -
       HWCOMPRESS
/*
```

Figure 8-20 JCL to encrypt using the HYENA.2048.PUB.KEY RSA key

6. Alexander receives the data from Anne. Alexander decrypts the data with the JCL shown in Figure 8-21.

```
//RESTORE  EXEC PGM=ADRDSSU
//SYSPRINT DD SYSOUT=*
//ENCFILE  DD DISP=SHR,DSN=ALEX.HYENA.DATA.DUMP01
//DECFILE  DD DISP=(NEW,CATLG),
//            UNIT=SYSDA,
//            SPACE=(CYL,(50,5),RLSE)
//SYSIN    DD *
  RESTORE DATASET (INCLUDE(**)) -
             INDD(ENCFILE) -
             OUTDDNAME(DECFILE) -
             CATALOG -
             RSA(RSA.2048.HYENA.2.MAGICBOX)
/*
```

Figure 8-21 JCL to decrypt using RSA.2048.MAGICBOX.2.HYENA private key

9

Using the Encryption Facility Java Client on z/OS

In this chapter, we describe our experience installing and using the Encryption Facility for z/OS Java Client. For the sake of simplicity, we refer to the “Java Client” in the rest of the chapter.

The Java Client can be installed on any platform that offers Java support, provided the following prerequisites are met:

- On z/OS:
  - IBM SDK for z/OS, Java 2 Technology Edition, 5655-I56, at PTF UQ90449 or later (SDK1.4.2)
  - IBM Developer Kit for OS/390, Java 2 Technology Edition, 5655-D35, at PTF UQ88094 or later (SDK1.3.1)
- On other platforms than z/OS:
  - Sun SDK 5.0
  - IBM JVM at SDK1.4.2
  - JVM with a JCE cryptographic provider installed that supports all the required algorithms

There is a special set of information, including PTF requirements for iSeries and other platforms, provided in the *readme* file in the downloadable package of the

Encryption Facility for z/OS Java Client. See 9.1.1, “Downloading the Java Client” on page 127 for further information about the Java Client download.

In this chapter, we use the Java Client on the z/OS JVM. Other Java-capable platforms require a similar setup.

9.1 Getting the Java Client installed and working

This section describes the installation process for the Java Client on z/OS and relates our own experience when installing and testing this Encryption Facility for z/OS feature.

9.1.1 Downloading the Java Client

We downloaded the Java Client from:

http://www.ibm.com/servers/eserver/zseries/zos/downloads/#efclient

The files are zipped and we received them on a PC where they were unzipped. The structure of the set of files after unzipping is shown in Figure 9-1.

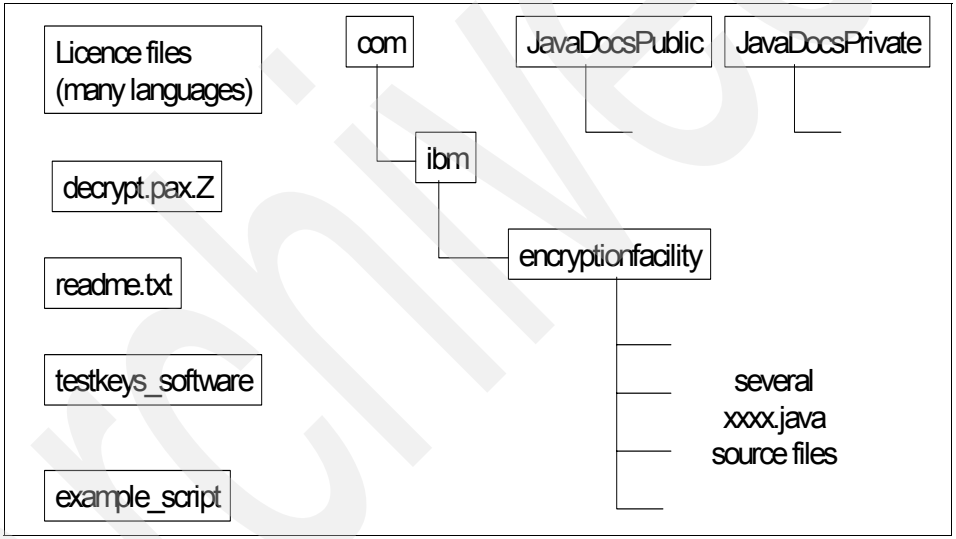

Figure 9-1 Unzipped Java Client files

Note that all source and documentation files are ASCII encoded. Therefore, you can browse them using PC text editors.

The downloaded files are:

- readme.txt
  A description of the package contents and the installation instructions along with service level recommendations.
- Many files written in different languages that specify the license agreement associated with the IBM Encryption Facility Java Client.

- example_script
  An example of how to invoke the execution of the IBM Encryption Facility Java Client.
- testkeys_software
  A JKS keystore that comes populated with an RSA private/public key pair.
- com
  A directory that contains the source code for the IBM Encryption Facility Java Client.
- JavaDocsPrivate
  A directory that contains the documentation pertaining to all classes used in the Java Client. The documents are in HTML and readable through a Web browser.
- JavaDocsPublic
  A directory that contains the documentation of the public interfaces for the Java Client classes. The documents are in HTML and readable through a Web browser.

9.1.2 Transferring the Java Client source to z/OS

The Java Client class files have to be compiled before being used. The Java classes source files have the extension .java and, in our case, have to be transferred to z//OS where they will be compiled.

In our z/OS, we created the following UNIX file system path and transferred the .java files from the PC to z/OS into the encryptionfacility directory:

```
/u/itsopk/encryptionclient/ibm/com/encryptionfacility
```

We used FTP in text mode because the source file must be in EBCDIC in z/OS. The z/OS FTP server performs the ASCII to EBCDIC translation of the text into the files as they are transferred from the PC.

Note: Our z/OS FTP server did not have the correct option set in the FTPDATA data set to handle the end-of-line character in the Java source files (restoring the source files as one single long line in the z/OS UNIX file).

This was corrected by adding the following to the FTPDATA parameters so that correct code page 1047 conversion was produced by the FTP server:

```
SBDATACONN  (IBM-1047,IBM-850)
```

We also transferred the example_script file to /u/itsopk/encryptionclient as a text file and the testkeys_software as a binary file.

9.1.3 Compiling the Java Client on z/OS

We assume that the JVM has already been installed and is operating properly. The minimum check that we recommend you to do is to request the JVM characteristics by the **`java -version`** command. To do so, go in to the z/OS UNIX shell (with the TSO OMVS command) and enter **`java -version`**. The expected output looks like what is shown in Figure 9-2.

```
 - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ITSOPK: /u/itsopk>java -version
java version "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM z/OS Persistent Reusable VM
build cm142-
20050929 (SR3) (JIT enabled: jitc))
```

Figure 9-2 Checking Java with the java -version command

Note: The first time we tried the **`java -version`** command on our system, it failed with messages about a segmentation violation and an abend 0C4. It appears that there is requirement to run in a TSO region of at least 128 MB to run Java code. We had a smaller region size, and increasing it solved the problem.

To compile the Java source, you must be within the “root” directory of where you received the com.ibm.encryptionfacility Java package. In our case (see the following note, we had to be in /u/itsopk/encryptionclient. Then, you issue the command:

```
javac com/ibm/encryptionfacility/*.java
```

Note: The Java code is delivered in a Java “package” which is here com.ibm.encryptionfacility (you can see the “package” statement in the downloaded Header.java file). We built the z/OS UNIX file path to receive the source files so that the last levels of our file path (com/ibm/encryptionfacility/) are identical to the Java package.

The root directory designates the directory just above the package, in our case, encryptionclient.

The compilation takes a few seconds and does not issue a message when successful. The results of the compilation can be seen because each .java file

that we downloaded is now complemented with a .class file with the same file name. The .class files are the compiled Java code.

9.1.4 Locating the Java Client class files

You need to specify the file path where the .java classes that you compiled can be found. Probably simplest way to specify it, if you are not, yet, a Java literate, is through the CLASSPATH environment variable. CLASSPATH must indicate the path to follow from the UNIX file tree root down to the directory just above the package. In our case, the CLASSPATH is specified by issuing the following command in the z/OS UNIX shell, or by adding it to our .profile file:

```
export CLASSPATH=$CLASSPATH:.:/u/itsopk/encryptionclient
```

This means that the value of CLASSPATH is the concatenation of:

- Whatever CLASSPATH values were already specified for this environment
- The current path and directory (“.”)
- The path to the Java package that we just compiled

Note: When called for execution, the Java classes have their file name prefixed with the package name. Therefore, the CLASSPATH must also end where the package name begins.

9.1.5 Getting the Java Client to work

You can obtain a command help list from the Java Client by typing in the z/OS UNIX shell:

```
java com.ibm.encryptionfacility.EncryptionFacility
```

EncryptionFacility is the Java class that handles the work request; what precedes it is again the package name.

We should have provided arguments with this command, because we did not, the help list opens, as shown in Figure 9-3.

```
ITSOPK:/u/itsopk>java com.ibm.encryptionfacility.EncryptionFacility

Usage:
java EncryptionFacility
-mode encrypt
       -underlyingKey <AES16,3DES,PBEWithSHA1And3DES,or
PBEWithSHA1AndAES16>
      -keyStoreName <Keystore Name>
      -keyStoreType <Keystore type>
     -keyStoreCertificateAlias <A Certificate Alias>
      -password <Password>
      -outputFile <File Name>
      -inputFile <File Name>
      -description <File Description>
      -iterations <Number Of Iterations>
      -recordFormat <UNDEFINED, FIXED>
      -recordSize <Fixed Record Size>
-mode decrypt
      -info
       -keyStoreName <Keystore Name>
-keyStoreType <Keystore type>
     -keyStoreCertificateAlias <A Certificate Alias>
      -password <Password>
      -outputFile <File Name>
      -inputFile <File Name>

* If using a AES16 or 3DES key -keyStoreName,-keyStoreType, and
-keyStoreCertificateAlias arguments are required
* If the -password argument is not specified, you will be prompted
for a passwor
d
* If using FIXED record format, the -recordSize argument is required
Enter password:
```

Figure 9-3 The help list displayed by the client

For more information regarding these arguments, refer to the JavaDocsPublic/com/ibm/encryptionfacility/EncryptionFacility.html file downloaded on your PC.

9.1.6 Using the example_script shell script

The example_script provided in the downloaded files can be used to check the proper installation of the Java Client. It initially comes as shown in Example 9-1.

Example 9-1 example_script

```
java -Djava.encryption.facility.debuglevel=0 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode encrypt \
-password "ASK NOT FOR WHOM THE BELL TOLLS" \
-underlyingKey PBEWithSHA1AndAES16 \
-iterations 1000 \
-inputFile data.dat \
-outputFile data.crypt
java -Djava.encryption.facility.debuglevel=0 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode decrypt \
-password "ASK NOT FOR WHOM THE BELL TOLLS" \
-inputFile data.crypt \
-outputFile data.clear
```

Note that the forward slash (\) is the continuation character for script lines.

What this example does:

- It calls the execution the EncryptionFacility class and also sets, through the -Djava option, the system property encryption.facility.debuglevel. The value given to the property here is “0”, meaning the least detailed tracing information.

  Note: The Java Client generates tracing information in case of failure that go into the file defined as stderr. The higher the value given to the encryption.facility.debuglevel property, the more detailed the tracing information is.

- It requests an encryption with an AES 16-byte key derived from a password.
- The key derivation process requires 1000 iterations.
- The clear text input file is the data.dat file in the current directory.
- The encrypted output file is the data.crypt file.

Then the script calls for a decryption of the previously encrypted file.

To run this example, you can store the text, modified if needed, in a shell script file, a file extension of .sh.

Here is what we did:

- We created the client_trial1.sh file in the /u/itsopk/ directory and stored the example_script text in this file.
- Before being stored, the text has been modified to store the result of the decryption process into the data1.dat file instead of data.clear (so that we can see if data has been created in the output file).
- We created the data.dat file in the /u/itsopk directory with the text to be encrypted.

The script can then be called, as shown in Figure 9-4, with the result of invoking the encryption and decryption services of the Java Client. A successful execution results in no messages being displayed; we just verified that data1.dat was identical to data.dat.

```
ITSOPK: /u/itsopk>

 ===> client_trial1.sh
```

Figure 9-4 Invoking the script

The -info argument

We also tested the -info argument, which lists the header of an encrypted file; that is, the client does not proceed with the decryption of the file but provides the information stored with the file. To do so, we created the infoshell.sh script shown in Figure 9-5. Note that in order to collect the output into a file, instead of having it displayed in the UNIX shell, we “piped” (the > character in the last line of the command) the stdout device to the ef.log file.

The resulting contents of ef.log is shown in Figure 9-6 on page 134.

```
java -Djava.encryption.facility.debuglevel=0 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode decrypt \
-password "ASK NOT FOR WHOM THE BELL TOLLS" \
-info \
-underlyingKey PBEWithSHA1AndAES16 \
-iterations 10 \
-inputFile data.crypt > ef.log
```

Figure 9-5 infoshell.sh script

```
IBM Encryption Facility header information
Eyecatcher:
0000: C8 C5 C1 C4 C5 D9                                  HEADER
Version of Header record:
0000: 00 01                                              ..
Description:
0000: C9 C2 D4 40 C5 95 83 99 A8 97 A3 89 96 95 40 C6 IBM.Encryption.F
0010: 81 83 89 93 89 A3 A8 40 C5 95 83 99 A8 97 A3 85 acility.Encrypte
0020: 84 40 C6 89 93 85 00 00 00 00 00 00 00 00 00 00 d.File..........
0030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Salt value for PBE:
0000: 30 17 46 54 41 2B 4A FA                            ..ãè....
Iteration count for PBE:
0000: 00 00 00 0A                                        ....
Modulus length in bits of the RSA key:
0000: 00 00                                              ..
Label of the RSA key:
0000: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 ................
0010: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 ................
0020: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 ................
0030: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 ................
CIPHER initial chaining vector:
0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Requested block factor:
0000: 00 00 00 00                                        ....
AES/Clear DES in use
  X'01 Clear TDES
X'01 Clear TDES
  X'02 Clear AES128
  X'03 Encrypted TDES
  X'04 Encrypted AES128:
0000: 02                                                 .
Flags
  Bit 0 - Input was tape: false
  Bit 1 - Encrypted data was compressed before encryption: false
  Bit 2 - Compression dictionary: false
  Bit 3 - Format for the data, 0 for text 1 for binary: true:
0000: 10                                                 .
Version of compression:
0000: 00 00                                              ..
Input Record Format
  Bit 0 = '1'b, Bit 1 = '0'b: Fixed
  Bit 0 = '0'b, Bit 1 = '1'b: Variable
  Bit 0 = '1'b, Bit 1 = '1'b: Undefined
  Bit 2 = '1'b: Blocked records
  Bit 3 = '1'b: ASA control character
... being continued
```

Figure 9-6 The -info output

9.2 Invoking the Java Client through JCL

The shell program that invokes the Java Client can itself be invoked by JCL using the BPXBATCH program. In that case, the STDIN and STDOUT DD cards in the JCL can be used to replace the -inputFile and -outputFile arguments, respectively.

> **Note:** The -inputFile argument has still to be specified as -inputFile /dev/fd0. From our experiences, we noted that when the -inputFile is not specified as shown earlier, the Java Client converts the input data from EBCDIC to ASCII.

For example, you can invoke the encryption of our example file with JCL by calling the shell program, encbatch.sh, shown in Figure 9-7.

```
java -Djava.encryption.facility.debuglevel=0 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode encrypt \
-password "ASK NOT FOR WHOM THE BELL TOLLS" \
-underlyingKey PBEWithSHA1AndAES16 \
-iterations 1000
-inputFile /dev/fd0
```

Figure 9-7 Encryption shell script (encbatch.sh) modified for JCL invocation

The JCL to be used is shown in Figure 9-8. Notice the STDERR DD definition: All tracing and error information goes into the error.log file.

```
//EFCLIENT JOB 1,RACFUSER,TIME=1440,NOTIFY=&SYSUID,REGION=0M,
//            CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//*****************************************************************
//* THIS JOB EXERCIZES THE ENCRYPTION FACILITY JAVA CLIENT          *
//*                  ON Z/OS                                        *
//*****************************************************************
//SHELL EXEC PGM=BPXBATCH,REGION=0M,
// PARM='SH /u/itsopk/eftrials/encbatch.sh'
//STDIN DD PATH='/u/itsopk/data.dat',
// PATHOPTS=(ORDONLY),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//STDOUT DD PATH='/u/itsopk/data.crypt',
// PATHOPTS=(OWRONLY,OCREAT),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//STDERR DD PATH='/u/itsopk/error.log',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

Figure 9-8 JCL to invoke the Java Client encryption script

Decryption is performed by invoking the shell program, decbatch.sh, in Figure 9-9 with the JCL shown in Figure 9-10 on page 137.

```
java -Djava.encryption.facility.debuglevel=0 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode decrypt \
-password "ASK NOT FOR WHOM THE BELL TOLLS" \
-iterations 1000 \
-inputFile /dev/fd0
```

Figure 9-9 Decryption shell script (decbatch.sh) modified for JCL invocation

```
//EFCLIENT JOB 1,RACFUSER,TIME=1440,NOTIFY=&SYSUID,REGION=0M,
//             CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//*********************************************************************
//* THIS JOB EXERCIZES THE ENCRYPTION FACILITY JAVA CLIENT            *
//*                    ON Z/OS                                        *
//*********************************************************************
//SHELL EXEC PGM=BPXBATCH,REGION=0M,
// PARM='SH /u/itsopk/eftrials/decbatch.sh'
//STDIN DD PATH='/u/itsopk/data.crypt',
// PATHOPTS=(ORDONLY),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//STDOUT DD PATH='/u/itsopk/data.dat',
// PATHOPTS=(OWRONLY,OCREAT),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//STDERR DD PATH='/u/itsopk/error.log',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

Figure 9-10 JCL to invoke the Java Client decryption script

9.3 Using RSA protected data keys with the Java Client

In this section, we explain how to set up the Java Client environment so that RSA keys and certificates can be used to protect the data-encrypting keys.

We use the Java “keytool” utility to generate and maintain RSA keys and certificates that the Java Client uses. Those of you already familiar with the gskkyman utility of z/OS System SSL will find many similarities between the two utilities.

For details about the use of the Java keytool utility, see:

http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html

9.3.1 Java implementation of keys storage

Java uses the concept of a “keystore.” The default implementation of keystores, provided by Sun Microsystems™, uses a proprietary keystore type (a format) named “JKS.”

Each private key in the keystore is protected by an individual password (the private key password), and access to the entire keystore is also protected with a

password (the store password), which can be the same as the private key password.

The keytool enables users to administer their own public/private key pairs and associated certificates, as they reside in the keystore.

Keystore contents and terminology

This section provides a brief overview of the keystore contents and terminology.

Keystore entries

There are two types of entries in a keystore:

- The *key entries*, which hold a private key and the corresponding certificate, the latter usually stored as a “certificate chain,” typically made of two certificates: the user certificate that contains the public key corresponding to the private key also stored in the keystore entry, and the self-signed certificate authority certificate that signed the user certificate. If intermediate certificate authorities are involved, they are also included in the stored certificate chain.
- The *trusted certificate entries*, which hold certificate, that is, the public key without the private key. Typically, these are certificates of partners that are being trusted by the keystore owner.

Keys and certificates entries are given a label, an *alias* in the Java keystore terminology.

Keystore location

The keystore location is the name of the persistent keystore file that hosts the keystore managed by keytool. Keystores are by default located in the user’s home directory with a file name of .keystore. The -keystore option of keytool enables the user to give a specific name to a keystore.

The cacerts Certificates File

The cacerts Certificates File is a system-wide keystore that comes already populated with well-known certificate authority self-signed certificates. It can also be managed with keytool.

9.3.2 Using keytool

keytool is designed to process data that resides in file-based keystores (Java allows keystores to be implemented in ways other than using persistent files).

Note: As default values, keytool uses values specified in the Java security properties file called java.security. However, because it might be difficult for less-experienced Java users to fully apprehend the use of this property file, we recommend explicitly specifying all parameters values along with the keytool commands.

keytool commands

Again, for more details about the keytool commands, see:

http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html

We focus on the use of keytool in the context of the Encryption Facility Java Client, and we, therefore, do not address all the possibilities offered by the utility.

The utility commands that are relevant to the use of the Java Client are:

-genkey This command generates an asymmetric key pair. The public key is wrapped into a self-signed certificate and is stored, along with the private key, in a new keystore entry labeled with the value of the -alias option.

> **Important:** By default, **-genkey** generates DSA keys. If RSA keys are desired, as it is the case with the Encryption Facility for z/OS, the keyword **-keyalg "RSA"** must be specified.

-import This command imports (reads) into a keystore entry certificates or a certificate chain that is stored in a file external to the keystore.

-selfcert With this command, the user can generate a self-signed certificate to be exported later.

> **Note:** The certificate signature's algorithm is derived from the algorithm of the underlying private key. If the key pair has been generated with the RSA algorithm, the certificate is signed with the RSA algorithm.

Note that a new keystore is created every time that the commands **-genkey**, **-import**, or **-identifydb** refer to a keystore that does not exist yet.

9.3.3 Example: Creating the Java Client RSA key pair

In preparation for the use of RSA keys, an RSA key pair and certificate are created with the keytool utility as shown in Figure 9-11. Note that an entry “javarsa” has been created in the keystore. The keystore itself is also created by the command, and because no specific name is provided in the command, it is created as the file .keystore in the requestor’s home directory.

The key entry and the keystore itself are given the same password, secretpassword.

```
ITSOPK: /u/itsopk>keytool -genkey -dname "CN=java test cert,
OU=itso, O=ibm, C=us" -keyalg "rsa" -alias javarsa
Enter keystore password:  secretpassword
Enter key password for <javarsa>
      (RETURN if same as keystore password):
ITSOPK: /u/itsopk>
```

Figure 9-11 Creating an RSA key pair and certificate

9.3.4 Modifying the encbatch.sh and decbatch.sh scripts for RSA protection of the data key

The encbatch.sh script has been modified as shown in Figure 9-12.

```
java -Djava.encryption.facility.debuglevel=1 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode encrypt \
-keyStoreName .keystore \
-keyStoreType jks \
-keyStoreCertificateAlias javarsa \
-password secretpassword \
-description "java RSA test" \
-inputFile /dev/fd0
```

Figure 9-12 Encryption script modified for RSA protection of the data key

In addition, the decryption script, decbatch.sh, has been modified as shown in Figure 9-13.

```
java -Djava.encryption.facility.debuglevel=1 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode decrypt \
-keyStoreName .keystore \
-keyStoreType jks \
-keyStoreCertificateAlias javarsa \
-password secretpassword \
-inputFile /dev/fd0
```

Figure 9-13 Decryption script modified for RSA protection of the data key

9.3.5 Example: Java Client to/from Encryptions Services

In this scenario, encrypted files are exchanged between the Java Client and the Encryption Services feature of the Encryption Facility for z/OS. The data encrypting key is protected by an RSA key.

We use the RSA key pair (javarsa) already generated for the Java Client. We now create the key pair to be owned by the Encryption Services.

Creating the Encryption Services key pair with RACF

We elected to use RACF to create the z/OS side RSA key pair and certificate. The RACDCERT command with the GENCERT keyword is used, as shown in Figure 9-14 on page 142, with the following results:

- An RSA key pair is created. Because of the PCICC keyword, the key pair, private and public keys, is kept in the PKDS with the specified PKDS label zoskeys.
- A self-signed certificate that contains the public key of the key pair is stored in the RACF database with the label z/OS keys.

Note: In the RACDCERT command examples, we assume that the user in charge of setting up the keys and certificate is EFUSER, by this implying that EFUSER has all the RACF permissions required for the RACDCERT options to work.

```
 ===> racdcert id(efuser) gencert subjectsdn(cn('zos
side')ou('itso')o('ibm')c('us')) withlabel('z/OS keys')
pcicc(zoskeys) keyusage(dataencrypt)
```

Figure 9-14 Creating the Encryption Services RSA key pair and certificate

Exporting the Encryption Services certificate

Figure 9-15 shows how the RACDCERT EXPORT command exports the Encryption Services certificate (the Encryption Services public key) as a binary X.509 V3 certificate. The certificate is stored in the data set EFUSER.TOJAVA.CER, which can then be transmitted to the Java Client.

```
racdcert id(efuser) export(label('z/OS keys')) dsn(tojava.cer)
format(certder)
```

Figure 9-15 Preparing the Encryption Services certificate for transmission

EFUSER.TOJAVA.CER is sent through FTP, in binary mode, to the Java Client system, where it is stored into fromzos.cer file.

Installing the Encryption Services public key in Java keystore

After the Encryption Services certificate has been received in the fromzos.cer file, it can then be installed in the keystore with the keytool utility, as shown in Figure 9-16, where it is given in the example a label (an alias) of zos_public_key.

```
ITSOPK: /u/itsopk/eftrials>keytool -import -alias zos_public_key
-file fromzos.cer
Enter keystore password:  secretpassword
Owner: CN=zos side, OU=itso, O=ibm, C=us
Issuer: CN=zos side, OU=itso, O=ibm, C=us
Serial number: 0
Valid from: 09/2/06 5:00 AM until: 09/3/07 4:59 AM
Certificate fingerprints:
     MD5:  4C:C4:41:11:51:EE:14:47:2B:E4:24:45:C0:6B:1B:AC
        SHA1:
8B:EF:A1:7A:9F:BC:E8:C5:B7:EA:17:4E:19:2C:1F:C0:28:39:D9:5D
Trust this certificate? [no]:  yes
Certificate was added to keystore
ITSOPK: /u/itsopk/eftrials>
```

Figure 9-16 Installing the Encryption Services certificate in the keystore

We now have two sets of keys in our keystore. We can invoke keytool to get a list of the keystore contents, as shown in Figure 9-17.

```
ITSOPK: /u/itsopk>keytool -list -keystore .keystore
Enter keystore password:  secretpassword

Keystore type: jks
Keystore provider: IBMJCE

Your keystore contains 2 entries

zos_public_key, Sep2, 2006, trustedCertEntry,
Certificate fingerprint (MD5):
4C:C4:41:11:51:EE:14:47:2B:E4:24:45:C0:6B:1B:AC
javarsa, Sep2, 2006, keyEntry,
Certificate fingerprint (MD5):
A7:26:C1:DD:6B:39:0B:24:56:86:19:FC:54:27:A5:9A
```

Figure 9-17 Listing the keystore contents

Exporting the Java Client certificate

The Java Client certificate is exported from the keystore into the tozos.cer file, as shown in Figure 9-18.

```
ITSOPK: /u/itsopk>keytool -export -alias javarsa -file tozos.ce
Enter keystore password:  secretpassword
Certificate stored in file <tozos.cer>
```

Figure 9-18 Exporting the Java Client certificate

The certificate is transmitted through FTP, in binary mode, to the Encryption Services z/OS host. It is received in the fromjava.cer file.

Installing the Java Client certificate into the RACF database

The RACDCERT ADD command installs the certificate into the RACF database, as shown in Figure 9-19 on page 144. The certificate is designated in the RACF database with the label “java public key” and, because of the PCICC keyword, the public key is stored in the PKDS with a label “javakey.”

Note also that we force RACF to give the TRUST status to the certificate because the certificate authority (actually this is a self-signed certificate) is not known in the RACF database.

```
 ===> racdcert id(efuser) add(fromjava.cer) trust withlabel('java
public key') pcicc(javakey)

IRRD119I certificate authority not defined to RACF.  Certificate
added with TRUST status.
 ***
```

Figure 9-19 Installing the Java Client certificate in the RACF database

9.3.6 Encryption Services to Java Client using CLRTDES with RSA

In this section, we describe how to use the Java Client triple DES encryption of data with RSA protection of the encryption key.

Encryption Services to the Java Client

The CSDFILENC program starts with the JCL shown in Figure 9-20.

When encrypted, the data is sent to the Java Client either physically or electronically, such as with FTP (in binary mode).

```
//JOBNMEN  JOB (ACCT,3C1),'ITSO',CLASS=A,
//         MSGCLASS=X,MSGLEVEL=(1,1)
//*
//ENRS     EXEC PGM=CSDFILEN
//SYSUT1   DD DSN=ITSOPK.TOJAVA.CLEAR,DISP=SHR
//SYSUT2   DD DSN=ITSOPK.TOJAVA.ENCR,DISP=(,CATLG),
//         UNIT=SYSDA,SPACE=(1024,(20,5)),AVGREC=K
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
DESC='z/OS to java with rsa key
CLRTDES
RSA=javakey
/*
```

Figure 9-20 JCL to invoke CSDFILENC with the Java Client public key

The execution of CSDFILEN yields the statistics report shown in Figure 9-21.

```
 CSDFILEN Encryption Utility 09/02/2006 (MM/DD/YYYY) 12:11:17 (HH:MM:SS)
INPUT:  DESC=Z/O TO JAVA WITH RSA KEY
INPUT:  RSA=JAVAKEY
 CSDFILEN:    RSA-PUB : JAVAKEY
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR     TDES KEY USING CPACF
  RECORDS READ:              1  WRITTEN:              1
  BYTES READ:                80
  BYTES WRITTEN:               560  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000003 DATA:       96 LOW:    0.000003 DAT
 TOTAL CIPHER TIME (IN SECONDS):     0.000003  CIPHERS:             1
  TOTAL ELAPSED TIME:   0:00:00.20
```

Figure 9-21 Encryption of the data to the Java Client

Decrypting at the Java Client

The Java Client decryption is invoked with the script shown in Figure 9-22, which is itself started with the JCL in Figure 9-23 on page 146. The STDIN DD card points at the encrypted data received from the Encryption Services (fromzos.enc).

```
java -Djava.encryption.facility.debuglevel=1 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode decrypt \
-keyStoreName .keystore \
-keyStoreType jks \
-keyStoreCertificateAlias javarsa \
-password secretpassword \
-inputFile /dev/fd0
```

Figure 9-22 Java Client decryption script decbatch.sh

```
//EFCLIENT JOB 1,RACFUSER,TIME=1440,NOTIFY=&SYSUID,REGION=0M,
//            CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//SHELL EXEC PGM=BPXBATCH,REGION=0M,
// PARM='SH /u/itsopk/eftrials/decbatch.sh'
//STDIN DD PATH='/u/itsopk/eftrials/fromzos.enc',
// PATHOPTS=(ORDONLY),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//STDOUT DD PATH='/u/itsopk/eftrials/fromzos.clr',
// PATHOPTS=(OWRONLY,OCREAT),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//STDERR DD PATH='/u/itsopk/eftrials/error.log',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

Figure 9-23 Invocating the Java Client decryption script

9.3.7 Java Client to Encryption Services using AES128 with RSA

In this section, we describe how to use the Java Client AES encryption of data with RSA protection of the encryption key.

Java Client to Encryption Services

The Java Client encryption is invoked through the script shown in Figure 9-24. The script itself is started using the JCL shown in Figure 9-25 on page 147. The tozos.enc file is sent to the Encryption Services either physically or electronically such as with FTP (in binary mode).

```
java -Djava.encryption.facility.debuglevel=1 \
com.ibm.encryptionfacility.EncryptionFacility \
-mode encrypt \
-underlyingKey AES16 \
-keyStoreName .keystore \
-keyStoreType jks \
-keyStoreCertificateAlias zos_public_key \
-password secretpassword \
-description "java to z/OS with RSA key" \
-inputFile /dev/fd0
```

Figure 9-24 Java Client script encbatch.sh for AES128 with RSA encryption

```
//EFCLIENT JOB 1,RACFUSER,TIME=1440,NOTIFY=&SYSUID,REGION=0M,
//            CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//SHELL EXEC PGM=BPXBATCH,REGION=0M,
// PARM='SH /u/itsopk/eftrials/encbatch.sh'
//STDIN DD PATH='/u/itsopk/eftrials/tozos.clr',
// PATHOPTS=(ORDONLY),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//*TDOUT DD DUMMY
//STDOUT DD PATH='/u/itsopk/eftrials/tozos.enc',
// PATHOPTS=(OWRONLY,OCREAT),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//STDERR DD PATH='/u/itsopk/eftrials/error.log',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

Figure 9-25 JCL to invoke the encbatch.sh script

Decrypting at the Encryption Services

The fromjava.enc file is received from the Java Client and the Encryption Services are invoked to decrypt the data with the JCL shown in Figure 9-26.

```
//JOBNMDE  JOB (ACCT,3C1),'ITSO',CLASS=A,
//         MSGCLASS=X,MSGLEVEL=(1,1)
//*
//DECR     EXEC PGM=CSDFILDE
//SYSUT1 DD PATH='/u/itsopk/fromjava.enc',
// PATHOPTS=(ORDONLY),
// PATHMODE=SIRWXU,
// PATHDISP=KEEP
//SYSUT2   DD DSN=ITSOPK.FROMJAVA.CLR,DISP=(,CATLG),
//            SPACE=(1024,(10,10))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
RSA=zoskeys
/*
```

Figure 9-26 Encryption Services JCL to decrypt AES128 with RSA

Note: The encrypted file produced by the Java Client can also be FPTed to the Encryption Services z/OS host into an MVS data set. Our recommendation here is to preallocate the receiving data set with a RECFM of U, because reformatting and reblocking by the FTP server itself might cause truncation of records and thus the received file is rejected by CSDFILDE.

The execution of CSDFILDE yields the statistics report shown in Figure 9-27.

```
 CSDFILDE Decryption Utility 09/02/2006 (MM/DD/YYYY) 22:24:23
(HH:MM:SS)
 CSDFILDE:   HEADER VERSION :     1
 CSDFILDE:   RSA-PUB : zos_public_key
INPUT: DESC = java to z/OS with RSA key
INPUT: LRECL     0 BLKSIZE   32760 RECFM U
INPUT:  RSA=zoskeys
**WARNING** NEW OUTPUT BLKSIZE. REQUESTED:   27920
RECORDS READ:           1  WRITTEN:          1
  BYTES READ:           528  BYTES RECOVERED:          47
CIPHER TIMES (IN SECONDS): HIGH:   0.000109 DATA:   32304 LOW:
0.000109 DATA:
 TOTAL CIPHER TIME (IN SECONDS):     0.000109  CIPHERS:
1
  TOTAL ELAPSED TIME:   0:00:00.01
```

Figure 9-27 Statistics report for the decryption of the Java Client file

9.3.8 Sending a file encrypted with ENCTDES to the Java Client

We also encrypted a file with ENCTDES and RSA with CSDFILENC, using the JCL shown in Figure 9-28, which yielded the statistics report shown in Figure 9-29.

```
//JOBNMEN  JOB (ACCT,3C1),'ITSO',CLASS=A,
//         MSGCLASS=X,MSGLEVEL=(1,1)
//*
//ENRS     EXEC PGM=CSDFILEN
//SYSUT1   DD DSN=ITSOPK.TOJAVA.CLEAR,DISP=SHR
//SYSUT2   DD DSN=ITSOPK.TOJAVA.ENCR,DISP=(,CATLG),
//         UNIT=SYSDA,SPACE=(1024,(20,5)),AVGREC=K
//SYSPRINT DD SYSOUT=*
//SYSIN     DD   *
DESC='Z/OS TO JAVA - ENCTDES + RSA'
RSA=zoskeys
ENCTDES
/*
```

Figure 9-28 Encryption with ENCTDES and RSA

```
 CSDFILEN Encryption Utility 09/02/2006 (MM/DD/YYYY) 22:44:05
(HH:MM:SS)
INPUT:  DESC='Z/OS TO JAVA - ENCTDES + RSA'
INPUT:  RSA=zoskeys
INPUT:  ENCTDES
 CSDFILEN:   RSA-PUB : zoskeys
INPUT:  LRECL   80 BLKSIZE  27920 RECFM FB
OUTPUT: BLKSIZE  27998
ENCRYPTION OF DATA: ENCRYPTED TDES KEY USING CRYPTO COPROCESSOR
  RECORDS READ:            1  WRITTEN:            1
  BYTES READ:              80
  BYTES WRITTEN:            560  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000147 DATA:       96 LOW:
0.000147 DAT
 TOTAL CIPHER TIME (IN SECONDS):     0.000147  CIPHERS:
1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure 9-29 Statistics report for the encryption with ENCTDES and RSA

Then, the file was sent to the Java Client and the decryption script was invoked. The decryption failed with a return code 255. Looking into the STDERR file, we found the information shown in Figure 9-30 that demonstrates that ENCTDES is not supported by the Java Client.

```
java.io.IOException: Underlying key not supported, only clear 3DES,
clear AES16, PBEWithSHA1And3DES, and PBEWithSHA1AndAES16 keys are
supported.
```

Figure 9-30 Java Client error information when receiving a file encrypted with ENCTDES

A

Encryption Facility for z/OS: Features compatibility

This appendix summarizes the possible partnership combinations between features of the Encryption Facility for z/OS, along with related miscellaneous use considerations. The “encryption” column indicates which feature is used to encrypt the data and the “decryption” columns address the possible selections of features to perform the decryption.

Table A-1 Features compatibility

	Decryption			
Encryption	**z/OS DFSMSdss**	**z/OS CSDFILEN**	**z/OS Decryption Client**	**Java Client**
z/OS DFSMSdss	► DFSMSdss must be installed at both the encryption site and the decryption site. ► z/OS EF DFSMSdss Encryption feature must be installed at encryption site and decryption site. ► Encryption options chosen must be compatible. For example, if RSA keys are chosen, the keys must be capable of being handled at both sites.[a] ► Do not use AL label tapes for storing data.[b] ► Although support is supplied for DFSMShsm to make use of encryption through DFSMSdss, there is no support for the ABARS component of DFSMShsm.			

	Decryption			
Encryption	**z/OS DFSMSdss**	**z/OS CSDFILEN**	**z/OS Decryption Client**	**Java Client**
z/OS CSDFILEN		► z/OS EF Encryption Services must be installed at both sites. ► Cryptographic options chosen must be compatible. For example, if RSA keys are chosen, the keys must be capable of being handled at both sites. See note a. ► Do not use AL label tapes for storing data. See note b.	► z/OS EF Encryption Services must be installed at encryption site. ► Cryptographic options chosen must be compatible. For example, if RSA keys are chosen, the keys must be capable of being handled at both sites. See note a. ► Do not use AL label tapes for storing data. See note b.	► z/OS EF Encryption Services must be installed at encryption site. ► Do not use compression facilities in CSDFILEN. ► Cryptographic options chosen must be compatible. Note that the Java Client does not support ENCTDES. See note a. ► Suitable for situations where file transfer is already in place, and the record structure is either not important or already handled.[c] ► Do not use AL label tapes for storing data. See note b.

	Decryption			
Encryption	**z/OS DFSMSdss**	**z/OS CSDFILEN**	**z/OS Decryption Client**	**Java Client**
Java Client		► z/OS EF Encryption Services must be installed at the decryption site. ► Java Client does not support compression of data. ► Cryptographic options chosen must be compatible. For example, if RSA keys are chosen, the keys must be capable of being handled at both sites. See note a. ► Suitable for situations where file transfer is already in place, and the record structure is either not important or already handled. See note c. ► Do not use AL label tapes for storing data. See note b.	► z/OS EF Encryption Services must be installed at the encryption site. ► Java Client does not support compression of data. ► Cryptographic options chosen must be compatible. For example, if RSA keys are chosen, the keys must be capable of being handled at both sites. See note a. ► Suitable for situations where file transfer is already in place, and the record structure is either not important or already handled. See note c. ► Do not use AL label tapes for storing data. See note b. ► Refer to the license agreement in order to assess the legality of this configuration.	► z/OS EF Encryption Services must be installed at encryption site or the decryption site. ► Cannot use file compression. ► Cryptographic options chosen must be compatible. For example, if RSA keys are chosen, the keys must be capable of being handled at both sites. See note a for considerations related to RSA keys. ► Suitable for situations where file transfer is already in place, and the record structure is either not important or already handled. See note c. ► Do not use AL label tapes for storing data. See note b. ► Refer to the license agreement in order to assess the legality of this configuration.

a. RSA key lengths are an important point here. CCF cryptographic engines only provide support for processing RSA keys up to 1024 bits in length. If PCICC cryptographic processors are used, you need to ensure that you have the correct LIC level microcode. Another point is the support for AES keys. AES-128 has hardware no support on the System z9 environment only. For all other systems, AES encryption and decryption is run in software. If you want high performance at both encryption and decryption sites, you need to consider this.
b. AL tapes use ASCII labels. When these labels are used, the access method assumes that the data is EBCDIC and translates to ASCII using the XLATE service of DFSMSdfp™. Encrypted data will contain bytes potentially using every possible binary value and so is *not* EBCDIC in nature. Translating binary data t ASCII will destroy it.
c. When the Java Client is used, the record structure of the file can be lost if the correct steps are not taken. The Java Client treats the file as a byte stream, so this must be re-constituted into records of the correct size, using some process outside of the Encryption Facility Java Client.

B

Some encryption basics

This appendix provides:

- Some basics about encryption
- Examples to distinguish between symmetric and asymmetric encryption

This appendix also summarizes the concepts of:

- Digital signatures
- X.509 V3 certificates
- Certificate authorities
- Packages
- Common Cryptographic Architecture

Concepts

System z encryption capabilities exist within the wider context of encryption capabilities in other computing environments. The standards used within the System z environment are taken from many other standards used across those other platforms and architectures.

The terminology used is frequently platform-agnostic and can be found in the documentation supplied by various standards bodies.

What is encryption?

Encryption is the capability of hiding data such that its true form is not revealed unless we have special information. Usually in computing terms, we use this to mean that a “key” is provided to encrypt (hide) data or to decrypt (reveal) data.

Many encryption systems deal with two types of encryption:

- Symmetric encryption
- Asymmetric encryption

Each method of encryption makes use of a key. This is the “special information” that one might use to hide or reveal the true information. So what is the difference between these two methods of encryption?

Symmetric encryption

Symmetric encryption involves the encryption of data using a key. The same key is used to decrypt the information as was used to encrypt the information. Therefore, possession of that key grants the ability to access the information that was hidden with the same key.

Asymmetric encryption

Asymmetric encryption uses two keys. Data that is encrypted using key1 can be decrypted using key2. Data that is encrypted using key2 can be decrypted using key1.

What are the important characteristics of each method?

Asymmetric encryption provides the ability to hide some information and then allow someone else access to the information but not allow that person to hide information using the same key.

For example, the sender encrypts data using key 1 of the pair. The sender passes key 2 only to the receiver who is then able to decrypt that data. Now, assuming that key 1 is a secret key kept by the sender (typically called the sender's "private key"), then the receiver knows with certainty the origin of the data. Key 2, the other key in the pair, is the sender's "public key."

Now, if the receiver uses key 2 to encrypt his/her answer, then only the sender will be able to see it in clear because the sender is the only owner of key 1.

This demonstrates that asymmetric keys can be used to establish privacy of data and proof of origin of data. The receiver knows the sender must have written the data encrypted under key 1 because the sender is the only one to possess that key.

Note: With an asymmetric algorithm, the secret key (private key) is never to be transmitted; it always remains securely kept by its owner.

Symmetric encryption cannot be used in that way. If data is encrypted using a secret key, the same secret key is used to decrypt the data. Which rises the issue of transferring securely the secret key at the receiver's site.

The other important distinction is in the speed of operation. Asymmetric encryption is slow. It involves a very computationally intense sequence of operations. Symmetric encryption, however, is comparatively fast.

The larger the amount of data to be encrypted, the more this difference is noted. Encrypting large messages using asymmetric encryption is very slow.

In consequence, the two methods are frequently used together to provide different types of encryption capabilities that are both secure and fast.

How are asymmetric encryption keys organized?

When asymmetric encryption is used the keys are usually referred to as the "private key" and the "public key."

If a person has a pair of asymmetric keys, that person will keep her private key to herself. However, that person's public key can be published and kept in many places. This has two advantages.

Firstly, the person can publish data encrypted with her private key. This can only be decrypted with the corresponding public key. Because this public key is associated with this person, we can be sure that the communication came from this person.

Secondly, other individuals can encrypt data with this person's public key. After having done this, only this person can read it, because it requires the private key to decrypt the message.

Private keys are kept carefully private by storing them in a file protected by a password or a smart card device with access controlled by a PIN code.

What about large messages?

As we said earlier, encrypting data with an asymmetric key is slow and uses a great deal of computing power. However, we can encrypt with a symmetric key at much greater speeds.

Consequently, the two methods are frequently used together.

Consider the following scenario.

Person A (the sender) needs to send a large secret message to person B (the receiver). This means that the receiver needs to be sure the message came from the sender, and that they are both sure the message was not read by anyone else. In addition, they want to perform the processing efficiently.

Assume:

- The sender and the receiver each have private and public keys.
- The receiver knows the sender's public key, and the receiver knows the sender's public key.
- Each of the private keys are known only to the individuals who own them.

The message exchange can take place as follows:

1. The sender generates a random symmetric key and encrypts the large message using it. See Figure B-1.

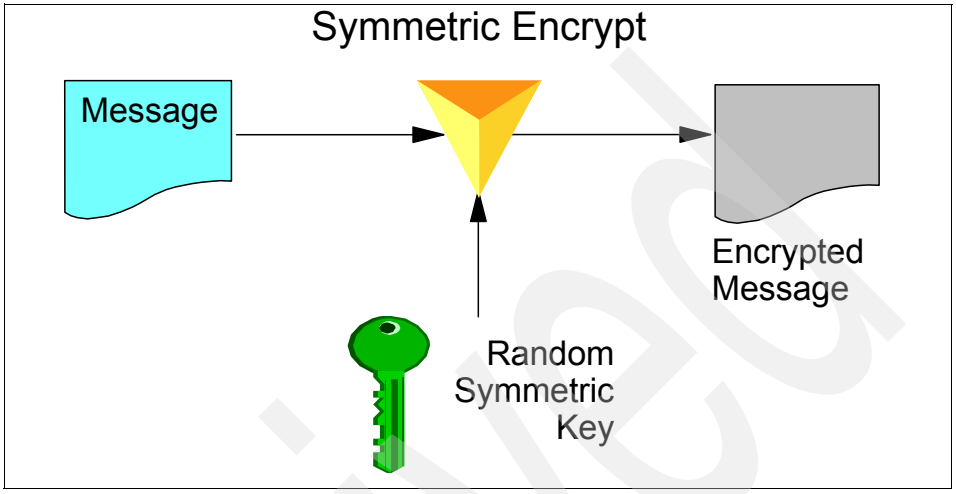

Figure B-1 Secure message process: Symmetric encryption of data

2. The sender now encrypts the random symmetric key using the receiver's public key. See Figure B-2.

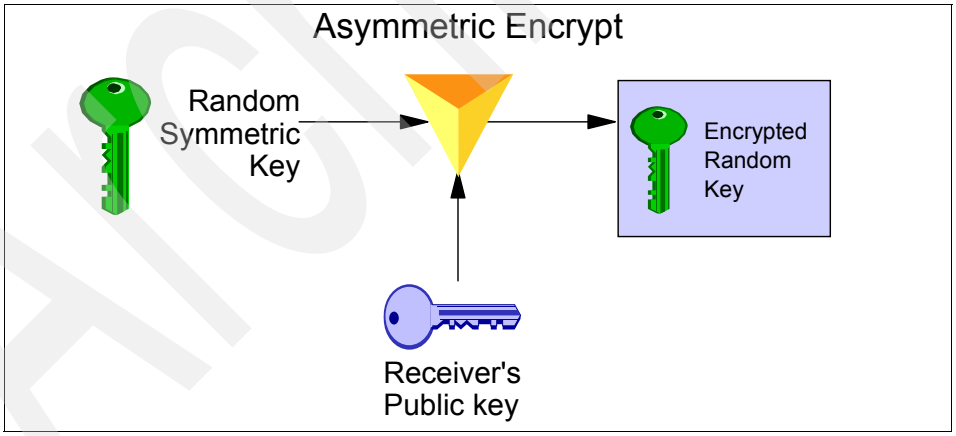

Figure B-2 Secure message process: Asymmetric encryption of symmetric key

3. The sender then sends the resulting package containing the message (encrypted under the random symmetric key) and the random key itself encrypted with the receiver's public key, as shown in Figure B-3.

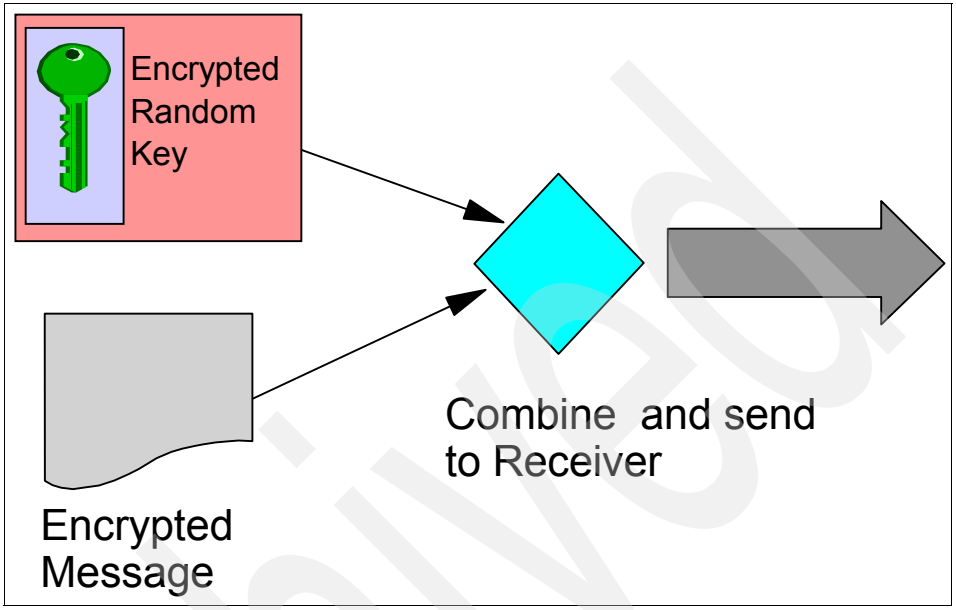

Figure B-3 Secure message process: Composition of the message

When the receiver receives the package, the receiver proceeds as follows:

1. The encrypted data key is decrypted with the receiver's private key. See Figure B-4.

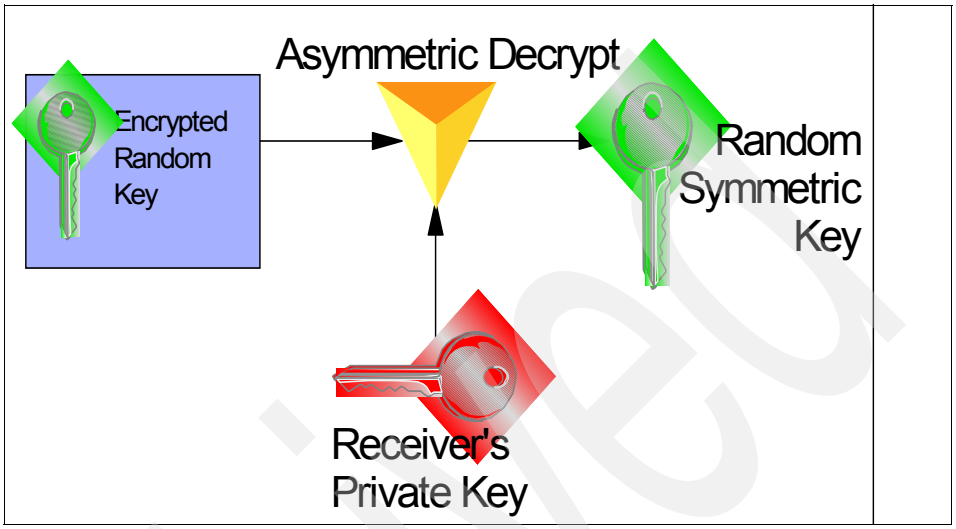

Figure B-4 Secure message process: Retrieval of the data key

2. The resulting symmetric key is then used to decrypt the long message. See Figure B-5.

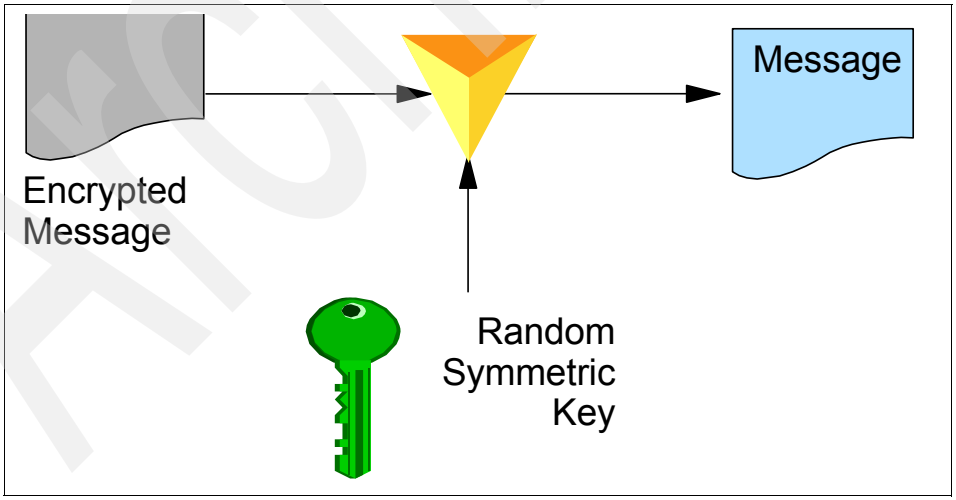

Figure B-5 Secure message process: Recovery of clear data at the receiver's site

This method of communication has ensured:

- The sender and the receiver each know that only the receiver can receive the message.
- The performance of the processing is reasonably fast because the large message is processed using symmetric encryption.

This description demonstrates how the use of keys can be combined to exchange secure messages. However, this is not to say that secure message exchange is always performed this way. The uses of encryption using these types of keys is limited only by the imagination of the designers.

Digital signatures

There are several methods available to produce a hash value from a message. This is a value that has a very high likelihood of reflecting changes to the original message. (By this, we mean that any change to the original message will produce a change in the hash value.) These methods include MD5, SHA-1, SHA-256, and so on.

If one of these hash values is produced for a message, and then encrypted with a private key, this can be used as a “digital signature.”

In order to ensure that a message has not changed, and that it has been sent from a particular individual, a hash value can be produced and then encrypted with the private key of the sender.

When the package is received, the hash value can be recalculated and then it can be compared with the decrypted value of the hash sent with the message. The decryption would, of course, be carried out using the sender’s public key.

If the values do not match, the message cannot be trusted.

Certificates

Because public keys are intended to be public knowledge, there is a need to unambiguously certify the identity of the owner of a public key. The wide-spread method used today to achieve this certification is to package one’s public key in a file called a “digital certificate.” The digital certificate contains, among other information, the public key value and the public key owner’s name. The contents of a certificate is digitally signed by a trusted certificate authority, and therefore, the owner’s name is cryptographically bound to the key value.

The most widely used format for these certificates is known as the X.509 V3 certificate.

X.509 is a full standard for a public key infrastructure. It was originally developed in 1988 as part of the wider X.500 directory standards. The X.509 standards used for certificates are really the X.509 version 3 standards.

An X.509 V3 certificate contains the following information:

- Certificate
  - Version
  - Serial number
  - Algorithm ID
  - Issuer
  - Validity
    - Not before
    - Not after
  - Subject
  - Subject public key info
    - Public key algorithm
    - Subject public key
  - Issuer unique identifier
  - Subject unique identifier
  - Extensions
- Certificate signature algorithm
- Certificate signature

As already mentioned, each certificate is signed using a digital signature. Therefore, it cannot be changed after issued by the certificate authority.

Who to trust: The certificate authority

In order for there to be common agreement regarding certificates, and in order to establish who has authority to bind the public key value to an owner's name, there needs to be a body that is trusted by all parties concerned with the use of digital certificates. Such a body is known as a certificate authority (CA).

This leads us into the wider discussion of a public key infrastructure. This book does not cover this subject, but for more information, refer to other IBM Redbooks about PKI, such as *Implementing PKI Services on z/OS*, SG24-6968.

How can we use certificates?

Although a fuller implementation of certificates with relation to certificate authorities can be used, this is not necessary. Certificates can be generated and

used just as public key containers, without the use of the wider context, as long as the parties involved are happy that a local trust agreement is adequate.

Packages

In addition to certificates issued in X.509 V3 format, public and private keys can be distributed using standards part of the Public Key Cryptography Standards (PKCS) set. These standards were originally developed by RSA Data Security Inc. These standards are now widely accepted and recognized worldwide. Some of the most relevant standards in the context of the Encryption Facility for z/OS are shown in Table B-1.

Table B-1 Some PKCS standards

Package	Name	Notes and comments
PKCS #7	Cryptographic Message Syntax Standard	See RFC 2315: http://tools.ietf.org/html/2315
PKCS #10	Certification Request Standard	See RFC 2986: http://tools.ietf.org/html/2986
PKCS #12	Personal Information Exchange Syntax Standard	Defines a file format commonly used to store private keys with accompanying public key certificates protected with a password-based symmetric key.

Note that PKCS #12 is the standard used to build a package containing both an X.509 V3 certificate containing a public key and the corresponding private key.

C

Encryption Facility for z/OS: Performance and sizing

This appendix contains information from a paper published previously using the IBM Techdocs facility. We include this paper here because it provides very useful performance information for anyone needing to understand the characteristics of the z/OS Encryption Facility.

Tip: Access IBM Techdocs with the following URL:

http://www.ibm.com/support/techdocs/atsmastr.nsf/Web/TechDocs

Introduction

This appendix describes the factors that can affect the performance of the Encryption Facility for z/OS tool and provides assistance in estimating resource requirements. We discuss both the Encryption Services feature and the DFSMSdss feature of the product. Although the two are functionally identical, the control statement parameters are different, and the performance characteristics are different.

Capacity and performance expectations for the IBM Encryption Facility for z/OS depend on a number of factors. As with all encryption, the choice of algorithm and whether that algorithm is supported by hardware is the significant consideration. In addition, the Encryption Facility provides the option to compress data before it is encrypted. This compression is done through specialized instructions in the general purpose CPs, not on the crypto hardware. Both the selection of the encryption algorithm and compression will impact your resource utilization and possibly other work in the system.

Choice of algorithm

The Encryption Facility provides three choices of encryption algorithms:

- CLRTDES: Clear 24-byte key, triple DES algorithm
- CLRAES: Clear 128-bit key, AES algorithm
- ENCTDES: Secure 24-byte key, triple DES algorithm

The two clear key options are supported on all platforms; however, if the algorithm is not supported by hardware on the system, the encryption will be done in the software.

The secure 24-byte key option requires secure key hardware to be installed and available on the system.

Table C-1 on page 167 shows that a Cryptographic Coprocessor Facility (CCF) is required to execute clear key, triple DES on an IBM eServer zSeries 800 (z800) or 900 (z900). On an IBM eServer zSeries 890 (z890) or 990 (z990) or IBM System z9 Business Class (z9 BC) or System z9 Enterprise Class (z9 EC), a CP Assist for Cryptographic Function (CPACF) must be available.

Table C-1 Required crypto hardware by option

	z800, z900	z890, z990	System z9
CLRTDES	CCF	CPACF	CPACF
CLRAES	(in software)	(in software)	CPACF
ENCTDES	CCF	PCIXCC/CEX2	CEX2C

For clear key AES, a CPACF is required on the z9 BC and z9 EC, but this algorithm is only supported in software on the earlier machines.

Similarly, to execute secure key triple DES, a secure key card (PCIXCC or CEX2) must be installed on a z890 or z990. On a z9 BC or z9 EC, a Crypto Express2 feature must be installed, and at least one of the crypto processors on that feature must be configured as a coprocessor. Secure key algorithms are not supported when the crypto processors are configured as accelerators. On a z800 or z900, a CCF provides the secure key support.

For more information about the clear key and secure key capabilities of the zSeries or System z9 processors, see the crypto documents on Techdocs. Search crypto on the following Web site:

http://www.ibm.com/support/techdocs

Compression and tapes

In addition to the choice of encryption algorithm and the availability of hardware, another significant performance factor is the decision of whether to compress the data prior to encrypting it. The Encryption Facility for z/OS provides this option, because most tape drives will compress the data within the drive as it is written to tape, saving space, but encryption negates compression, driving up space requirements.

Compression algorithms look for repeating strings that can be replaced with shorter strings. Encryption, however, will eliminate repeating strings, as it makes the encrypted data appear to be random. Eliminating repeating strings makes compression less efficient. So, if the data is not compressed first, the data will be encrypted byte for byte, and passed to the tape drive, where it is likely that little, if any, compression will occur, thus increasing the number of tapes required. An unencrypted file that uses only a single tape might require as many as two or three tapes if it is encrypted. If the COMPRESS option is specified, then that same data set, after it is compressed and encrypted by the Encryption Facility, might still fit on a single tape, but might expand as much as 50%, requiring another tape.

The impact of compression on the amount of space required for the final data set depends greatly on the type of data being compressed. Random data (for example, the output metrics from a nuclear reactor might be very random) does not lend itself to compression. Data that has lots of repeating strings, such as logs from an IMS database, might compress very efficiently.

If the user chooses the Compress option, the Encryption Facility uses the compression algorithm implemented on the general purpose CPs, which is not the same as the algorithms implemented in the tape drives. This compression is performed before passing the data to the encryption hardware. While this reduces the amount of space required for the encrypted data set, the compression is done by native hardware instructions, not by the crypto hardware.

Because disk devices do not typically have compression support, data encrypted to disk will have similar space requirements to the original data set. The encrypted data set will be slightly larger because a header record is added to the encrypted output file.

Protecting the data key

There are also hardware requirements associated with the protection of the data key, although these typically have minimal performance impact. The Encryption Facility will either generate a random key value and use the RSA option to protect that key value, or use the PASSWORD option to calculate a data key.

With the RSA option, the user provides a label for a key that is available in the PKDS. The public key is retrieved and used to encrypt the random key value that was selected as the data key. The encrypted data key is stored in the header record in the encrypted file. When the data needs to be decrypted, the label associated with the key will again be provided to the utility, and the corresponding private key will be retrieved to decrypt the data key. That decrypted data key is used to decrypt the data.

With the PASSWORD option, the user specifies a password, or string, that will be passed through a hashing algorithm to create a unique thumbprint of that string. That thumbprint is used to create the data key. When the data needs to be decrypted, the same password must be provided to the utility, which will generate the same hash value and the same data key to decrypt the data.

The choice of the RSA or PASSWORD option will generally have a minimal impact on the performance of the Encryption Facility, because the process only occurs once during the execution of the utility.

The choice of RSA or PASSWORD does impact the hardware required. The PASSWORD option uses hashing that is available on all the base crypto hardware (CCF or CPACF). The RSA option, however, is supported by several hardware devices. On the z800 and z900, RSA keys of 1024 bits or less are supported on the CCF. For longer RSA keys, up to 2048 bits, a PCICC is required simply for the RSA option. On the z890 and z990, a PCIXCC or Crypto Express2 card is required to support RSA keys. The z9 BC and z9 EC require a Crypto Express2 card configured as a coprocessor to use the RSA option.

Encryption services sizing

For estimation purposes, you can use Table C-2 to calculate how many megabytes per second (MBps) can be encrypted on a single engine, on a given platform, when running the Encryption Services feature. These figures are based on tests run in a controlled, unconstrained environment, with no other workloads running. (See Table C-4 on page 173 and Table C-5 on page 174 for a description of the test environments.)[1]

Table C-2 shows that a single engine on a z9 EC can encrypt approximately 143 MBps when using clear key triple DES. If the data is compressed by the Encryption Facility, approximately 64 MBps can be processed (because the general purpose CPs are also handling the compression work).

Note: The numbers for CLRTDES on the z990 and z890 have been updated from the original version of this document. Additional test runs have demonstrated that the MB per CPU second numbers are better than originally reported.

Table C-2 Encryption Services feature: MB per CPU second

System	Clear key triple DES	Clear key triple DES with compression	Clear key AES	Clear key AES with compression	Secure key triple DES	Secure key triple DES with compression
z9 EC	143 MB/CPU second	64 MB/CPU second	167 MB/CPU second	67 MB/CPU second	52 MB/CPU second	42 MB/CPU second
z990	126	44	33	29	34	29
z890	94	33	25	21	26	23

[1] All test runs were completed in a controlled, unconstrained environment, with no other workloads competing for resources.

System	Clear key triple DES	Clear key triple DES with compression	Clear key AES	Clear key AES with compression	Secure key triple DES	Secure key triple DES with compression
z900	27	20	15	15	27	20
z800	20	15	11	11	20	15

Note the difference between clear key AES on a z9 EC (167 MBps) and clear key AES on a z990 (33 MBps). This is because CLRAES is supported on the CPACF hardware on the z9 BC and EC, but not on the CPACF on the z990, so this work is done in software.

If your client needs to encrypt up to 100 MB of data using the Encryption Services feature on a z900 machine without compression, using the clear key triple DES option, it will require approximately four CPU seconds (100 MB/27 MB per CPU second). The same work will take approximately one CPU second on a z9 EC (100 MB/143 MB per CPU second). As another example, if your client needs to encrypt a larger file, 100 GB, to tape, and the data needs to be compressed to save tapes, it will take approximately 1.4 CPU hours using the clear key triple DES option on a z900 (100,000 MB/20 MB per CPU second). On a z9 EC, the same work will take roughly 26 CPU minutes (100,000 MB/ 64 MB per CPU second).

Keep in mind that the previous numbers provide an estimate of the CPU seconds required to encrypt 1 MB of data. The amount of wall-clock time will depend on several additional factors. First, the I/O and device configuration can be a significant factor. FICON® channels will provide better elapsed time than an ESCON® environment if the configuration is limited by the bandwidth of the ESCON channel. Faster tape drives will provide better elapsed time than slower tape drives if the tape write speed is the performance bottleneck in the environment. Secondly, the use of the secure key triple DES algorithm when running on a PCI card will take longer than secure key triple DES on a CCF. The PCI cards are installed in the I/O cage on the z890, z990, z9 BC, and z9 EC, and there is an additional delay involved in routing the work to these devices. These cards will help reduce the workload on the general purpose CPs, but do have a longer path length, which can result in longer wall-clock times. Finally, the compressibility of the data will influence wall-clock time. Data that compresses well might take less CPU resources, and the smaller amount of data being passed to the encryption hardware will typically improve throughput in the crypto hardware and the smaller amount of data being passed to the tape drives will also improve the write time on the drive.

DFSMSdss sizing

The DFSMSdss Encryption feature of the Encryption Facility has a built-in efficiency because the crypto hardware works best with large blocks, and DFSMSdss by its design uses the largest block sizes that are supported on the devices that it is reading from and writing to. The Encryption Services feature is flexible enough to read small blocks or records and combine them into large blocks to be passed to the crypto hardware, but the performance is impacted by the additional processing involved in the handling these small blocks. The DFSMSdss feature simply takes the large blocks from DFSMSdss and passes them to the crypto hardware without further processing. In addition, DFSMSdss uses EXCP processing, as opposed to the sequential access method used by the Encryption Services feature. The larger input blocks and EXCP processing are reflected in the following performance numbers.

DFSMSdss has always had a COMPRESS option, which compresses repeated sequential bytes. With the Encryption Facility, there is now also a HWCOMPRESS option that compresses repeated patterns of bytes and takes advantage of the compression instructions available on the general purpose CPs. The HWCOMPRESS option implements the same functionality as the COMPRESS option in the Encryption Services feature, but the logic is not identical.

As with the Encryption Services numbers, for estimation purposes, you can use Table C-3 on page 172 to calculate approximately how many MBps can be encrypted on a single engine, on a given platform, when running the DFSMSdss feature. These figures are based on tests run in a controlled, unconstrained environment, with no other workloads running.

The test environment for the DFSMSdss benchmarks was different from the test environment for the Encryption Services feature. In addition, the performance metrics that are used to calculate throughput for the two features were different, so it is not appropriate to compare actual results between the Encryption Services feature and the DFSMSdss feature. (See Table C-4 on page 173 and Table C-5 on page 174 for a description of the test environment.[2])

[2] All test runs were completed in a controlled, unconstrained environment, with no other workloads competing for resources. Benchmark jobs performed physical dumps of the volumes; however, similar performance is expected for logical dumps.

Table C-3 DFSMSdss: MB per CPU second

System	Clear key triple DES	Clear key triple DES with compression	Clear key AES	Clear key AES with compression	Secure key triple DES	Secure key triple DES with compression
z9 EC	269 MB/CPU second	93 MB/CPU second	269 MB/CPU second	93 MB/CPU second	252 MB/CPU second	87 MB/CPU second
z990	183	69	43	42	192	67
z890	137	52	32	31	144	50
z900	36	26	18	18	35	26
z800	27	19	13	13	26	19

This table shows that a single engine on a z9 EC can encrypt approximately 269 MB/CPU second when using clear key triple DES. If the data is compressed through HWCOMPRESS by DFSMSdss, approximately 93 MB/CPUs per second can be processed. As with the Encryption Services feature, this compression is done on the general purpose CPs, but it is done by DFSMSdss, not by the Encryption Facility code.

Note the difference between clear key AES on a z9 EC (269 MBps) and clear key AES on a z990 (43 MBps). This is because CLRAES is supported on the CPACF hardware on the z9 BC and EC, but not on the CPACF on the z990, so the CLRAES work is done in software on the z890/z990 and earlier machines.

If your client needs to encrypt up to 100 MB of data using the DFSMSdss feature on a z900 machine without compression, using the clear key triple DES option, it will require approximately three CPU seconds (100 MB/36 MB per CPU second). The same work will take approximately .3 CPU seconds on a z9 EC (100 MB/269 MB per CPU second). As another example, if your client needs to encrypt a larger file, 100 GB, to tape, and the data needs to be compressed to save tapes, it will take approximately 1.1 CPU hours using the clear key triple DES option on a z900 (100,000 MB/26 MB per CPU second). On a z9-109, the same work will take roughly 17 CPU minutes (100,000 MB/ 93 MB per CPU second).

The previous numbers provide an estimate of the CPU seconds required to encrypt 1 MB of data. Just as with the Encryption Services feature, the amount of wall-clock time will depend on additional factors, including I/O and device configuration, use of secure key versus clear key, and the compressibility of the data.

For a better estimate of the resource requirements, we recommend that you take advantage of the Sizing Tool for the Encryption Facility. This tool can provide calculations for both the Encryption Services feature and the DFSMSdss feature. Contact your IBM representative or IBM Business Partner who can assist in completing and submitting the form.

Table C-4 and Table C-5 on page 174 provide information about our test environments.

Table C-4 Hardware configuration: Encryption Services

Configurations for the Encryption Services test runs	
z9 EC	2094 Model 718 Processor, dedicated LPAR with 4 dedicated CPUs and 4 CPACF devices (3 CPUs/CPACFs configured offline); 4 GB main storage; 4 CEX2 cards configured as Crypto Coprocessors but only one card online; dedicated ESCON I/O to disk (2 paths) z/OS 1.7 ICSF HCR7730
z990	2084 Model 316 Processor, dedicated LPAR with 4 dedicated CPUs and 4 CPACF devices (3 CPUs/CPACFs configured offline); 4 GB main storage; 4 CEX2 cards, but only one card online; dedicated ESCON I/O to disk (2 paths) z/OS 1.7 ICSF HCR7720
z890	Results extrapolated using relative system capacity ratios
z900	2064 Model 116 Processor, dedicated LPAR with 4 dedicated CPUs (only one CPU was configured online) and 2 CCFs (only 1 CCF online, the CPU associated with the second CCF was configured offline); 8 GB main storage dedicated ESCON I/O to disk (2 paths) z/OS 1.7 ICSF HCR7720
z800	Results extrapolated using relative system capacity ratios

Table C-5 Hardware configuration: DFSMSdss feature

Configurations for the DFSMSdss feature test runs	
z9-109	2094 Model 701 Processor, dedicated LPAR with 1 shared CPU and 1 CPACF device; 8 GB main storage; 1 Crypto Express2 feature, with both engines configured and online as Crypto Coprocessors dedicated FICON I/O disk (8 path) to 3592/3590 tape z/OS 1.7 ICSF HCR7730
z990	2084 Model 301 Processor, dedicated LPAR with shared CPU and 1 CPACF device; 4 GB main storage; 1 Crypto Express2 feature, with both crypto coprocessors online; dedicated FICON I/O disk (8 path) to 3592/3590 tape z/OS 1.7 ICSF HCR7720
z890	Results extrapolated using relative system capacity ratios
z900	2064 Model 101 Processor, dedicated LPAR with 1 shared CPU and 1 CCF; 4 GB main storage; one PCICC feature installed, but only a single coprocessor engine available to the LPAR; dedicated ESCON I/O disk (2 path) to 3590 tape z/OS 1.7 ICSF HCR7720
z800	Results extrapolated using relative system capacity ratios

D

EBCDIC and ASCII

This appendix contains two programs that can be used as exits within the IEBGENER utility program for converting data from EBCDIC to ASCII or from ASCII to EBCDIC.

Note: The code supplied here has not been subjected to any formal IBM test and is distributed on an “AS IS” basis without any warranty either express or implied. The implementation of any of the techniques described or used herein is a client responsibility and depends on the client’s operational environment. While each item may have been reviewed for accuracy in a specific situation and may run in a specific environment, there is no guarantee that the same or similar results will be obtained elsewhere. Clients attempting to adapt these techniques to their own environments do so at their own risk.

Data conversion

Data that has been transferred to another platform can require converting from EBCDIC to ASCII or vice versa.

There are many issues with such conversion, not least the fact that the data must be of character format, rather than binary. The programs included here can be used to convert data held in FB or VB format z/OS data sets from one format to the other. They require the use of the standard MVS utility program called IEBGENER.

They are subject to the following restrictions:

- They process only data with a RECFM of F, FB, V, or VB.
- They handle the character translation by making use of the DFSMS macro XLATE. This uses a fixed translation table, which is documented in Appendix 1.6.5 of *z/OS DFSMS Using Data Sets*, SC26-7410.
- There is no allowance for different code pages.
- There is no attempt to cater for carriage return and line feed characters, which might or might not be required by your application.

The programs can be invoked using IEBGENER through JCL. See Example D-1 and Example D-2 on page 177.

Although these programs are subject to the restrictions mentioned earlier, they can be modified to provide further support if required.

Example: D-1 IEBGENER: First exit example

```
//COPY1   EXEC PGM=IEBGENER
//STEPLIB  DD  DSN=ITSOLD.A.LOAD,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//IDIOFF   DD    DUMMY
//SYSIN    DD  *
ETOA      GENERATE
          EXITS     DATA=EBTOAS00
//SYSUT1   DD  DSN=ITSOLD.TEST.RECFMV,DISP=SHR
//SYSUT2   DD  DSN=ITSOLD.TEST.RECFMV.ASCII,DISP=(,CATLG),
```

Example: D-2 IEBGENER: Second exit example

```
//COPY2   EXEC PGM=IEBGENER
//STEPLIB  DD  DSN=ITSOLD.A.LOAD,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  *
ATOE      GENERATE
          EXITS     DATA=ASTOEB00
//SYSUT1   DD  DSN=ITSOLD.TEST.RECFMV.ASCII,DISP=SHR
//SYSUT2   DD  DSN=ITSOLD.TEST.RECFMV.EBCDIC,DISP=(,CATLG),
//  LIKE=ITSOLD.TEST.RECFMV
```

EBTOAS00 program

Example D-3 shows the EBTOAS00 program.

Example: D-3 Exit program EBTOAS00

```
EBTOAS00 CSECT                                                          00010000
EBTOAS00 AMODE 31                                                       00020006
EBTOAS00 RMODE 24                                                       00030000
* ------------------------------------------------------------------- * 00040000
*                                                                     * 00050000
*      E B C D I C   T O   A S C I I                                  * 00060000
*      -----------------------------                                  * 00070000
*                                                                     * 00080000
*        Last Updated 10/08/2006                                      * 00090005
*                                                                     * 00100000
*         Module : EBTOAS00                                           * 00110000
*   Date written : 3rd August 2006                                    * 00120000
*      Function : IEBGENER exit for EBCDIC to ASCII conversion        * 00130000
*                                                                     * 00140000
*        HISTORY : 03/08/2006 - Initial Version                       * 00150000
*                                                                     * 00160000
*   Module attributes: RENT,REUS,AMODE(31),RMODE(24)                  * 00170000
*                                                                     * 00180000
* ------------------------------------------------------------------- * 00190000
*                                                                       00200000
         SAVE  (14,12)                                                  00210000
*                                                                       00220000
         BASR  R12,R0              Get first base reg                   00230000
*                                                                       00240000
         USING *,R12               Module addressability                00250000
*                                                                       00260000
         LM    R4,R5,0(R1)         Load Parameters                      00270000
*                                                                       00280000
         USING IHADCB,R5           DCB addressability                   00290000
*                                                                       00300000
         XR    R3,R3               Clear record length                  00310000
```

```
         LR    R6,R4                Point to record                    00320000
         TM    DCBRECFM,DCBRECV     Variable length records?           00330000
         BO    RECFMV               yes, branch                        00340000
*                                                                      00350000
         ICM   R3,B'0011',DCBLRECL Get record length                   00360000
         B     XLATE                                                   00370000
*                                                                      00380000
RECFMV   DS    0H                                                      00390000
*        R6 points to data. RDW is just 4 bytes before record          00400003
*                                                                      00410003
         S     R4,=F'4'             back off 4 bytes for RDW           00420003
         ICM   R3,B'0011',0(R4)     Get record length                  00430000
         S     R3,=F'4'             Reduce for RDW length              00440000
*                                                                      00450000
XLATE    XLATE (R6),(R3),TO=A                                          00460000
*                                                                      00470000
         XR    R15,R15              Clear return code                  00480000
         RETURN (14,12),RC=(15)     Return                             00490000
*                                                                      00500000
         DROP  R12                                                     00510000
         DROP  R5                                                      00520000
*                                                                      00530000
         DCBD  DSORG=PS                                                00540000
         YREGS                                                         00550000
*                                                                      00560000
         END                                                           00570000
```

ASTOEB00 program

Example D-4 shows the ASTOEB00 program.

Example: D-4 Exit program ASTOEB00

```
ASTOEB00 CSECT                                                         00010000
ASTOEB00 AMODE 31                                                      00020004
ASTOEB00 RMODE 24                                                      00030000
* ------------------------------------------------------------------ * 00040000
*                                                                    * 00050000
*     A S C I  I  T O   E B C D I C                                  * 00060000
*     ----------------------------                                   * 00070000
*                                                                    * 00080000
*       Last Updated 10/08/2006                                      * 00090003
*                                                                    * 00100000
*        Module : ASTOEB00                                           * 00110000
*   Date written : 3rd August 2006                                   * 00120000
*       Function : IEBGENER exit for ASCII to EBCDIC conversion      * 00130000
*                                                                    * 00140000
*       HISTORY : 03/08/2006 - Initial Version                       * 00150000
*                                                                    * 00160000
*   Module attributes: RENT,REUS,AMODE(31),RMODE(24)                 * 00170000
*                                                                    * 00180000
```

```
* ---------------------------------------------------------------- * 00190000
*                                                                    00200001
         SAVE  (14,12)                                              00210001
*                                                                    00220001
         BASR  R12,R0              Get first base reg               00230001
*                                                                    00240001
         USING *,R12               Module addressability            00250001
*                                                                    00260001
         LM    R4,R5,0(R1)         Load Parameters                  00270001
*                                                                    00280001
         USING IHADCB,R5           DCB addressability               00290001
*                                                                    00300001
         XR    R3,R3               Clear record length              00310001
         LR    R6,R4               Point to record                  00320001
         TM    DCBRECFM,DCBRECV    Variable length records?         00330001
         BO    RECFMV              yes, branch                      00340001
*                                                                    00350001
         ICM   R3,B'0011',DCBLRECL Get record length                00360001
         B     XLATE                                                00370001
*                                                                    00380001
RECFMV   DS    0H                                                   00390001
*        R6 points to data. RDW is just 4 bytes before record       00400001
*                                                                    00410001
         S     R4,=F'4'            back off 4 bytes for RDW         00420001
         ICM   R3,B'0011',0(R4)    Get record length                00430001
         S     R3,=F'4'            Reduce for RDW length            00440001
*                                                                    00450001
XLATE    XLATE (R6),(R3),TO=E                                       00460001
*                                                                    00470001
         XR    R15,R15             Clear return code                00480001
         RETURN (14,12),RC=(15)    Return                           00490001
*                                                                    00500001
         DROP  R12                                                  00510001
         DROP  R5                                                   00520001
*                                                                    00530001
         DCBD  DSORG=PS                                             00540001
         YREGS                                                      00550001
*                                                                    00560001
         END                                                        00570001
```

E

A simple installation verification program

We assume that ICSF is up and running.

The IVP consists of two batch jobs, ENCRYPT and DECRYPT that use the PASSWORD option.

```
//ENCRYPT  JOB 'D999',REGION=8192K
//*
//ENCRYPT1 EXEC PGM=CSDFILEN
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=ITSOPL.DATA.IN,DISP=SHR
//SYSUT2   DD DSN=ITSOPL.DATA.SAFE,
//         UNIT=SYSDA,DISP=(NEW,CATLG),
//         SPACE=(1024,(60,10)),AVGREC=K
//SYSIN    DD *
DESC='MY SECURE DATA'
CLRT-DES
PASSWORD=ITSOITSO
/*
//
```

Figure E-1 The ENCRYPT IVP batch job

The DECRYPT batch job is similar.

```
//DECRYPT  JOB 'D999',REGION=8192K
//*
//DECRYPT1 EXEC PGM=CSDFILDE
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=ITSOPL.DATA.SAFE,DISP=SHR
//SYSUT2   DD DSN=ITSOPL.DATA.OUT,
//         UNIT=SYSDA,DISP=(NEW,CATLG),
//         SPACE=(80,(60,10)),RECFM=FB
//SYSIN    DD *
PASSWORD=ITSOITSO
/*
//
```

Figure E-2 The DECRYPT IVP batch job

The test data is in the data set ITSOPL.DATA.IN.

```
EDIT       ITSOPL.DATA.IN                                Columns 00001 00072
Command ===>                                               Scroll ===> PAGE
****** **************************** Top of Data ******************************
000100 The quick brown dog jumped over the lazy fox.
****** *************************** Bottom of Data ***************************
```

Figure E-3 ITSOPL.DATA.IN data set

Before we run the RACF job, we gave our user ID access to the CSFSERV class.

```
SETR CLASSACT(CSFSERV)
SETR RACLIST(CSFSERV)
RDEFINE CSFSERV * UACC(NONE)
PERMIT * CLASS(CSFSERV) ID(ITSOPL) ACCESS(READ)
SETR RACLIST(CSFSERV) refresh
```

Figure E-4 Grant user ID access to CSFSERV class

Everything is set, so we submitted the ENCRYPT job. The CSDFILEN program wrote the following to SYSOUT.

```
 CSDFILEN Encryption Utility 07/26/2006 (MM/DD/YYYY) 15:50:09 (HH:MM:SS)
INPUT:  DESC='MY SECURE DATA'
INPUT:  CLRT-DES
INPUT:  PASSWORD=********************************
 CSDFILEN:            :
INPUT:  LRECL    80 BLKSIZE   27920 RECFM FB
OUTPUT: BLKSIZE   27998
ENCRYPTION OF DATA: CLEAR     T-DES KEY USING CPACF
  RECORDS READ:              1  WRITTEN:              1
  BYTES READ:                80
  BYTES WRITTEN:               560  WITH HEADER AND PAD
CIPHER TIMES (IN SECONDS): HIGH:    0.000003 DATA:       96 LOW:    0.000003 DATA: 96
 TOTAL CIPHER TIME (IN SECONDS):     0.000003  CIPHERS:              1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure E-5 SYSOUT DD for the ENCRYPT IVP job

Here is a copy of the contents of the safe file.

```
HEADER..MY SECURE DATA                                                  ...}×..¨
```

Figure E-6 First 80 bytes of the encrypted data

Next, we ran the DECRYPT IVP job. The DECRYPT program generated the following to SYSPRINT.

```
 CSDFILDE Decryption Utility 07/26/2006 (MM/DD/YYYY) 17:04:38 (HH:MM:SS)
 CSDFILDE:    HEADER VERSION :      1
 CSDFILDE:            :
INPUT: DESC = MY SECURE DATA
INPUT: LRECL     80 BLKSIZE   27920 RECFM FB
INPUT:  PASSWORD=********************************
RECORDS READ:              1  WRITTEN:              1
  BYTES READ:               560  BYTES RECOVERED:                80
CIPHER TIMES (IN SECONDS): HIGH:   0.000123 DATA:   27536 LOW:   0.000123 DATA:   27536
TOTAL CIPHER TIME (IN SECONDS):     0.000123  CIPHERS:              1
  TOTAL ELAPSED TIME:   0:00:00.02
```

Figure E-7 SYSPRINT DD from the DECRYPT IVP job

The output data set from the DECRYPT IVP job is as expected.

```
VIEW       ITSOPL.DATA.OUT                                Columns 00001 00072
Command ===>                                                 Scroll ===> PAGE
****** ***************************** Top of Data ******************************
000100 The quick brown dog jumped over the lazy fox.
****** **************************** Bottom of Data ****************************
```

Figure E-8 Output data set from the DECRYPT IVP job

If ICSF has not been properly started, the following SYSPRINT output is generated.

```
 CSDFILEN Encryption Utility 07/27/2006 (MM/DD/YYYY) 11:17:42
(HH:MM:SS)
INPUT:  DESC='MY SECURE DATA'
INPUT:  CLRT-DES
INPUT:  PASSWORD=********************************
    **ERROR** CSNBOWH   08  000000
**INFO** CSNBOWH POSSIBLE SAF AUTHORITY VIOLATION
```

Figure E-9 SYSPRINT output from ENCYRPT IVP job when ICSF is not active

```
 CSDFILDE Decryption Utility 07/27/2006 (MM/DD/YYYY) 11:21:34
(HH:MM:SS)
 CSDFILDE:   HEADER VERSION :     1
 CSDFILDE:          :
INPUT: DESC = MY SECURE DATA
INPUT: LRECL    80 BLKSIZE   27920 RECFM FB
INPUT:  PASSWORD=********************************
    **ERROR** CSNBOWH   08  000000
**INFO** CSNBOWH POSSIBLE SAF AUTHORITY VIOLATION
```

Figure E-10 SYSPRINT output from DECRYPT IVP job when ICSF is not active

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 186. Note that some of the documents referenced here may be available in softcopy only.

- *Exploiting S/390 Hardware Cryptography with Trusted Key Entry*, SG24-5455
- *Implementing PKI Services on z/OS*, SG24-6968
- *S/390 Crypto PCI Implementation Guide*, SG24-5942
- *zSeries Crypto Guide Update*, SG24-6870
- *IBM @server zSeries 990 (z990) Cryptography Implementation*, SG24-7070
- *z9-109 Crypto and TKE V5 Update*, SG24-7123

Other publications

These publications are also relevant as further information sources:

- *Encryption Facility for z/OS: User’s Guide Version 1 Release 1.0*, SA23-1349
- *z/Architecture Principles of Operation*, SA22-7832
- *z/OS Cryptographic Services Integrated Cryptographic Service Facility Administrator’s Guide*, SA22-7521
- *z/OS Cryptographic Services Integrated Cryptographic Service Facility Application Programmer’s Guide*, SA22-7522
- *z/OS Cryptographic Services Integrated Cryptographic Service Facility Overview*, SA22-7519
- *z/OS DFSMS Using Data Sets*, SC26-7410
- *z/OS DFSMSdss Storage Administration Guide*, SC35-0423
- *z/OS ICSF Administrator’s Guide*, SA22-7521
- *z/OS ICSF Application Programmer’s Guide*, SA22-7522
- *z/OS ICSF Messages*, SA22-7523

- *z/OS ICSF System Programmer's Guide*, SA22-7520
- *z/OS Security Server RACF Command Language Reference*, SA22-7687
- *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683
- *z/OS Trusted Key Entry Workstation User's Guide 2000*, SA22-7524

Online resources

These Web sites are also relevant as further information sources:

- Decryption Client download

  http://www.ibm.com/servers/eserver/zseries/zos/downloads/#asis
- IBM Techdocs

  http://www.ibm.com/support/techdocs
- National Institute of Standards and Technology

  http://csrc.nist.gov/
- The latest ICSF release

  http://www.ibm.com/servers/eserver/zseries/zos/downloads/
- Details about the use of the Java keytool utility

  http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

D

E

F

G

H

I

Q

R

S

T

U

V

W

X

Z

IBM Redbooks

Encryption Facility for z/OS Version 1.10

(0.2”spine)
0.17”<->0.473”
90<->249 pages

Encryption Facility for z/OS Version 1.10

Principles of operations and options explained

Examples of setup and use of all the features

Expert considerations and recommendations

This IBM Redbook introduces IBM Encryption Facility for z/OS, Program Product 5655-P97, from the description of its principles of operation to information related to its setup intended for system programmers and security officers.

We include recommendations, based on our own experience, in the chapters addressing these topics, including warnings and explanations about problems we discovered during our own trials.

This redbook provides also several examples of the use of the following Encryption Facility for z/OS features:

- Encryption Services
- Encryption Facility for z/OS Java Client
- DFSMSdss Encryption feature

This book also addresses in detail the many options proposed by the product and provides you with considerations and recommendations regarding proper selection of these options in the planned context of use.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7318-00 ISBN 0738489557