IBM

Encryption Facility for z/OS V1.2 OpenPGP Support

- Introduction to OpenPGP and review of cryptography concepts
- Expert guidance to achieve high security and high performance
- Detailed implementation procedures and practical usage scenarios

Patrick Kappeler
Saheem Granados
Stanley Jon
Benjamin Rogers
Ken Rogers
Richard Schultz

Redbooks

ibm.com/redbooks

International Technical Support Organization

Encryption Facility for z/OS V1.2 OpenPGP Support OpenPGP Support

September 2007

SG24-7434-00

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

First Edition (September 2007)

This edition applies to Version 1, Release 2, Modification 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:

Redbooks (logo)  ™	DFSMS™	Redbooks®
OpenPGP	DFSMSdss™	RACF®
eServer™	DFSMShsm™	REXX™
z/Architecture®	IBM®	System z™
z/OS®	IMS™	System z9™
zSeries®	MVS™	Tivoli®
z9™	Net.Commerce™	WebSphere®
DB2®	OS/390®	

The following terms are trademarks of other companies:

Java, JDK, JNI, JRE, JVM, Sun, 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.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

OpenPGP and Pretty Good Privacy are trademarks of OpenPGP Corporation.

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

Preface

This book is about the OpenPGP support available in the Encryption Facility for z/OS® V1.2 (Program Product 5655-P97). It begins with a discussion of the principles of operation of the OpenPGP protocol and a review of some basic cryptographic topics. It presents detailed steps for installing and configuring Encryption Facility for z/OS and implementing OpenPGP support.

Practical examples from our lab and from the authors' real world experiences demonstrate how to set up and use the capabilities of OpenPGP support. The options available within the product are discussed, and recommendations for appropriate selections within the context of your intended use are offered.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Patrick Kappeler has held many international positions, all dealing with mainframes hardware and software technical support and education, during his 36-year career in IBM®. He is now a lead consulting IT Specialist in the Montpellier European Product and Solutions Support Center (PSSC), and has specialized for the past 10 years on e-business security. Patrick provides advanced technical support and consulting on this topic worldwide along with extensive teaching and writing. He is also the co-author and leader of many other ITSO projects on z/OS Security and e-business.

Saheem Granados is an Advisory software engineer who has worked for IBM for over 9 years. He is a Certified Information Systems Security Professional who throughout his career has been a designer and developer for critical z/OS security software, including: Net.Commerce™ for OS/390®, Security Server LDAP Server, Tivoli® Federate Identity Manager, and Trusted Key Entry. As the development product owner of the Encryption Facility V1.2 OpenPGP support, Saheem designed the solution and led a team of developers to successful completion of all required development activities.

Stanley Jon is an Advisory Remote Technical Support Specialist with IBM Canada. He has been with IBM and working on the System z™ platform for ten years. He is part of the team supporting Canadian customers on z/OS defect

issues and North American customers for Q&A questions. He has supported ICSF since 1999 and System SSL since 2003.

Benjamin Rogers is a Staff Software Engineer for IBM Systems & Technology Group, System z Lab Services organization. As a security services professional, he works with IBM customers to design and implement security solutions ranging from enterprise identity management to cryptographic solutions. Before joining Lab Services, Ben spent three years in the z/OS System Test organization architecting, implementing, and testing a security environment that exploited LDAP, RACF®, PKI, EIM, SSL, Kerberos, and ICSF. He spent the three years prior to joining the security team testing Java™ on z/OS and Linux® for zSeries®, with a focus on Java security and test tool development. Ben holds a Bachelor's degree in Computational Mathematics from Michigan State University.

Ken Rogers has more than 25 years systems programming experience in all IBM System z (and previous mainframe) operating systems. His areas of expertise include security and data protection. He has designed and implemented financial and manufacturing protection policies for a variety of customers. Working in almost every business sector, including financial, retail, manufacturing, extreme high technology, medical, and services industries, he has developed the externalist approach of problem solving to an uncommon level. Recently part of the Lab Services, Ken has designed and activated cost-effective, client-focused solutions to all types of problems.

Richard Schultz has more than 28 years of experience in IBM, during which he has held several positions dealing with mainframe hardware and software technical support, with focus on performance measurement and analysis. For the past five years he has specialized on the performance evaluation of z/OS middleware cryptographic products. He provides advanced technical support on this topic within IBM and provides data on this topic to be published for customers worldwide.

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

Paola Bari, Robert Haimowitz, and Richard M. Conway
International Technical Support Organization, Poughkeepsie Center

John C. Dayka, Ravinder Gummadavelli, Bob St John, James W. Sweeny, and Wai Choi
IBM STG Development, Poughkeepsie

Thank to the PGP Corporation for their contribution and for the use of their material.

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will 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 will 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 books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks® publications in one of the following ways:

- Use the online **Contact us** review Redbooks 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

Introduction

This chapter presents a summary of IBM Encryption Facility for z/OS V1.1 and introduces the IBM Encryption Facility for z/OS V1.2 new enhancements and features. This chapter also describes 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.

1.1 Why the Encryption Facility for z/OS

The Encryption Facility for z/OS processes data at rest and is intended for encryption of media whose contents must be securely transported: physically moved such as being shipped in a truck, for example, 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 reported examples in which companies’ finances and image were affected after losing track of physical media with sensitive contents that are known to be easily readable when 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.

Encryption Facility can make use of ICSF to perform encryption and decryption and to manage cryptographic keys. To encrypt data files, Encryption Facility uses the following kinds of cryptographic keys:

- TDES triple-length keys
- 128–bit AES keys

Although the Encryption Facility for z/OS V1.1 implementation was based on a proprietary data format, the V1.2 release provides support for the OpenPGP Message Format standard as defined in RFC 2440. The OpenPGP standard was originally derived from PGP (Pretty Good Privacy). This support will meet the demand for standards-based solutions.

While the functionality of Encryption Facility for z/OS V1.1 continues to be available, this book focuses on the OpenPGP portion of Encryption Facility V1.2. The Encryption Facility for z/OS V1.1 was described in *Encryption Facility for z/OS Version 1.10*, SG24-7318.

1.2 Encryption Facility features/packaging

There are three components (or features) delivered under the generic name of “Encryption Facility for z/OS,” as shown in Figure 1-1.

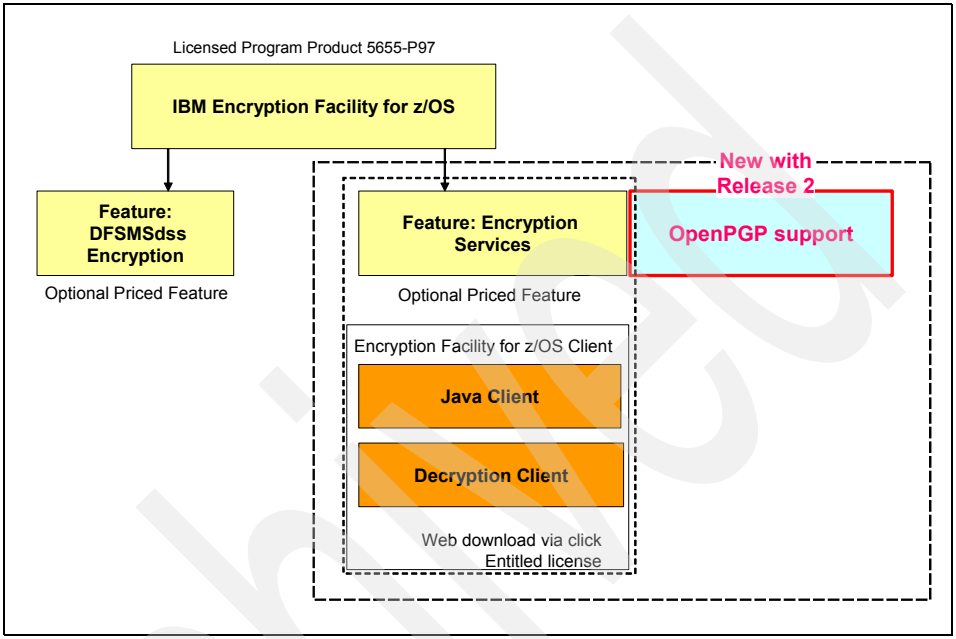


Figure 1-1 Encryption Facility V1.2 packaging

- Encryption Services feature
  This is an optional feature of the Encryption Facility for z/OS and supports both the “System z format” originally introduced in Encryption Facility for z/OS v1.1 and the OpenPGP format. Encryption Facility for OpenPGP provides encryption and decryption of messages and data files in accordance with the OpenPGP standards.

  Note: RFC 2440 defines the set of data structures that an OpenPGP system must produce and consume. Encryption Facility for z/OS V1.2 will implement all of these data structures.

  This feature executes exclusively on z/OS (z/OS V1R6 and later) and exploits, whenever possible, 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 and ElGamal for OpenPGP.

The Encryption Services feature supports data in the following formats as input and output to the encryption and decryption process:

- Physical sequential data sets as members of partitioned data sets (PDS) and partitioned data sets extended (PDSE)
- Sequential data sets
- 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.

The Encryption Facility feature includes the following functions:

- Batch programs CSDFILEN and CSDFILDE to respectively encrypt and decrypt z/OS data
- Encryption Facility OpenPGP Support that supports the OpenPGP standard as described in the Internet Standard RFC 2440

- Encryption Facility for z/OS Client
  This feature is a no-cost, Web deliverable, separately licensed program that provides two components in a single package that support encrypted data in "z format" only.
  - The Encryption Facility for z/OS Java Client is a Java Reference implementation. It 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, except for the OpenPGP support. As with the Encryption Services feature, the Java Client provides data encryption key protection through password or RSA public key cryptography.
  - 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, except for data encrypted with the OpenPGP support.
- 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.

You can download the 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

The Encryption Services, the DFSMSdss Encryption features, and the Decryption Client—that is, all the System z “native” programs—can optionally compress data and, when not using the OpenPGP support, exploit the hardware-accelerated compression available on the System z platform. The OpenPGP support provides software-only compression.

Limitations of the Java Client

The Encryption Facility Java Client does not support data compression and decompression and does not natively support hardware crypto, nor does it support receiving data encrypted with a secure triple DES (T-DES) key.

Figure 1-2 summarizes the possible exchanges between systems hosting the different features of the Encryption Facility for z/OS.

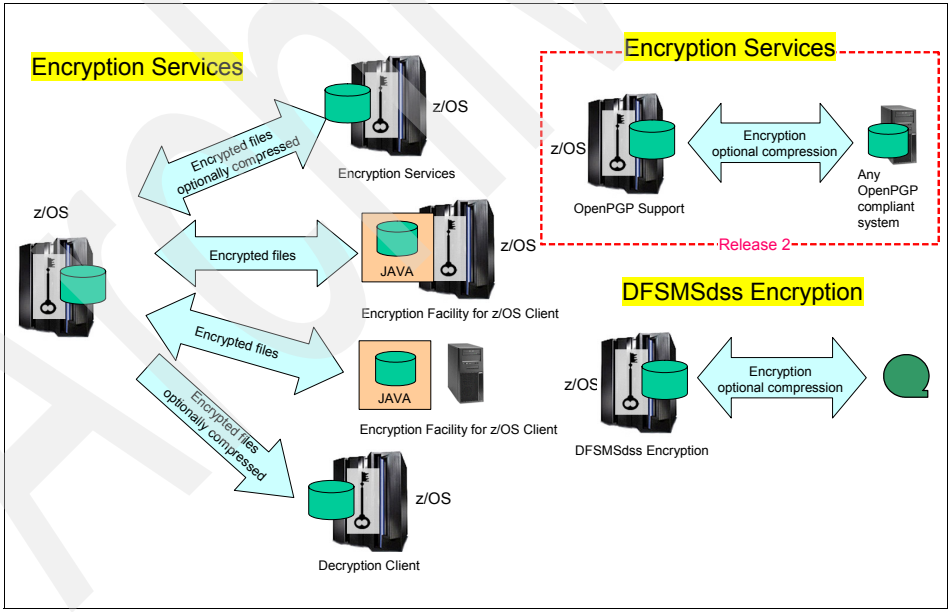


Figure 1-2 Data exchanges with the Encryption Facility for z/OS

1.3 What is new with V1.2

Encryption Facility for z/OS V1.2 provides a new function compared to the previous version: it provides a secure and scalable OpenPGP client exploiting the existing security facilities of z/OS. This function is provided as part of the Encryption Facility Services feature and is implemented as a Java application that can be invoked in an interactive mode or via batch. In this book we are calling this function “Encryption Facility for OpenPGP” or “Encryption Facility OpenPGP support” interchangeably.

Information about system pre-requisites is given in Chapter 4, “Installation” on page 53.

The OpenPGP Internet draft standard protocol defines a syntax for packaging data into packets, where each packet provides the context for a data integrity service like encryption or digital signature.

Encryption Facility for OpenPGP implements all of the required services as described in the Internet draft standard protocol for OpenPGP and specifically provides the following services:

- Public key-based encryption of a session key
- Passphrase-based encryption (PBE) of a session key
- Modification detection of encrypted data
- Compression of data prior to encryption and decompression of data after decryption
- Importing and exporting of OpenPGP certificates in binary or “ASCII armorized” formats
- Digital signatures of data

The Encryption Facility for z/OS OpenPGP support is also able to make use of X.509 certificates provided in a public key infrastructure (PKI) environment, so that the basis of trust for OpenPGP environments can be complemented or extended.

With Encryption Facility for OpenPGP, you can apply many of these services to the same data to form an OpenPGP message that you can exchange with other OpenPGP RFC2440-compatible applications. Encryption Facility for OpenPGP provides leverage of the existing security facilities of z/OS to provide a secure and scalable OpenPGP client.

For example, the Encryption Facility OpenPGP support enables you to do the following:

- Use, as input or output, HFS/zFS files on USS or z/OS partitioned (PDS and PDSE) or sequential data sets
- Exploit the cryptographic acceleration provided by System z hardware
- Use the RACF database and the ICSF PKDS data set as key repositories

Note: For more information and details about OpenPGP support, refer to “Using the Encryption Facility for OpenPGP” on page 63.

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 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, be 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,” meaning 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 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 algorithms, 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 is bigger than the original uncompressed input[1]).

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

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 occurs 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 exceed the volume of the same data when it is compressed only.

In compliance with OpenPGP standards that recommend compressing data for encryption, Encryption Facility for OpenPGP supports compression and decompression of OpenPGP messages and other data; however, there is no hardware assistance provided for OpenPGP compression and decompression of data.

The Encryption Facility OpenPGP support proposes the zlib or zip algorithms to compress data. Both are widely known compression services libraries.

1.5 Miscellaneous information

The z/OS bibliography of the Encryption facility for z/OS includes two books:

- *Encryption Facility for z/OS: Planning and Customizing*, SA23-2229
- *Encryption Facility for z/OS: Using Encryption Facility for OpenPGP*, SA23-2230

Attention: We are referring in the rest of this book to the IBM RACF product as one possible certificates and keys repository. Other non-IBM products, which can replace RACF and are accessible through the z/OS SAF, may also provide this functionality. It is up to the reader to consult the product's vendor regarding whether this is actually the case.

2

An introduction to OpenPGP

This chapter provides a high-level description of the security services and the data exchange messages formats as described in the OpenPGP standard. Whenever appropriate we highlight specific features of the OpenPGP implementation in the IBM Encryption Facility for z/OS V1.2, so that the reader can more precisely position this IBM offering with respect to the overall contents of the standard.

2.1 OpenPGP

OpenPGP is a non-proprietary protocol for encrypting data in files or messages, which can then be recovered using public key cryptography or a secret passphrase. It is based on PGP (Pretty Good Privacy) protocol as originally developed by Phil Zimmermann and sold as a commercial product since 1991. The OpenPGP protocol defines standard formats for encrypted or signed data, digital signatures, and certificates that can be used to exchange public keys.

The OpenPGP Working Group was formed in the Internet Engineering Task Force (IETF) in 1997 to define this standard. PGP, and later OpenPGP, have been very successful in becoming the widely spread standards for the encryption of e-mail and other sets of data that require protection, and emerge today as the leading standards for any requirements that pertain to data security and integrity with a heavy need for large scale interoperability.

OpenPGP is an IETF proposed standard described in RFC 2440, which can be read at `www.ietf.org/rfc/rfc2440.txt`.

The OpenPGP Alliance keeps working with member companies to promote the position of OpenPGP as the universal encryption standard and to model the Public Key Infrastructure (PKI) concepts that are emerging from the OpenPGP community.

2.1.1 Understanding OpenPGP

The goal of OpenPGP is to enable trusted partners to exchange data with confidentiality and integrity. This relies on the use of symmetric and asymmetric encryption algorithms that provide the required service and are also used to establish a mutual trust between exchanging partners. Readers who are not familiar (or who were once familiar) with these algorithms are invited to read the refresher in Appendix A, “Some encryption basics” on page 223.

The way trust is established between partners is discussed later.

Session key

OpenPGP uses symmetric encryption when requested to encrypt data. In OpenPGP terminology, the data encryption key is termed a “session key.”

The Encryption Facility for OpenPGP encrypts data using a randomly generated session key and a symmetric encryption algorithm (such as TDES or AES). The session key is then itself protected by encryption and inserted next to the data in the resulting output file or message.

It is up to the receiving application to decrypt the session key and then use it to recover the clear data. The mechanisms that can be used to protect the session key are discussed in the following sections.

Note: The OpenPGP session key should be randomly generated. Therefore each new request for symmetric encryption gets a new, unpredictable session key.

Session key exchange using public key cryptography

To protect the session key using public key cryptography, it is assumed that the data recipient owns a public and private key pair and provided his or her public key to the data sender. OpenPGP can then be directed to encrypt the dynamically generated session key using this public key, and deliver it encrypted in a “Public Key Encrypted Session Key Packet” that also contains information for the recipient regarding the asymmetric algorithm and public key identifier that were involved in the process. This encrypted key “packet” is then delivered along with the “Symmetrically Encrypted Data Packet” to the data recipient.

Packets, *subpackets*, and *messages* are part of the OpenPGP terminology to designate formatted sets of data intended to be exchanged with OpenPGP-compliant partners. This is further described in “OpenPGP messages” on page 14.

The Encryption Facility for OpenPGP proposes the RSA or ElGamal asymmetric algorithms for public key encryption of the session key.

Session key exchange using passphrase-based encryption

Passphrase-based encryption (PBE) provides a symmetric encryption of the randomly generated session key using a symmetric key that is derived from a shared secret passphrase. The session key encrypted with PBE is sent in a “Symmetric Key Encrypted Session Key” packet. This packet contains information on the symmetric algorithm used to encrypt the session key, and also on the derivation process used to obtain this symmetric key with the so-called “string-to-key” (S2K) set of information. When decrypting the OpenPGP message, the data recipient should provide the secret passphrase as the input to an identical derivation process.

Data compression

OpenPGP offers as an option the capability of compressing clear data before encryption or decompressing clear data after decryption.

The Encryption Facility for OpenPGP proposes the zlib or zip algorithms for the compression of data. *zlib* is a general purpose compression library that includes

one implementation of the DEFLATE compression method, which is a combination of Lempel-Ziv and Huffman encoding of data. *zip* is the compression algorithm at the core of the PKZIP engine, which many platforms have implemented.

2.2 OpenPGP messages

This section provides a high-level description of the data formats used to exchange data processed by OpenPGP services that provide encryption, decryption, signing, and key management functions. If more details are needed, refer to RFC2440.

The data intended to be exchanged are “messages” made of “packets.” A packet is a segment of data with a specific content, the nature of which is indicated by the packet tag located in the packet header. The packet content itself includes items such as a time stamp, a key Identifier, the actual text message, and so forth.

Each packet consists of a packet header followed by the packet body, which may in turn contain “subpackets.” The subpackets are other specific segments of data that are imbedded in the packets; each subpacket contains a subpacket header and a subpacket body.

Note that OpenPGP certificates are also composed of packets, and are stored as such in the OpenPGP key ring. More details about OpenPGP certificates are presented in 2.5, “OpenPGP certificates”.

2.3 OpenPGP versions and corresponding messages formats

The basis for the design of OpenPGP was PGP release 5.x (also known as “PGP 3”). The previous PGP releases were the 2.6.x releases, which used messages and packets formats now deprecated and different from the PGP 5.x messages and packets formats. Generally speaking, PGP 2.6.x was supporting the Version 3 format of packets, whereas PGP 5.x introduced the more expandable Version 4 format.

Note: For the sake of interoperability the Encryption Facility for OpenPGP accepts both version 3 and version 4 packets, but *produces* only version 4 packets.

2.4 ASCII armor

The OpenPGP “ASCII armorized” message is a variation of the Base-64 encoding commonly used to encode messages that may encounter transmission problems when flowing in their initial binary form. Usually these problems occur in networks where devices could interpret binary sequences of data bits as control characters, or more generally speaking are expecting to receive only ASCII printable characters.

The OpenPGP standard specifies one optional data encoding scheme, where binary strings of data can be transformed into ASCII printable character strings. The OpenPGP encoding yields messages in the “ASCII Armor” format. Data encoded into an ASCII armorized message has specific headers that are used by the OpenPGP receiving side to reconstruct the initial binary data string.

Figure 2-1shows how an ASCII armorized message displays. This example was created with an ASCII armorized OpenPGP certificate that resides in a z/OS UNIX file. The display shows ASCII printable characters, which can therefore be copied from the z/OS attached display device screen and pasted onto an ASCII system attached display device. However, since z/OS is an EBCDIC system, this display results from an EBCDIC hexadecimal representation of these printable characters in the z/OS UNIX file. This z/OS ASCII armorized message should therefore be sent to an ASCII system using EBCDIC-to-ASCII character translation (something the FTP commonly does). Conversely, an ASCII armorized message generated on an ASCII system should be sent with ASCII-to-EBCDIC character conversion to z/OS.

Note: Sending or receiving data being “ASCII armored” is an option supported by the Encryption Facility for OpenPGP only for OpenPGP certificate messages.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: IBM Encryption Facility for z/OS V1.2

xo0ERftnwAEEAN34VgMNCxGaTTgXO+LwzWgbydbF/6vkOMzT22KEQYI+JIB7KZBIizRev7CeG2PH
r+hkiSkctwLAKFgJjsVWBzJz/diwpRBVfL39FlhFzpoZuagrdcEYjTJPzyzSI/POsvSugrEy8Ly5
wOuIK6pNeIvASZlsDT345rq13YyEJ9Z1ABEBAAHCtQQfAQIAHwUCRjofPAIeAQIbDwULAgkIBwUV
AgoJCAMWAQICF4AACgkQo8sMavB6vg/S6wP+L7zZfqL3Egc6r7HVC/WhbOQ7ZjX84DWYy7VLWN+y
wpJaiK+neOAVyOkOnotCRdoF8h1mgrKWOQF7N8ILEpytW6fdZxDzEC2DVjlXuXbOOUND4ZdGwKa5
fxDcsCPamDiimFig2cxolk1qJtPYAn66hNip1OdNiRqBxHes2ghHaPfNK2thcHBlbGVyIDxubyBj
b21tZW5OPiA8a2FwcGVsZXJAZnIuaWJtLmNvbT7CtQQTAQIAHwUCRjofPQIeAQIbDwULAgkIBwUV
AgoJCAMWAQICF4AACgkQo8sMavB6vg9MuAP+OvOtwyx8pSINBzj2dL5bZc5jDdBiar5rsMXIvM9/
wfeTvrD6gCObN6KHvtOLNwffu1yOvGrHqtHx1jYPzZlanSOBqOS6PuinrI3QHppG16gpEo8tNx5y
fhEV3yQoXDuh6QrzLE9vB7LbdiVgRJqfqbQq9zOzlWuAgPPMmco2kyo=
=UOFg
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 2-1 ASCII armorized OpenPGP certificate

2.5 OpenPGP certificates

OpenPGP certificates use a specific format that differs from the widely accepted X.509 format. This section presents a high-level description of these two formats, with considerations regarding the characteristics of the environments in which they are used.

2.5.1 A word on the X.509 standards

X.509 is a set of standards that have been adopted by the IETF PKIX group to promote interoperability between entities involved in Public Key Infrastructure (PKI) interactions. A conceptual view of a PKI is shown in Figure 2-2, where clients are asking digital certificates to a Certificate Authority. The Certificate Authority signs and provides certificates to the properly identified and entitled clients. It also publishes certificate revocation information containing a list of previously granted certificates that, for various reasons, must not be accepted anymore as a binding between a public key and a user identity. The prevalent technologies today to make this revocation information available are either the publishing of a Certificate Revocation List (CRL) in an LDAP directory, or the real time interrogation of the Certificate Authority via the Online Certificate Status Protocol (OCSP).

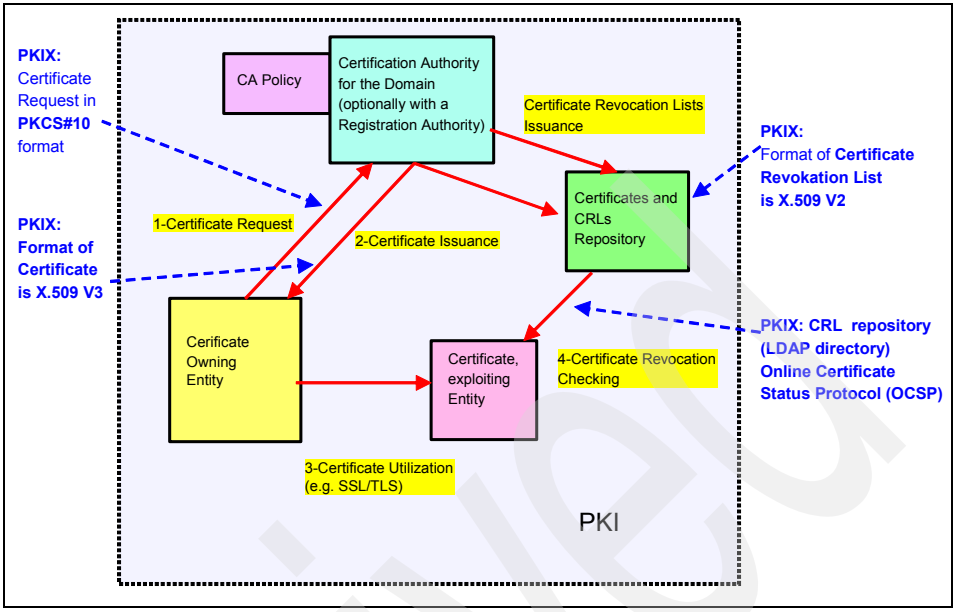


Figure 2-2 The PKIX Public Key Infrastructure

The X.509 V3 certificate

The PKIX recommended format for the digital certificate is the X.509 V3 format, as defined in RFC 2459, which is shown in Figure 2-3.

The certificate contains mandatory fields along with optional extension fields; the digital signature of the certificate by the Certificate Authority (or by the owner of the certificate for a “self-signed” certificate) is calculated using a hash value generated from the totality of the certificate’s fields.

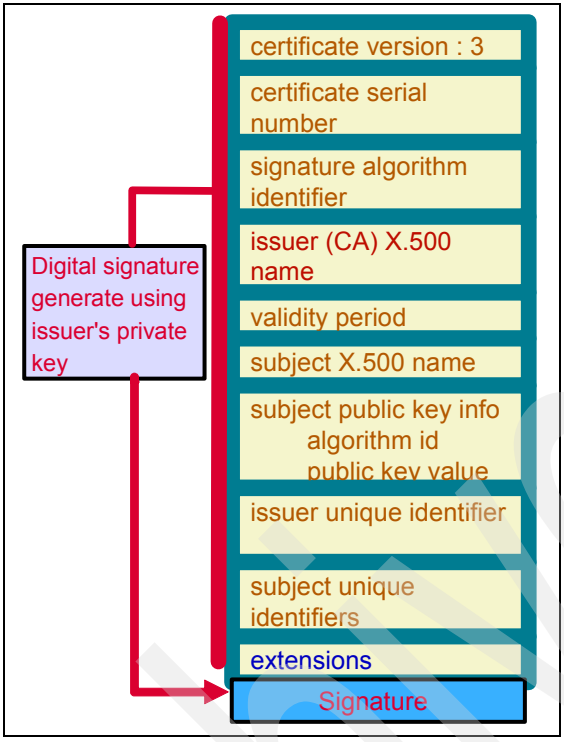


Figure 2-3 The X.509 V3 digital certificate

2.5.2 The OpenPGP certificate

The OpenPGP certificate can be thought of as a message containing specific packets that are similar to the X.509 certificate fields. A high-level graphical view of an OpenPGP certificate is given in Figure 2-4.

One can retrieve in this certificate the public key to be bound to the user ID; however, be aware that the key owner may have several different user IDs, each of them being bound to the public key.

Note that the OpenPGP naming model does not exploit distinguished names as proposed in the X.500 standard. It identifies the owner of the certificate with an OpenPGP User ID. The OpenPGP user ID is composed of a user real name, an optional e-mail address, and an optional comment.

Note: Search commands or utilities for OpenPGP userID are actually looking for the specified substring in the string resulting from the concatenation of the userID full name, comment, and e-mail address.

In OpenPGP terminology, the public key is also called the “primary key,” or the “top-level key.” OpenPGP also requires that the primary key pair has signing capability, as DSA keys have. If the user also requires key pairs for encryption only, as is the case for ElGamal keys or RSA keys flagged for key-management only, then the user should provide the corresponding public keys as “subkeys” in the certificate. To ensure that the subkeys belong to the user, they are also signed with the primary key.

Note: An OpenPGP certificate is always self-signed, and may also include other signers’ signatures.

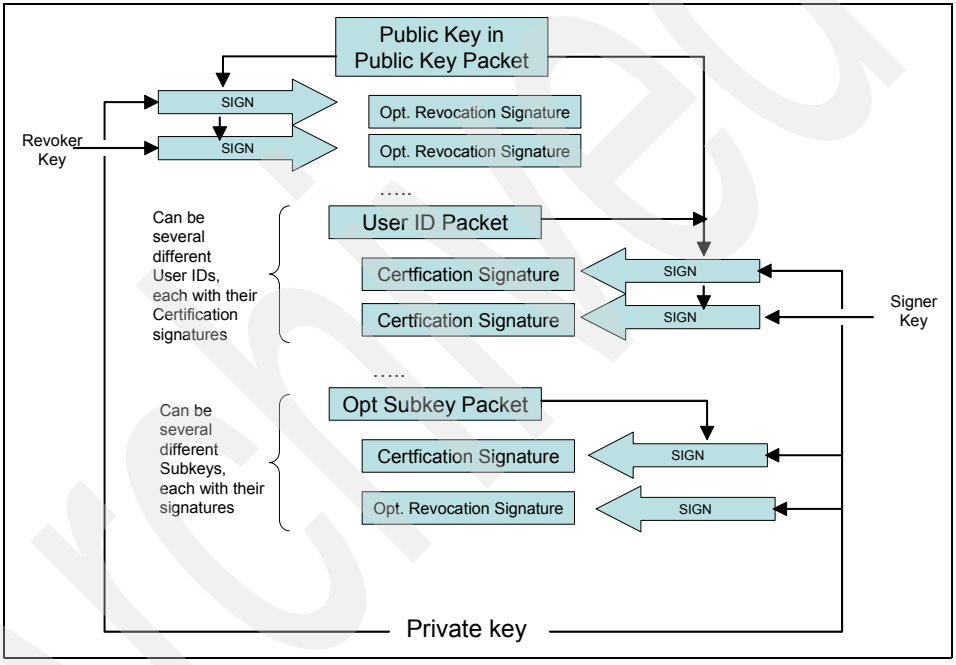


Figure 2-4 The OpenPGP certificate

A very specific feature of OpenPGP is that certification, or revocation, is not intended to be granted by a Certificate Authority. Instead the OpenPGP certificate exhibits sets of other OpenPGP users’ signatures that vouch for the certification or the revocation of the keys. Actually, OpenPGP introduces the “Web of Trust” concept as an alternative to a single centralized trusted Certificate Authority. The Web of Trust concept is explained in 2.6, “The OpenPGP Web of Trust concept” on page 21.

Note also that key bindings can be revoked using revocation signatures that are generated by the certificate owner or users designated as possible revokers.

The OpenPGP Version 4 Signature

Figure 2-5 shows, still at a high level, how OpenPGP signature packets are formed in the OpenPGP certificates. There are still in use today two versions of the OpenPGP signature packet format. Version 4 is the current version, which has superseded Version 3.

Note: The Encryption Facility for OpenPGP accepts Version 3 signatures; however, it produces only Version 4 signatures.

The signature is delivered in a Signature Packet and is created using a combined hash of the public key value and the contents of the so-called “hashed subpackets.” The hashed subpackets contain information similar to the X.509 V3 certificate fields and are also cryptographically bound to the public key value.

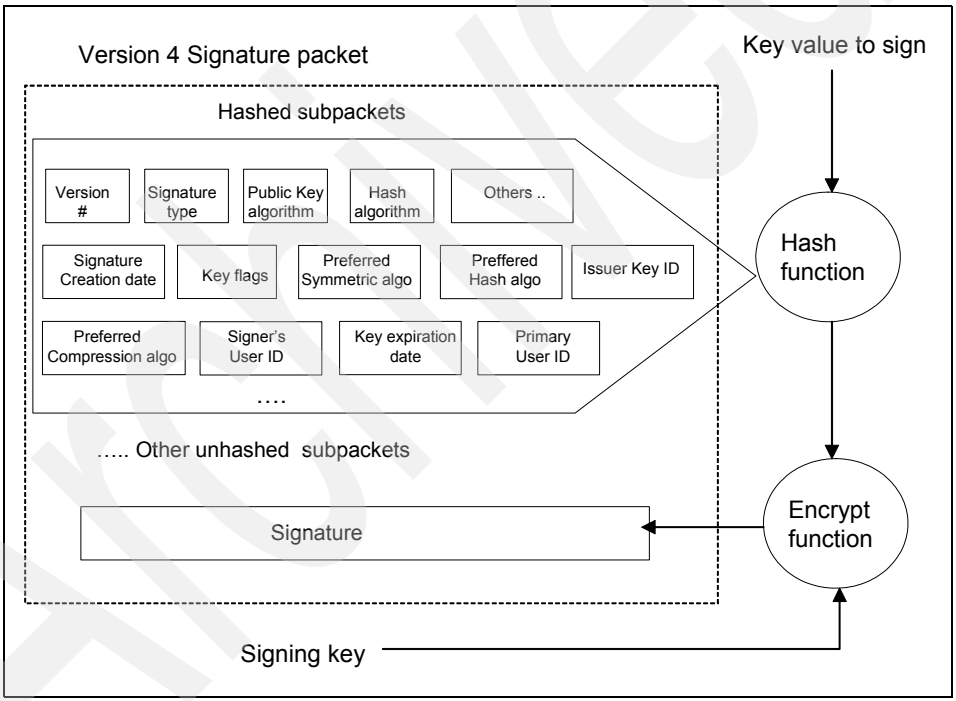


Figure 2-5 Version 4 signatures in the OpenPGP certificate

2.5.3 The OpenPGP keyring

Users keep OpenPGP certificates and keys in files that implement the concept of a “keyring.” Certificates and keys are stored in a keyring as messages, packets, and subpackets. Figure 2-6 is a conceptual view of an OpenPGP certificate and

keys in a keyring. Note that the certificate and keys entries are labelled both with an OpenPGP user ID and a Key ID, the latter being a hash value calculated from the public key value. Note that OpenPGP looks also for signers' certificates in the keyring when it comes to verify a signature.

When a certificate is imported into the OpenPGP keyring it can be granted a trust level (from 0 to 255) by the owner of the keyring, along with comments that are kept with the certificate. This trust level is exploited by OpenPGP implementations that automatically attempt to assess how many chained signer certificates are required to get to a pre-defined level of confidence in the trustworthiness of a signature.

Note: Encryption Facility for OpenPGP does not exploit the certificate trust level beyond keeping it stored locally with the certificate.

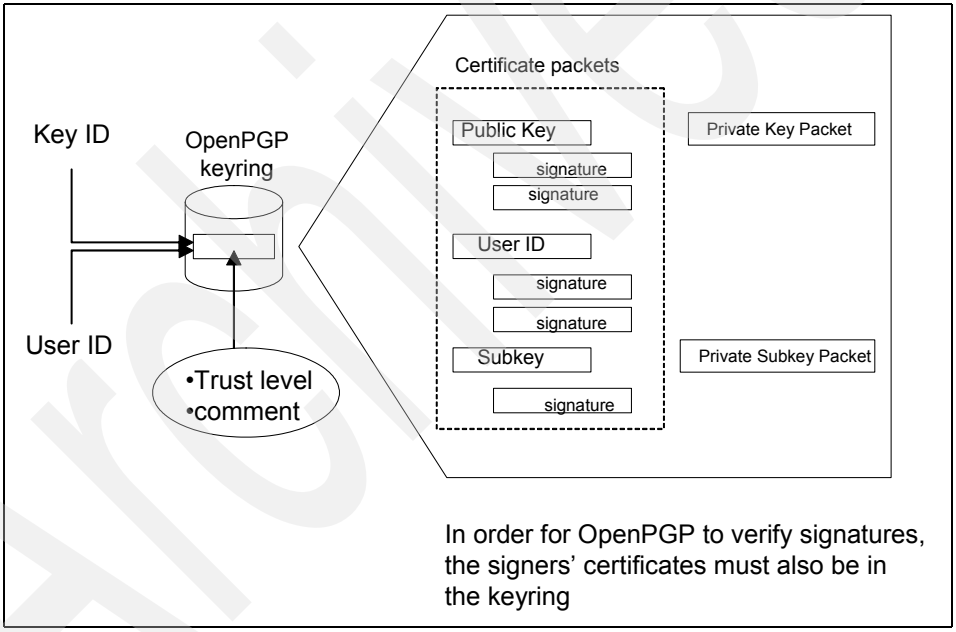


Figure 2-6 Conceptual view of the OpenPGP keyring

2.6 The OpenPGP Web of Trust concept

This section presents a high-level description of the Web of Trust concept. Note that this concept does not have any support implementation in the Encryption Facility for OpenPGP.

As already mentioned, the objective of OpenPGP is not to rely on a central Certificate Authority to establish a trusted binding between a public key and its owner's identity, although this could be done. Instead, the binding between keys and user IDs is still achieved via digital signature; however, these are other OpenPGP users' signatures and there might be several different signers for one single certificate. The idea is that when one receives an OpenPGP certificate signed by many other users, then one can find among the signers users that he or she trusts. We are still dealing here with the axiom of trust in the signer as implemented in the X.509 PKI, except that who is trusted as a signer here may vary based on who is receiving the certificate and the individual trust relationship he or she has with the signers.

From a practical standpoint, once you have generated an OpenPGP key pair and certificate (subkeys can be added later), you may want to send the certificate to somebody known to be a trusted signer and ask this user to sign and return the signed certificate to you. This assumes, of course, that the signer was able to trust, by some method, the certificate as belonging to you. The signer is now a "trusted introducer." In some organizations a company Security Officer can be appointed as being the trusted introducer for the company.

A consequence of this approach is that one has to keep on his or her keyring an adequate collection of trusted potential signers' certificates in order to maximize the chances of being able to validate any new certificate being received. Trusted introducers' certificates kept in a key ring can also be signed by the owner of the keyring to enforce their trustworthiness.

In order to automate the process of establishing as efficiently and as automatically as possible the "trust paths" in a Web of Trust kind of environments, some OpenPGP implementations also make use of an external OpenPGP key server to retrieve missing signer certificates.

2.7 Considerations for OpenPGP implementation

This section identifies considerations that apply to the OpenPGP as implemented by the Encryption Facility for z/OS.

2.7.1 Hybrid X.509 and OpenPGP certificates and keys

The Encryption Facility for OpenPGP does use OpenPGP certificates to exchange public keys with other OpenPGP-compliant partners. It also internally exploits the support built into z/OS for X.509 certificates and keys. This is further explained in 5.5, "Encryption Facility for OpenPGP and digital certificates" on page 70.

2.7.2 OpenPGP concepts that are not implemented

The following OpenPGP concepts are not supported in the Encryption Facility for z/OS V1.2.

Revocation

The Encryption Facility for openPGP does not provide a certificate revocation function. It does warn the user, however, when a revoked key is encountered.

Signing other users' certificates

The Encryption Facility for OpenPGP does not support signing other users' certificates, and therefore the z/OS user cannot participate in a Web of Trust as a trusted introducer or any other signing entity.

Web of Trust

The Encryption Facility for OpenPGP does not provide any functions that would allow automatically retrieving trust paths between certificates beyond looking for signers' certificates in the local OpenPGP keyring, nor does not provide ways of getting information from a key server.

3

Java and Java cryptography

This chapter is geared to readers who are not familiar with the Java environment. The OpenPGP support provided through the Encryption Facility product is a Java-based application, so this overview of Java cryptography topics is relevant to an overall understanding of the OpenPGP environment.

This chapter also provides a detailed description of the Java cryptographic APIs and services, including how and when they are invoked.

3.1 Java infrastructure

This section provides a high-level view of how to run a Java stand-alone application on a z/OS system and how it relates to the Encryption Facility for z/OS V1.2 product. An application is defined as “stand alone” when the Java programs only require a Java Virtual Machine (JVM™) in order to run and do not require a J2EE-compliant middleware environment such as WebSphere® Application Server.

When we talk about “Java,” we usually mean Java Development Kit (JDK™). The JDK product was originally built for developers and it has been by far the most widely used Java Software Development Kit (SDK). There is some level of confusion with the terminology, between the JDK and SDK terms. By definition, JDK is a subset of what is defined as an SDK in the general sense, since SDK refers to the full package, composed of additional software, such as application servers, debuggers, and documentation.

JDK 1.5.0 is the level of JDK currently available on z/OS and it is what we use to run Java programs on z/OS. JDK provides a full Java Runtime Environment (JRE™) that consists of a Java Virtual Machine and all of the class libraries that will be used in the production environment, as well as libraries useful to developers.

The z/OS Java implementation provides the same full function Java APIs as on all other IBM platforms. In addition, the z/OS Java program products have been enhanced to allow Java access to z/OS-specific data access methods. Also, Java on z/OS provides a z/OS implementation of the Java Native Interface (JNI™).

The Java implementation on z/OS, as on other platforms, includes an Application Programming Interface (API) and a Java Virtual Machine (JVM) to run programs. The existence of a Java Virtual Machine means that applications written in Java are largely independent of the operating system being used.

The Java Virtual Machine runs in the z/OS UNIX environment. This is largely because the JVM kernel itself is partly based on the C/C++ programming language that relies on the z/OS UNIX System Services.

The JDK is installed in an HFS file on the z/OS system (for example, in /usr/lpp/java/J5.0) and is made available through the environment variables setup provided in /etc/profile, which can be overridden by the setting in the .profile file in the user’s home directory.

After the installation of the JDK, your path should contain the binary directory where Java is located. The following environment variables should be set or exported when using Java:

```
PATH=/usr/lpp/java/J5.0/bin
LIBPATH=/usr/lpp/java/J5.0/include
JAVA_HOME=/usr/lpp/java/J5.0/
```

The PATH statement is an environment variable to tell a UNIX-like system where the programs (binaries) are located, in this case for the application called *java*.

You can always verify if Java is installed on your system by entering the **java -fullversion** command from your OMVS session. Figure 3-1 is an example of the command output.

```
java -fullversion
java full version "J2RE 1.5.0 IBM z/OS build pmz31dev-20070201
(SR4)"
```

Figure 3-1 java -version command sample

Executing a Java application is no different than executing a traditional application. Java has a main method, which is similar to a main entry point of a traditional COBOL or PL/1 load module. The main method is the first method to be executed when the application starts, and it may have parameters fed to the application. Another characteristic is that a Java stand-alone application runs in its own JVM, which is actually started first, before the first statements of the application are executed.

Java source code is contained in the .java HFS file. You can use the **javac** command to compile; what you obtain as a result is a .class file that is in a byte code format. Class files can be compared to objects in a traditional environment. Classes are portable and should be able to be executed on any platform. Class files can be archived into .jar files for compression and management simplicity.

Java relies on the CLASSPATH environment variable to tell the JVM and other Java applications where to find the class libraries, including user-defined class libraries, so whenever a Java application refers to a user-defined class, the Java runtime environment searches for the class in the class path. The class path is a list of directories, jar or zip files.

This means that when you are executing the Java code from an OMVS session, the JVM can use the classpath facility and you can point directly to the jar file containing the main method instead of specifying the full class path. The jar file contains a *Manifest* that automatically will have a pointer specifying what the main method to be executed is and where it is located. On the other hand, if you

execute the Java code using a facility such as JZOS Java batch launcher, then you need to refer to the fully extended name of the class file containing the main method.

When designing a Java application for z/OS, a key consideration is whether the application will run as a batch program or an online program. In some cases, the decision is obvious, but most applications can be designed to fit either model. The most obvious reason for using batch is the need to process large amounts of data in a manner that does not require interaction with humans, or to run a program that does not need to provide a response back immediately. On the z/OS platform, stand-alone Java programs can be executed in a number of ways:

- In a UNIX shell command, optionally submitted to run in the background, by using the **&** parameter.
- In JCL running BPXBATCH to run programs in a z/OS UNIX environment.
- In JCL running BPXBATSL to run programs in a z/OS UNIX environment that access z/OS data sets.
- Using the JZOS job launcher to run programs in a z/OS UNIX environment.

Important: We recommend using the JZOS facility to run the Encryption Facility programs. For this reason, this document will focus on the JZOS support to run Java application on z/OS.

3.1.1 JZOS

The JZOS job launcher and runtime APIs are the latest inclusion in the IBM Java SDK distribution for z/OS.

JZOS includes Java classes that make console communication with Java applications easy. Instead of implementing the console communication via Java Native Interface (JNI) calls, JZOS provides a framework to register a *listener* for interaction with operators using WTOs. It also provides a method to write messages to z/OS system logs directly from Java applications. Using this framework, Java programs are able to write messages to the z/OS system log when they require special attention from the operator. This can be used as a means to interface with the system automation tools to monitor the status of the running Java batch programs. It is also possible to receive commands from the operator while Java batch jobs are running (through a **`modify`** command). Consequently, it is possible to program Java programs to change behavior depending on the commands received from the operator without restarting the job.

JZOS enables Java programs to be integrated seamlessly with other job steps within a job. Execution of job steps is typically controlled by *return codes* from the previous job steps. JZOS reliably delivers the exit status of Java programs to the following steps in a job, which allows for inclusion of Java program steps in other z/OS utilities and programs.

Another big advantage of JZOS is its full support for DD statements. Because Java programs launched via JZOS run in the same address space as any other steps, Java programs are able to access DD statements specified in a job. For example, it is common to allocate a temporary data set to store output from a step and be used by the following steps in a job. Using JZOS, Java programs are able to read and write from such data sets, which makes Java programs more appealing when considering replacing legacy applications.

The JZOS batch launcher directs stdout and stderr input streams to the standard z/OS data sets and the JES SYSOUT data set. Also, Java programs are able to read from stdin in standard z/OS data sets. Consequently, operators are able to monitor the output from Java programs via System Display and Search Facility (SDSF), just like monitoring any other job steps in z/OS environments.

Table 3-1 summarizes DD names that are used by the JZOS batch launcher.

Table 3-1 DD names used by JZOS batch launcher

DD name	Description	Required
SYSOUT	Messages from the batch launcher and any system messages that are written to the UNIX stderr file descriptor.	YES
SYSPRINT	Any system messages that are written to the UNIX stdout file descriptor. This is normally used.	NO
STDOUT	The output from Java *System.out*. This data is translated to the JZOS_OUTPUT_ENCODING codepage.	YES
STDERR	The output from Java *System.err*. This data is translated to the JZOS_OUTPUT_ENCODING codepage.	YES
STDENV	A UNIX shell script used to configure environmental variables.	YES
STDIN	The input to Java *System.in*. This data is translated from the JZOS_OUTPUT_ENCODING codepage to the default Java file.encoding codepage.	NO
MAINARGS	Can be used to supply arguments to the main Java class.	NO

Table 3-2 summarizes additional environmental variables that control the behavior of JZOS and Java applications launched via the JZOS job launcher.

Table 3-2 Description of JZOS environmental variables

Environmental variables	Description
JZOS_ENABLE_MVS_COMMANDS = {true | false}	This variable specifies whether JZOS will allow processing of the MVS™ operator commands START(S), MODIFY(F) and STOP (P). If set to 'false,' the JZOS batch launcher will not respond to MVS operator commands. The default is true.
JZOS_OUTPUT_ENCODING = {codepage}	This variable specifies the codepage used by JZOS for its output to stdout and stderr. If not specified, the default codepage for the current locale is used (IBM-1047 EBCDIC codepage).
JZOS_ENABLE_OUTPUT_TRANSCODING = {true | false}	If set to false, raw bytes written to System.out and System.err are not transcoded to the JZOS_OUTPUT_ENCODING codepage.
JZOS_GENERATE_SYSTEM_EXIT = {true | false}	If set to true, JZOS will generate a System.exit() call upon completion of main(). This will cause JZOS to complete, even if there are active non-daemon threads. The default is false, which means JZOS will wait for non-daemon threads to complete before exiting.
JZOS_MAIN_ARGS = {classname and arguments} JZOS_MAIN_ARGS_DD = {ddname | main args}	Allows for additional arguments for the main method that JZOS invokes.

Figure 3-2 shows sample JCL that integrates traditional batch job steps with a Java application.

```
//JZOSRUN JOB REGION=300M,CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//PROCLIB JCLLIB ORDER=USER.JZOS.SAMPJCL
//*********************************************************
//* EXECUTE REXX EXEC TO WRITE DATA TO A TEMP DS REXXOUT
//*********************************************************
//GENREC   EXEC PGM=IRXJCL,PARM='GENRCDS'
//SYSTSIN  DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//OUTFILE  DD DSN=&&REXXOUT,DISP=(NEW,PASS),
//            SPACE=(TRK,(2,1))
//SYSEXEC  DD DSN=USER.EXECS,DISP=SHR
//*********************************************************
//* SORT TEMPORARY DATA SET (REXXOUT) AND WRITE RESULTS
//* TO ANOTHER TEMPORARY DATA SET (SORTDATA).
//*********************************************************
//SORTSTEP EXEC PGM=SORT
//SYSIN DD *
    SORT FIELDS=(1,80,CH,A)
/*
//SYSOUT DD SYSOUT=*
//SORTIN  DD DSN=&&REXXOUT,DISP=(OLD,DELETE)
//SORTOUT  DD DSN=&&SORTDATA,DISP=(NEW,PASS),
//            SPACE=(TRK,(2,1))
/*
//*********************************************************
//* JAVA PROGRAM USING JZOS. JAVA PROGRAM READS FROM THE TEMP DS
//* (SORTDATA), AND WRITE TO A PERMANENT FILE (USER.DATA).
//* This is the default procedure that will call the JZOS JVMLDM50
//* program.
//*********************************************************
//JAVA EXEC PROC=JVMPRC50,
// JAVACLS='ProcessRecords'
//INDATA  DD DSN=&&SORTDATA,DISP=(OLD,DELETE)
//OUTDATA DD DSN=USER.DATA,DISP=SHR
//STDOUT  DD SYSOUT=*
//STDERR  DD SYSOUT=*
//STDENV  DD DSN=USER.JAVAENV(JPRPTY50),DISP=SHR
/*
```

Figure 3-2 Sample JCL with JZOS to run Java application

3.1.2 JRIO

The Java Record I/O (JRIO) library lets Java applications use traditional mainframe data access methods that the Java I/O library does not support. In addition, it provides the record-oriented view of the data stored in a data set and provides facilities to access the records sequentially, randomly, or using keys. JRIO is an integral part of all SDKs available for the z/OS platform. Refer to the following Web site for product information and instructions for downloading and installation:

http://www.ibm.com/servers/eserver/zseries/software/java/jrio/overview.html

The JRIO library enables Java applications to access the following types of mainframe file systems:

- Partitioned Data Set (PDS) and Partitioned Data Set Extended (PDSE)
- Sequential files
- Virtual Sequential Access Method (VSAM) data sets (only KSDS)
- Hierarchical File System (HFS or zFS)

3.1.3 Cryptographic facilities

Now that we have described how you can execute a Java program on a z/OS system, the next step is to see how the Java environment naturally exploits the hardware cryptographic facilities provided by the system.

Figure 3-3 is a schematic view of the Encryption Facility implementation in the System z environment and depicts the overall layout of how the components interact. 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.

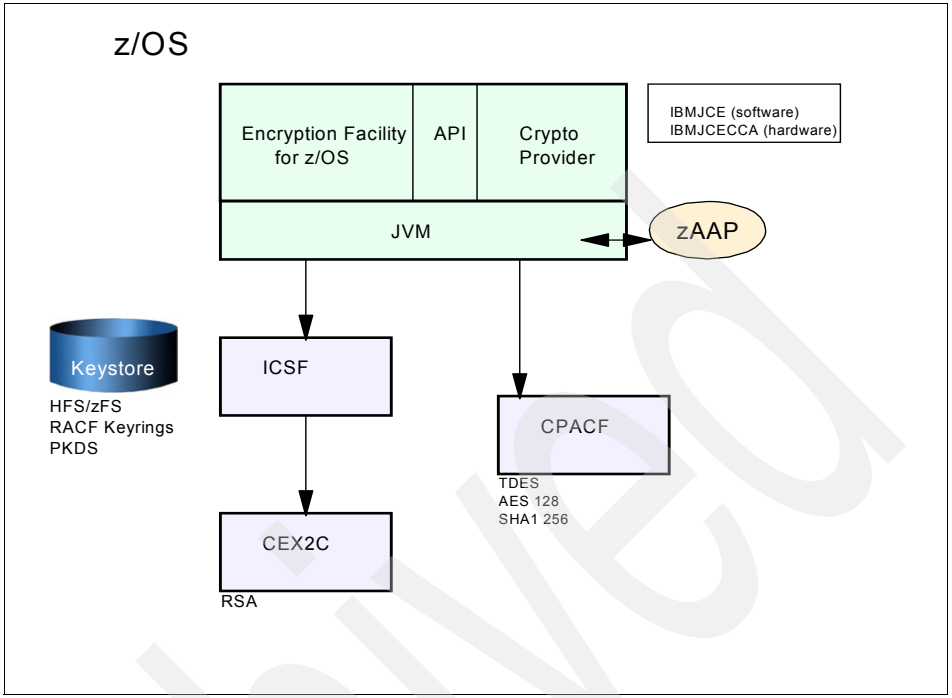


Figure 3-3 Overview of Java and Encryption Facility

The interaction between the components is as follows:

The Encryption Facility runs as a Java application and the recommended way to run is through the JZOS support. Starting the Java application creates the JVM, which can take advantage of the zAAP processor to run the Java code.

As encryption services are requested, cryptographic services APIs are provided to the application that, via the JVM, call the Java cryptographic services providers that are defined in the environment. Depending on the configured providers, JVM will give control to the crypto providers, either software (when com.ibm.crypto.provider.IBMJCE is specified) crypto service or hardware (when com.ibm.crypto.hdwrCCA.provider.IBMJCECCA is specified). Details on crypto providers are covered in the next section.

When hardware cryptography is called, the JVM interacts with ICSF which ultimately drives the operations at the Crypto Express2 Coprocessor (CEX2C) or at the CPACF facility that resides in each processing unit of the z9, z990 and z890 servers, although some crypto goes directly to CPACF without going through ICSF.

3.2 Java cryptography

In this section we provide a general explanation of how standard cryptographic services are provided to Java applications. After briefly reviewing the available APIs and relevant service providers, we focus on the cryptographic services that the Encryption Facility for z/OS OpenPGP support exploits, how they are made available, and offer some recommendations based on our experience.

3.2.1 Java cryptographic APIs and services

Following is a description of the Java interfaces and services for cryptographic functions.

What they are, what they provide

Complex security services, including cryptographic services, have been available for Java applications for many years, as add-ons to the different SDK implementations. Starting with SDK 1.4.0 these functions are integrated in the base Java 2 framework, meaning that the functions are part of the set of common APIs made available to Java applications across platforms (also termed the Java 2 framework).

It is important to understand the underlying implementation architecture and what setup is required to make these services actually available to applications. Although many of the security services provided today deal indirectly with cryptography, the Java 2 framework offers a specific API that gives explicit access to elementary cryptographic services such as data encryption or decryption. This is the JCE (Java Cryptographic Extension) API.

The JCE API provides access to:

- Digital signatures
- Hashing
- Symmetric encryption/decryption
- Asymmetric encryption/decryption with RSA
- Key agreement with Diffie-Hellman

Later sections describe how Encryption Facility for z/OS OpenPGP support exploits the JCE services.

Note: See Appendix A, “Some encryption basics” on page 223 for a refresher on cryptography and cryptographic algorithms.

The underlying architecture

The elements of the underlying architecture are described in this section.

Service providers

The functions offered by the API are performed by calling Java service classes (also termed *engines*), which themselves rely on *service providers* to implement the required functions and algorithms. This implementation architecture supports use of plug-replaceable components. Cryptographic service providers are available from Sun™, IBM, and other vendors. Each one of these providers is a specific implementation of the JCE, JSSE or other APIs.

IBM provides two JCE providers, which come as .jar files:

- The IBMJCE cryptographic provider
- The IBMJCECCA cryptographic provider

Further details on these providers are given in 3.2.2, “The IBM JCE providers”.

Keystores

As for any implementation of cryptographic services, keys need to be kept and managed in secure repositories. Java uses the concept of a *keystore*, where individual secret keys are kept under a label, called the *alias*, and encrypted with a password. The repository itself is also protected with its own *store password*.

The keystore concept can be implemented in several ways – for example, using a flat file, or using other technologies, as is the case with the z/OS ICSF PKDS. The *keystore type* specifies what keystore implementation to consider. The keystore types supported in an installation are also related to the cryptographic providers that are used.

Policies

As for any cryptographic product, the SDK is natively limited, due to governmental regulations, regarding the length of the keys that can be used. This limitation is implemented via the Java cryptography *jurisdiction policy files*. Extension of the key length is granted by replacement, with due control, of the base policy files by other files containing *unrestricted policies*.

3.2.2 The IBM JCE providers

As mentioned earlier, Encryption Facility for z/OS OpenPGP support exploits the JCE services. These services are implemented in either of the IBM JCE providers.

The IBMJCE provider

The IBMJCE provider implements the JCE services by software only. It provides the following services at the SDK V5 level:

- Digital signatures with the DSA or RSA algorithm, in combination with the hashing algorithms listed in the next item.
- Hashing with SHA1, SHA256, SHA384, SHA512, MD2, MD5.
- Symmetric encryption/decryption with DES, triple DES, AES128, AES192, AES256, PBE, Blowfish, Mars, RC2, RC4.
- Asymmetric encryption/decryption with RSA and ElGamal.
- Key agreement with Diffie-Hellman.
- RSA with Optimal Asymmetric Encryption Padding (OAEP).
- Hash-based Message Authentication Code (HMAC) with MD5, SHA1, SHA256, SHA384 and SHA512.

The keys used by the IBMJCE provider are generated and managed with the Java *keytool* utility. The keytool utility comes with the provider and therefore requires that IBMJCE be specified in the providers list.

The IBMJCECCA provider

The IBMJCECCA provides those JCE services that can be performed by the hardware cryptographic coprocessors through ICSF in z/OS. It actually calls ICSF for the IBM Common Cryptography Architecture services and therefore requires ICSF to be in operation and to be set up to provide these services.

Note that IBMJCECCA provides, in addition, a software implementation of variants of these services for variants that are not supported by the hardware cryptographic facilities of System z.

Information on the IBMJCECCA provider can be found at:
`http://www.ibm.com/servers/eserver/zseries/software/java/j5jcecca.html`

Note: IBMJCECCA provider extends and replaces the former IBMJCE4758 provider that was available with prior releases of the IBM SDK. At this time the IBMJCE4758 and the IBMJCECCA providers are functionally equivalent; however, we strongly recommend that you install and use the IBMJCECCA.

IBMJCE4758 provider uses its own keystore types JCE4758KS and JCE4758RACFKS. These keystores are supported by the IBMJCECCA provider. Additional details are presented in “Keystores that can be used with the IBM JCE providers” on page 42.

ICSF is a started task that requires at least one CPACF to be enabled on the system and to be allocated two VSAM data sets to be used as optional key repositories. The Public Key Cryptographic Data Set (PKDS) can be used to securely store the RSA keys to be used with the IBMJCECCA provider.

The other data set is the CKDS, which it is not used when working with the Encryption Facility, although it is required by ICSF. Figure 3-4 on page 38 is a high-level view of the infrastructure layout. For further information on ICSF refer to *z/OS Cryptographic Services ICSF Administrator's Guide,* SA22-7521.

The IBMJCECCA provider supports the following algorithms:

- Digital signatures via RSA and DSA
- Hashing - SHA1, MD2, MD5, SHA-256
- Keystore - Symmetric and asymmetric keys protected by triple DES
- Symmetric algorithms - DES, triple DES, PBE, AES128
- Cipher modes - ECB, CBC, CFB, OFB, PCBC
- Asymmetric algorithms - RSA
- HMAC - MD5, SHA1
- Pseudo random number generation

Note: The DSA algorithm is performed by hardware on the systems z800 and z900 only. Beginning with systems z890 and z990 there is no longer hardware support for DSA in the cryptographic coprocessors.

IBMJCECCA provides the *hwkeytool* to generate and manage keys in a format appropriate to be used in cryptographic hardware operations.

Note: Since ICSF is transparently called by IBMJCECCA, RACF protection for ICSF callable services (via the CSFSERV class of profiles), and keys in the PKDS (via the CSFKEYS class of profiles), can also be achieved.

Access control is performed on the basis of the RACF ID of the user that started the Java application.

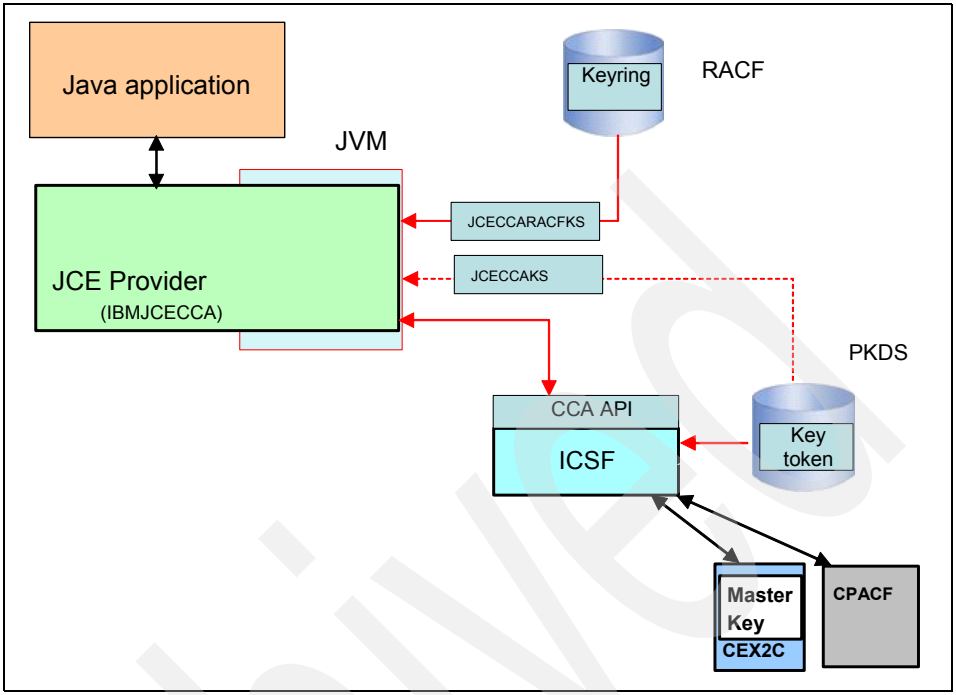


Figure 3-4 IBMJCECCA layout

Since there is a requirement to use CFB chaining for OpenPGP symmetric encryption, which is a chaining mode not natively supported in the CPACF, Figure 3-4 shows an invocation of the CPACF only through ICSF, where ICSF takes care of the CFB chaining process.

Dependencies on hardware cryptography technologies

The Encryption Facility for z/OS OpenPGP support gets a hardware assist for cryptographic operation from the following hardware cryptographic coprocessor technologies, via the IBMJCECCA provider:

- The Cryptographic Coprocessor Facility (CCF) for DES, triple DES, SHA-, RSA (with a key length <= 1024 bits, and without key generation), DSA
- The PCI Cryptographic Coprocessor (PCICC) for RSA (with a key length up to 2048 bits)
- The CP Assist for Cryptographic Functions (CPACF) for DES, T-DES, AES128, SHA-1, SHA-256
- The PCIX Cryptographic Coprocessor (PCIXCC) or Crypto Express 2 Coprocessor (CEX2C) for RSA (with a key length up to 2048 bits)

The availability of these technologies depends on the System z model the Encryption Facility for OpenPGP executes in. Figure 3-5 describes the cryptographic technology migration path when moving from zSeries to System z models.

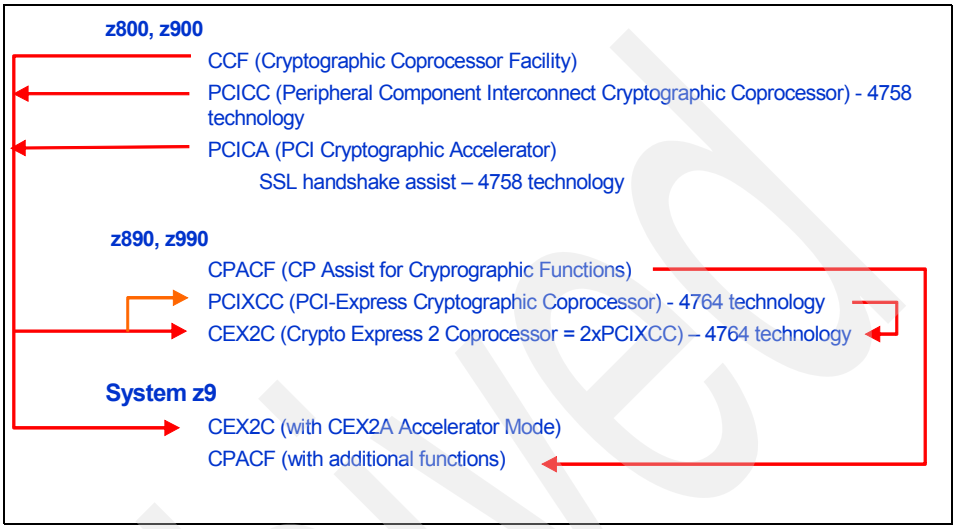


Figure 3-5 Hardware cryptography technologies migration path in System z

DES algorithms and IBMJCECCA

The DES and triple DES algorithms work on successive blocks of data of 64 bits each. Each block is encrypted or decrypted in a process that involves merging the output of a previous block's encryption or decryption output within the current flow of data to encrypt or decrypt. This is called *chaining*. Several variants of chaining exist (including Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and others) that are also offered by the IBMJCECCA provider.

For more details refer to:

`http://csrc.nist.gov/CryptoToolkit/modes/`

Note: Only the DES or triple DES algorithms with CBC chaining (the latter with APAR OA19177 correction installed) are actually supported by the System z hardware cryptography. Other chaining variants, if needed, are provided by software in IBMJCECCA.

The Encryption Facility for z/OS OpenPGP support uses triple DES with CFB chaining only, and therefore uses hardware cryptography (with APAR OA19177).

For short bursts of data, DES or triple DES hardware cryptography may prove to be slower than software cryptography. The user can set up a threshold in IBMJCECCA, the *clip level*, that specifies the minimum data length for which encryption, or decryption, should be performed by hardware rather than software. For data lengths below the clip level the IBMJCECCA provider performs the encryption or decryption by software only.

The clip level is a system property called *ibm.DES.usehdwr.size*; it has a default value of 60. Thus, by default, any DES or triple DES processing where the data size is 60 bytes or less will be performed using software cryptography and any data size greater than 60 bytes will be performed using hardware cryptography.

You can adjust the clip level simply by changing the system property to the desired level. If you set the system property to 0, all DES or triple DES processing will be performed using hardware cryptography. If you set the system property to -1, all DES or triple DES processing will be performed using software cryptography.

Important: We recommend to always force the triple DES algorithm to be run by hardware when using the Encryption Facility for z/OS OpenPGP support. This can be done at the invocation of the product using the Java command to set up the property:

```
java -Dibm.DES.usehdwr.size=0 ...
```

Or by setting an environment variable:

```
export IBM_JAVA_OPTIONS="$IJO -Dibm.DES.usehdwr.size=0"
```

Important: Based on our experience, IBMJCECCA today does not pass the request to another JCE provider, such as IBMJCE, for instance, if it fails to get access to the cryptographic hardware facility of the system. It is therefore mandatory to have ICSF up and running in the system, along with the proper coprocessors enabled.

3.2.3 Setting up the list of providers

Provider classes are shipped as .jar files that are, for the IBMJCE and IBMJCECCA, delivered with the SDK. The list of providers accessible to the SDK is specified in the file <java-home>/lib/security/java.security.

The order of the providers specified within the list establishes the look-up order, and therefore a provider preference order, when an application requests a

cryptographic service without specifying the provider to use. As an example, consider when the list in the java.security file specifies:

```
security.provider.1=com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
security.provider.2=com.ibm.crypto.provider.IBMJCE
```

This implies that as many as possible algorithms will be run using the hardware cryptography of the system. Requests for algorithms that are not supported by IBMJCECCA are automatically forwarded to the next provider in line, that is, IBMJCE (and the algorithm is therefore executed by software only). However, due to the large scope of algorithms covered by IBMJCE, if IBMJCE had been first in the list, then no request would be forwarded to IBMJCECCA and there would not be any use of the hardware cryptography.

Note: Currently there are circumstances where the preference order of the providers may prevent required services from being provided. This can occur:

- Because the IBMJCECCA provider does not fail over to the next provider if a problem is encountered with the hardware cryptography.
- Due to the generation of keys in the keystore, as explained in “Using the Java utilities” on page 48.

Encryption Facility for z/OS OpenPGP configuration file

The OpenPGP support code can get options from the ibmef.config configuration file. The file is located in the directory identified in the -homedir command option when invoking the Encryption Facility for z/OS OpenPGP. Further information on the configuration file is given in 5.6.4, “Options in the configuration file” on page 77.

The provider list option JCE_PROVIDER_LIST

One of these options is the specification of a list of providers that is taken into consideration by the Encryption Facility for z/OS OpenPGP support before it looks up the providers list in the java.security file. The option is *JCE_PROVIDER_LIST* and its recommended value is:

```
com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
```

Note: Our recommendation is that you leave the list of providers in java.security as it is, so as not to interfere with the other Java cryptographic applications that may be running in the system, and set instead the JCE_PROVIDER-LIST with the value shown here.

The random number generator option RNG_JCE_PROVIDER

In a configuration where the system has only the CPACF facilities and no coprocessor card, there is no hardware generation of random number. In that case the recommended value for the RNG_JCE_PROVIDER option is:

```
com.ibm.crypto.provider.IBMJCE
```

If the RNG_JCE_PROVIDER option is not specified, the Encryption Facility for z/OS OpenPGP support selects the first random number generator function available in the list of providers.

3.2.4 Jurisdiction policy files

Allowable cryptography strengths are defined by laws (for example, by U.S. export regulations) and cryptographic products vendors should have a way of controlling this strength at the user's site. JCE key length limits are enforced, transparently to the application developers, by *jurisdiction policy files*. These files are <java-home>/lib/security/local_policy.jar and <java-home>/lib/security/US_export_policy.jar.

These default policy files currently limit the length of the keys to 512 bits for RSA encryption and 64 bits for all other encryption algorithms.

This restriction must be lifted for the Encryption Facility for z/OS OpenPGP support to work with longer key lengths, such as the ones used with triple DES or today's widespread use of RSA. This is done by replacing the two default jurisdiction policy files with *unrestricted policy files*. The unrestricted policy files that we used can be downloaded from:

```
http://www.ibm.com/servers/eserver/zseries/software/java/j5jcecca.html
```

The files we used were FTPed from the Web site, in binary, into the z/OS UNIX files.

Note: There are no unrestricted policy files available for SDK V5; however, according to the Web site, SDK 1.4.2 files are usable with V5.

3.2.5 Keystores that can be used with the IBM JCE providers

The keystore concept can be implemented in different ways, which are reflected in the *keystore type*. The support for a given keystore type depends on the provider in use, and the keystore type is specified when invoking the provider.

Important: The keystore can host either symmetric or asymmetric keys. In the context of use of the Encryption Facility for z/OS OpenPGP support, the symmetric keys are not to be kept in keystores. Therefore, any mention in this document of keys stored in a keystore refers to asymmetric keys only.

It is also important to note that keys stored in the ICSF PKDS are RSA keys only. DSA keys are not supported in the PKDS any longer, beginning with systems z890 and z990.

IBMJCE

IBMJCE supports the following keystore types, which are explained in further details in the following sections:

- JKS - This is the Sun original keystore proprietary implementation in a z/OS UNIX flat file. Keys in the JKS can be managed using the keytool Java utility (for further information on the keytool utility refer to:

  `http://www.ibm.com/developerworks/java/jdk/security/142/secguides/keytoolDocs/KeyToolUserGuide-142.html`

  On z/OS the JKS keystore is backed up by a z/OS UNIX file.

- JCEKS - This is the IBM implementation of the Sun keystore concept, that is, the class java.security.KeyStore class. It is implemented with a strong protection of private keys using triple DES password-based encryption.

  Keys in the JCEKS are managed by keytool as well.

  You can “upgrade” your keystore of type JKS to the type JCEKS by changing the password of a private key entry in your keystore. Note that once you have upgraded your keystore this way, then it becomes usable only via the IBMJCE provider.

- JCERACFKS - This keystore type is backed up by keyrings in RACF (or another compatible External Security Manager that is accessible via the SAF interface). RACF keyrings are used to aggregate RSA or DSA key pairs and certificates for their use by the keyring’s owner. Note that keyrings can also be shared between other authorized RACF users. Further details on the RACF keyring management and exploitation are provided in Chapter 7, “Session key protection: Public key cryptography” on page 99.

  Note that in order to comply with the keystore concept, access to keys kept in the JCERACFKS is still protected with a key entry password and a keystore password. However these passwords are not used because access to the keyring is under control of RACF profiles. Figure 3-6 illustrates the layout of the JCERACFKS, where the application provides the RACF userID and the keyring name to the RACF callable service R_datalib. R_datalib returns the key or certificate if the specified user ID is permitted to these resources. Note

also that the application provides a keystore entry and stored password, which are not used, but which need to be provided and identical.

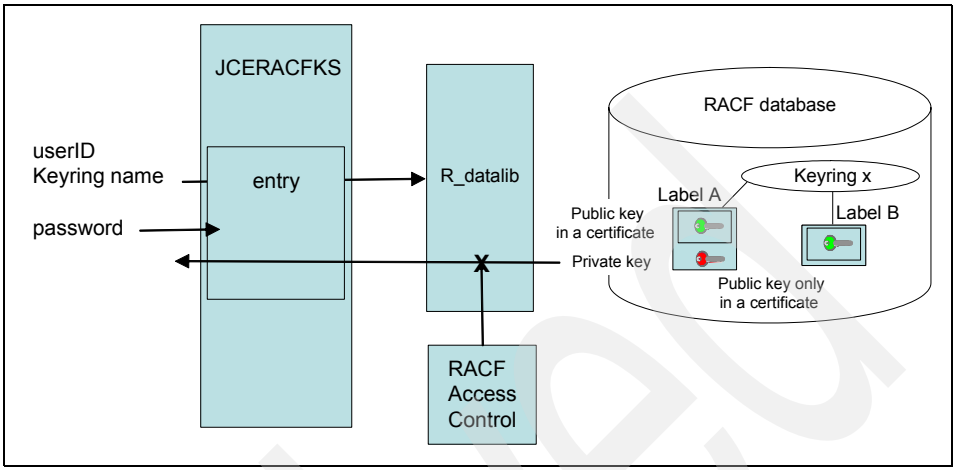


Figure 3-6 JCERACFKS implementation

Note: The JCERACFKS is a read-only keystore. No Java application or utility can write keys in the keystore. The writing of keys in the JCECCARACFKS is actually performed by the RACF administrator, or via delegated privileges, using the RACF RACDCERT command.

IBMJCECCA

The IBMJCECCA provider relies on the use of ICSF and therefore requires keys to be kept in ICSF *key token* constructs. IBMJCECCA therefore uses its own specific keystore types.

- JCECCAKS - This IBM implementation of a Java keystore is intended to only store keys that are to be used with the cryptographic hardware coprocessors. The keys in the JCECCAKS can actually exist in three different forms, depending on the way they have been generated. They can be either:
  - A retained key: The private key of the key pair is kept inside the cryptographic coprocessor itself. Note that IBM does not recommend using retained keys with ICSF.
  - A PKDS key: The key pair is kept in the PKDS, with the private key encrypted with the cryptographic coprocessor's Master Key.
  - A clear key: The keys of the key pair are packaged as ICSF key tokens, without encryption by the cryptographic coprocessor's Master Key.

  When using the JCECCAKS, the clear keys reside in the keystore, with an additional protection of a password, whereas the retained and the PKDS keys

are only pointed at by a label that is stored in a keystore. All key information in the JCECCAKS is designated by the application using a keystore entry alias. This is illustrated in Figure 3-7, where the three key types are shown. Note that only a clear key is returned to the application. The PKDS key is picked up directly by ICSF for the use of the coprocessor and the retained key, by definition, is already inside the coprocessor.

Note: Access to the PKDS and retained keys can be controlled by RACF profiles in the CSFKEYS class on the basis of the RACF user ID of the user who called the Java application. The permission to a CSFKEYS profile is verified by ICSF.

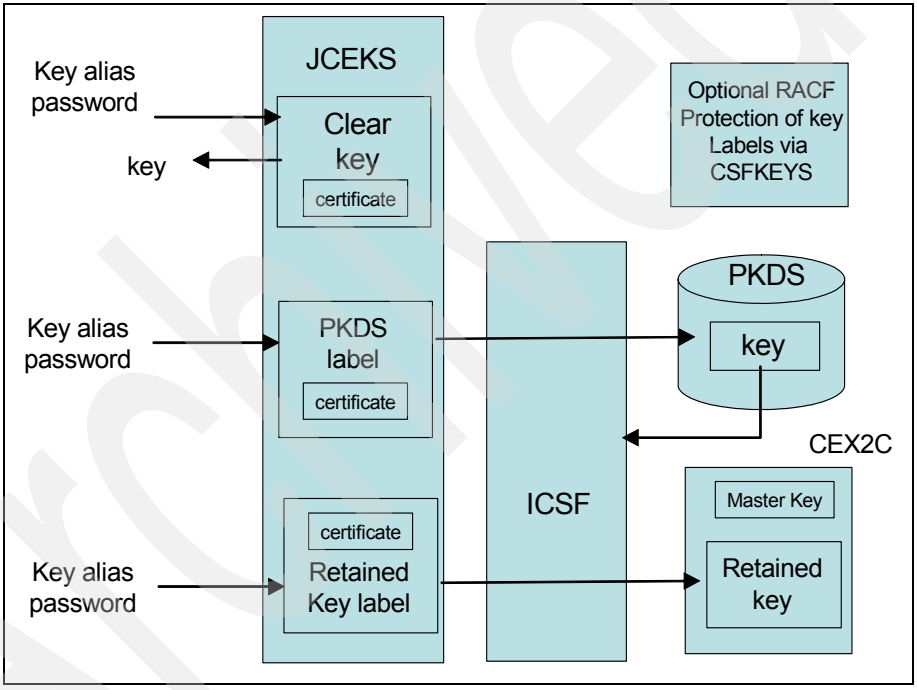


Figure 3-7 JCECCAKS implementation

Attention: ICSF does not offer any means to retrieve the clear value of an RSA private key that has been added to the PKDS. The key remains, whatever happens, encrypted with the cryptographic coprocessor Master Key. Should a recovery of the key be needed, the key generation process that allows creating a backup copy of the key clear value must have been used. Note that, as some key generation methods imply to directly generate the key in the PKDS, it is not possible in that case to get a clear value of the key with which to prepare a backup.

Note that the ICSF PKDS can be used in many different contexts, and it may happen that, when deciding to use the Encryption Facility for z/OS OpenPGP support with the JCECCAKS keystore, RSA keys already reside in the PKDS that you want to use with the Encryption Facility for z/OS OpenPGP support. It is then necessary to make key entries in the JCECCAKS that are mapped to the PKDS keys. This operation is discussed in “Re-synchronizing a keystore with existing PKDS key tokens” on page 50

- JCECCARACFKS - This is a variation of the JCERACFKS, where the keys are still referred to as aggregated in a RACF user’s keyring; however, the keyring contains a PKDS label. The actual key is kept, encrypted with the coprocessor Master Key, in the PKDS.

  This is illustrated in Figure 3-8, where the private key with a RACF *Label C*, connected to *Keyring x*, does not actually reside in the RACF database, as do the other keys in the figure. It is securely stored in the PKDS; the keyring provides a label that is conveyed to ICSF which, in turn, retrieves the actual key in the PKDS.

  The JCECCARACFKS is a read-only keystore, that is, no Java application or utility can write keys in the keystore. The writing of keys in the JCECCARACFKS is actually performed using the RACF RACDCERT command.

  There is no need to “build” a JCECCARACFKS keystore for the Encryption Facility for z/OS OpenPGP support. It is only required to provide the following keywords to locate the contents of the RACF keyring:

-keystore-type	JCECCARACFKS
-keystore	<keyring name>
-racf-keyring-userid	<key ring owner’s user iD>
-rA "	<racf key label in the keyring>
-keystore-password	<password>
-key-password password	<password>

  The keystore and key password can be anything but must be identical.

Important: The user who is to access the keyring must be permitted to the RACF profile IRR.DIGTCERT.LISTRING in the FACILITY class:

- With READ access if the user is the keyring's owner
- With UPDATE access if the user is not the keyring's owner

Getting access to a certificate in a keyring does not imply getting access to its associated private key. Only the owner of the certificate can access the private key. The owner of the certificate is the ID specified in the RACDCERT GENCERT or ADD command.

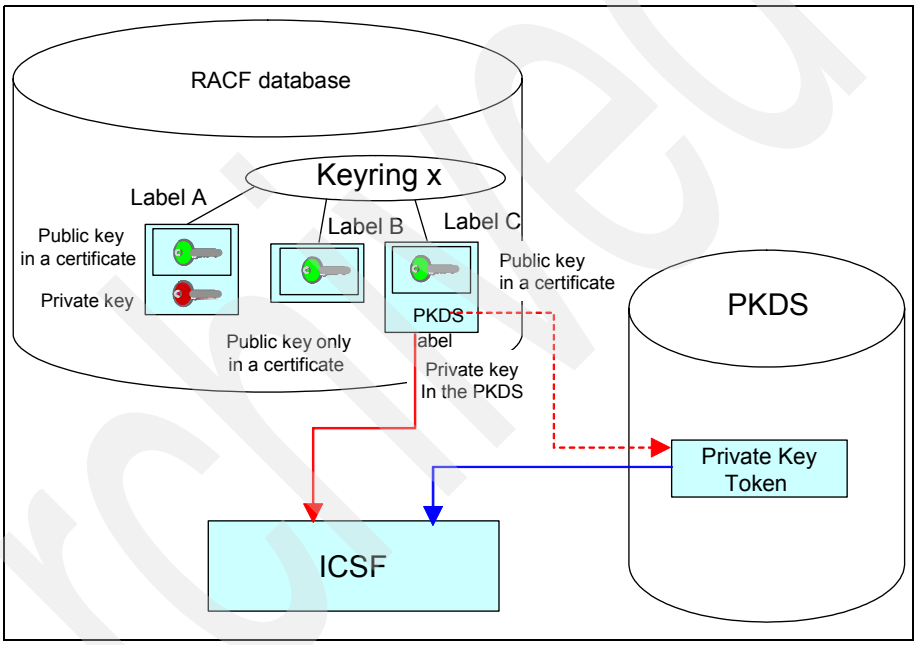


Figure 3-8 JCECCARACFKS implementation

IBMJCECCA and clear keys

The IBMJCECCA can also work with clear keys that are provided by the application in the form of ICSF key tokens. The *hwkeytool* utility can be used to generate these clear keys. Clear keys are kept in the JCECCAKS flat file.

Note: JCECCAKS and JCECCARACFKS keystores are deprecating the use of JCE4758 KS and JCE4758RACFKS. However, for the sake of downward compatibility, the JCE4758... keystore types are supported by IBMJCECCA.

3.2.6 Managing keys in the keystores

Several utilities can be used to manage keys in the keystores depending on the keystore type. Utilities are provided in the IBM SDK, or for the specific case of the JCERACFKS and JCECCARACFKS keystores, key management has to be performed using the RACF RACDCERT command.

Another specific utility is also provided via ICSF ISPF panels (ICSF must be FMID HCR7731, or earlier releases with APAR OA15156 installed).

Using the Java utilities

The IBM SDK comes with two utilities:

- The *keytool* utility, in the IBMJCE provider. This utility is intended to manage keys in the JCEKS, that is, it operates by software only.

  More details on the externals of keytool are provided at:

  `http://www.ibm.com/developerworks/java/jdk/security/142/secguides/keytoolDocs/KeyToolUserGuide-142.html`

- The *hwkeytool* utility, in the IBMJCECCA provider. This utility provides services similar to keytool, but is intended to operate on ICSF key tokens. It is used to manage keys in the JCECCAKS keystore. Information on hwkeytool can be found at:

  `http://www.ibm.com/servers/eserver/zseries/software/java/hwkeytool.html`

Attention: If you have installed the IBMJCECCA hardware provider, be sure when generating software keys that IBMJCE is before IBMJCECCA in the provider list.

Using the RACF RACDCERT command

RACDCERT is a RACF command that is used to manage the asymmetric keys, RSA or DSA keys that can be kept in the RACF database.

The command and the entities it operates on are described in *z/OS Security Server RACF Security Administrator's Guide,* SA22-7683 and *z/OS Security Server RACF Command Language Reference,* SA22-7687.

The command has keywords for generation of key pairs, generation and handling of X.509 V3 certificates, and installation of private and public keys in the PKDS. In the latter case, only RSA keys can be imported in the PKDS.

Once keys and certificates are made available in the RACF database, specific RACF users are given access to them via keyrings that these users own. The RACDCERT command has options for creating keyrings for specific users and

“connecting” keys and certificates to these keyrings, so that they are available to these users.

The execution of the RACDCERT command is controlled by privileges such as the ability to generate keys and certificates, to create keyrings and connect the keys and certificates to the keyring for oneself or for somebody else. The strategy is that the end user is given access to keys and certificates via a keyring prepared by a security administrator authorized to the RACDCERT command.

In the context of use of the Encryption Facility for z/OS OpenPGP support, the following profiles need to be set up in the RACF FACILITY class of profiles to control access to the RACDCERT command basic functions:

- IRR.DIGTCERT.ADD and IRR.DIGTCERT.GENCERT - For the generation of keys and certificates
- IRR.DIGTCERT.ADD - For the installation of keys and certificates into the RACF database or the PKDS
- IRR.DIGTCERT.ALTER - For the manipulation of the TRUST status granted to certificates in the RACF DB
- IRR.DIGTCERT.EXPORT - To export certificates or key files (in PKCS#7 or PKCS#12 format) out of the RACF database
- IRR.DIGTCERT.ADDRING, IRR.DIGTCERT.DELRING- To create keyrings on behalf of a user or to delete keyrings
- IRR.DIGTCERT.CONNECT and IRR.DIGTCERT.REMOVE - For the connection or removal of keys and certificates to or from the keyring
- IRR.DIGTCERT.LISTRING - To list the contents of a keyring

As for other resources in RACF, the access mode granted with the permission to these profiles, that is READ, UPDATE, and so forth, determines whether the user of the RACDCERT command can issue commands to operate on his or her own resources, or can handle somebody else’s resources.

Further examples of the use of the RACDCERT command are given in Chapter 7, “Session key protection: Public key cryptography” on page 99.

Note: For an application to exploit the keys and certificates that are connected to the keyring, it is not enough for a RACF user ID to own the keyring. This user ID must also be permitted to the IRR.DIGTCERT.LISTRING profile.

Using the ICSF PKDS key management utility ISPF panels

Starting with ICSF FMID HCR7731, that is, the FMID delivered in z/OS V1R8, users can interactively generate RSA keys in the PKDS, import RSA public keys

into the PKDS, and produce X.509 V3 certificates out of public keys in the PKDS. The function is also made available on earlier releases of ICSF with APAR OA15156.

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 then 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 the deletion of keys, the exportation of an asymmetric public key to a self-signed X.509 V3 certificate, and also the ability to import a public key that is contained in an existing X.509 V3 certificate file.

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

Complete details about the ICSF PKDS Key Management enhancement can be found at:

`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, whereas 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.

Keys generated in or added this way to the PKDS need to be “declared” as being keys in the JCECCAKS keystore if they are to be used by the Encryption Facility for z/OS OpenPGP support, as discussed in the next section.

Re-synchronizing a keystore with existing PKDS key tokens

If keys already exist in a key repository that should be transported into a keystore, the usual way one may expect to proceed is to export the keys from their current repository with utilities providing interoperable packages (such as PKCS#7 or PKCS#12), then to import the key package into the keystore.

For the case of keys already existing in the PKDS that should eventually become entries in the JCECCAKS keystore, the Encryption Facility for z/OS OpenPGP support provides the **-prepare** command. The command requires the user to designate the specific key label in the PKDS and the alias it is going to correspond to in the JCECCAKS keystore. The -prepare command builds an object in the JCECCAKS that is the PKDS label of the already existing PKDS key.

Note: There is no need to re-synchronize the JCECCARACFKS to existing keyring contents because access to the key is actually mediated through the RACF keyring itself. The fact that the private key may already be installed in the PKDS is also handled by RACF, which then returns a PKDS key label to the IBMJCECCA provider.

4

Installation

This chapter provides a brief description of the Encryption Facility V1.2 prerequisites and information on its installation.

4.1 Installation

The IBM Encryption Facility for z/OS, program number 5655-P97, is made up of the following features:

- Encryption Facility Encryption Services Feature, FMID HCF7740
- Encryption Facility DFSMSdss Encryption Feature, FMID HCF773D

Each of the features can be ordered separately, or they can be ordered together as a single package. There are no install interdependencies between the features.

The installation follows the RECEIVE, APPLY, and ACCEPT SMP/E tasks. You can install both features at the same time or sequentially depending on your needs. After you receive the product, you can extract from the RELFILE the job that can be used to complete the installation:

- CSDALLOC to allocate the target and distribution libraries.
- CSDDDDEF to define SMP/E DDDEFs.
- CSDISMKD to execute the MKDIR exec for the HFS paths for the IBM Encryption Facility. If you plan to use a new HFS file, remember to update the BPXPRMxx parmlib member to reflect the change across IPLs.
- CSDMKDIR to create the HFS directories.

As a result of the installation, the following libraries are created or updated:

- ASAMPLIB - Distribution library
- ACSDHFS - Source for the HFS file
- ACSDMOD0 - Distribution library
- ALINKLIB - Distribution library
- SAMPLIB - Sample target library
- LINKLIB - Target library
- SCSDHFS - Target HFS file with /usr/lpp/encryptionfacility/IBM/ path

You need also to create the HFS directories /var/encryptionfacility and /etc/encryptionfacility. Copy the ibmef.config file from /usr/lpp/encryptionfacility to /etc/encryptionfacility. This file should be edited in /etc/encryptionfacility.

Note: For more detail on the installation tasks, refer to *Program Directory for IBM Encryption Facility for z/OS V1.2.0*, GI10-0771.

4.2 Prerequisites

The following sections describe hardware and software requirements for the Encryption Facility for z/OS V1.2.

4.2.1 Hardware requirements

The options that you specify for encryption and decryption, including cryptographic keys, depend on the processor type and the cryptographic hardware that you have installed on the system. The Encryption Services and the DFSMSdss Encryption features of the Encryption Facility for z/OS run on the following IBM servers:

- System z9 BC or z9 EC, or equivalent
- zSeries z900 or z990, or equivalent
- zSeries z800 or z890, or equivalent

The cryptographic options for Encryption Facility v1.2 have the following requirements:

- For the PASSWORD option, use one of the following:
  - CPACF only
  - CCF
- For the Clear-TDES and Clear-AES128 (no ENCTDES), use one of the following:
  - CPACF only, or CPACF with PCIXCC/CEX2C
  - CCF, or CCF with PCICC
- For 2048-bit keys, use one of the following:
  - CEX2C
  - PCIXCC
  - PCICC with PCI Crypto 2048 bit Enablement Feature 0867
- For RSA keys generated either through RACF using the RACDCERT GENCERT command with the PCICC parameter, or directly through the ICSF ISPF panels, use one of the following:
  - CEX2C
  - PCIXCC
  - PCICC
- 1024-bit RSA keys in the Modulus-Exponent form, as built by the RACDCERT GENCERT or ADD command with the ICSF parameter.

4.2.2 OpenPGP support and hardware cryptography

For AES or TDES symmetric encryption use one of the following:

- CPACF only
- CPACF with PCIXCC / CEX2C
- CCF
- CCF with PCICC

For signatures or session key encryption using 2048-bit keys or 2048-bit RSA key generation, use one of the following:

- CEX2C
- PCIXCC
- PCICC with PCI Crypto 2048 bit Enablement Feature 0867

For signatures or session key encryption using the 1024-bit RSA keys in the Modulus-Exponent form, as built by the RACDCERT GENCERT or ADD command with the ICSF parameter, a CCF is required.

For signatures or session key encryption using RSA keys generated either through RACF using the RACDCERT GENCERT command with the PCICC parameter, or directly through the ICSF ISPF panels, use one of the following:

- CEX2C
- PCIXCC
- PCICC

4.2.3 Software requirements

This section summarizes the software requirements for Encryption Facility Version 1.2.

The Encryption Services feature supporting the System z format requires:

- z/OS (5694-A01) or z/OS.e (5655-G52) V1.6 or later release
- PTF for z/OS DFSMS™ APAR OA09868 and QSAM APAR OA13571
- Integrated Cryptographic Services Facility (ICSF) Web deliverable (FMID HCR7720) or later
- APAR OA15156 for PKDS Key Management Enhancements

The Encryption Services for OpenPGP requires:

- z/OS (5694-A01) or z/OS.e (5655-G52) V1.6 or higher
- For z/OS (5694-A01) or z/OS.e (5655-G52) V1.6, PTF for z/OS APAR OA10229
- Integrated Cryptographic Services Facility (ICSF) Web deliverable (FMID HCR7720) or later

- IBM 31-bit SDK for z/OS, Java 2 Technology Edition, Version 5, product 5655-N98 at Service Refresh level SDK5 SR4 or later
- PTF for z/OS ICSF APAR OA19177

The Java Client of the Encryption Facility for z/OS Client (optional) requires:

- To run on z/OS, one of the following:
  - 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)
- To run on other platforms, one of the following:
  - Sun SDK 5.0
  - An IBM JVM at SDK1.4.2
  - A JVM with a JCE cryptographic provider installed that supports all the required algorithms

Decryption Client for z/OS requires:

- z/OS (5694-A01) or z/OS.e (5655-G52) V1.4 or higher. (Note: The Decryption Client for z/OS only runs on z/OS and is supported with both Encryption Facility for z/OS, V1.1 and V1.2.)
- PTF for z/OS DFSMS APAR OA09868.
- z/OS Cryptographic Services - Integrated Cryptographic Services Facility with z990 Cryptographic Support Web deliverable (FMID HCR770A) or later. Some hardware features require the z990 and z890 Enhancements to Cryptographic Support Web deliverable (FMID HCR770B) or later. Decryption Client for z/OS runs only on the z/OS platform and only provides decryption services.

The DFSMSdss Encryption feature of Encryption Facility for z/OS requires:

- z/OS (5694-A01) or z/OS.e (5655-G52) V1.6 or higher
- Integrated Cryptographic Services Facility (ICSF) Web deliverable (FMID HCR7720) or later
- Either the DFSMShsm/DFSMSdss combination priced feature or the DFSMSdss priced feature of z/OS or z/OS.e V1.6 or higher
- PTF for z/OS DFSMS APARs OA13300, OA13453, and OA13687

4.2.4 Configuration setup

The Encryption Facility for OpenPGP functions are executed as Java commands that use a configuration file to contain the options for the command run. For details on where to locate the file and its content, refer to 5.6.4, “Options in the configuration file”.

4.2.5 Exploitation of the z/OS integrated cryptography

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.

You could potentially update the crypto provider in the security properties to make use of the hardware cryptography.

Table 4-1 summarizes the exploitation of the system cryptographic devices by the Encryption Facility for z/OS. The following sections provide detailed information about these devices and which cryptographic services are called.

Table 4-1 Summary of hardware cryptographic functions used by the Encryption Facility for z/OS

System	Hardware crypto-graphic facility	Data Encrypt/ Decrypt with triple DES	Symmetric Key Encrypt/ Decrypt with RSA key up to	RSA key generation up to	RSA signing key up to	Random number generation
z890 z990 z9	CPACF	Use clear keys	N/A	N/A	N/A	N/A
	CEX2C	Use secure keys	2048 bits	2048	2048	YES
	PCIXCC	Use secure keys	2048 bits	2048	2048	YES
z800 z900	CCF	Use secure keys	1024 bits	N/A	1024	YES
	PCICC	N/A	2048 bits	2048	2048	N/A

The hardware cryptographic facilities in z9, z990, and z890

In this section, we briefly 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. While this method will increase the path length, it provides the potential advantage of exploiting some ICSF built-in capabilities that might be relevant to the specific user 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 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 or SHA-256 algorithm.

Note: Because the Encryption Facility for OpenPGP requires the symmetric algorithms to be run in CFB (Cipher Feedback) chaining mode, which is implemented via a change to the ICSF software, the CPACF is always called via ICSF for the OpenPGP encryption or decryption.

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 Encryption Facility for OpenPGP does not use secure keys and therefore does not use the CEX2C for T-DES encryption.
- 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 somewhat 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 predecessor 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 enable the cryptographic devices on the system.

- 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 "cards." Beware of this somewhat 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.

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.

Note: For more details on the ICSF services used by the Encryption Facility for z/OS, refer to the *Encryption Facility for z/OS Version 1.10*, SG24-7318.

5

Using the Encryption Facility for OpenPGP

This chapter provides a high-level description of how the OpenPGP support has been implemented in the Encryption Facility for z/OS V1.2 program product, along with discussion and examples on the use of the OpenPGP commands.

5.1 Encryption Facility for z/OS support of OpenPGP

OpenPGP as supported by the Encryption Facility for z/OS provides security and integrity services for data files or datasets exchanged between z/OS and partner z/OS or non-z/OS systems. The security and integrity services are based on the data format and protocol described in the Internet Standard RFC 2440.

By additionally using IBM mainframe cryptographic functions, the OpenPGP support gets the benefits of improved performance and centralized key management provided by the coprocessors, ICSF and RACF.

In order to establish the trust required to achieve secure exchanges of data between OpenPGP users, the Encryption Facility for OpenPGP makes use of secret passphrases or pairs of private and public keys.

OpenPGP requires the two parties to get the same shared secret key, which is used to encrypt and decrypt the data. The OpenPGP Internet draft RFC 2440 describes a format that encrypts the shared secret key and then imbeds the now protected secret key in the OpenPGP message element. Either Passphrase-based encryption (PBE) or public key-based encryption could be chosen to protect the secret key. Whenever public keys are needed, the Encryption Facility for OpenPGP supports exchange of OpenPGP certificates with other OpenPGP compliant systems. The OpenPGP support in the Encryption Facility for z/OS internally uses X.509 certificates and keys and therefore exploits the z/OS built-in facilities to manage the X.509 entities.

The OpenPGP support in the Encryption Facility for z/OS includes, but is not limited to, the following items, as they are described in RFC2440:

- Passphrase-based enciphering of session key
- Digital signatures of data
- Import or export of OpenPGP certificates, with proper support for Versions 3 and 4 of OpenPGP signatures
- RSA, ElGamal, and DSA key generation
- ASCII Armor for OpenPGP certificates
- Data encryption with a randomly generated symmetric session key using AES 128, 192, or 256 bit key, triple DES, or Blowfish algorithms
- Symmetric encryption (PBE) of randomly generated symmetric session key using AES 128, 192, or 256 bit key, triple DES, or Blowfish algorithms
- Asymmetric encryption of randomly generated symmetric keys using the RSA or ElGamal algorithms
- Compression using ZIP or ZLIB algorithm libraries; digest and hash using SHA-1, MD5, MD2, SHA-256, SHA-384, or SHA-512 algorithms
- Digital signature using DSA with SHA1 or RSA (with all supported hashes listed previously) algorithms

- Optional MDC (Modification Detection Code) generation and verification for the exchanged data

5.2 Overview of the OpenPGP support implementation

Figure 5-1 is a conceptual view of the Encryption Facility for OpenPGP functional blocks.

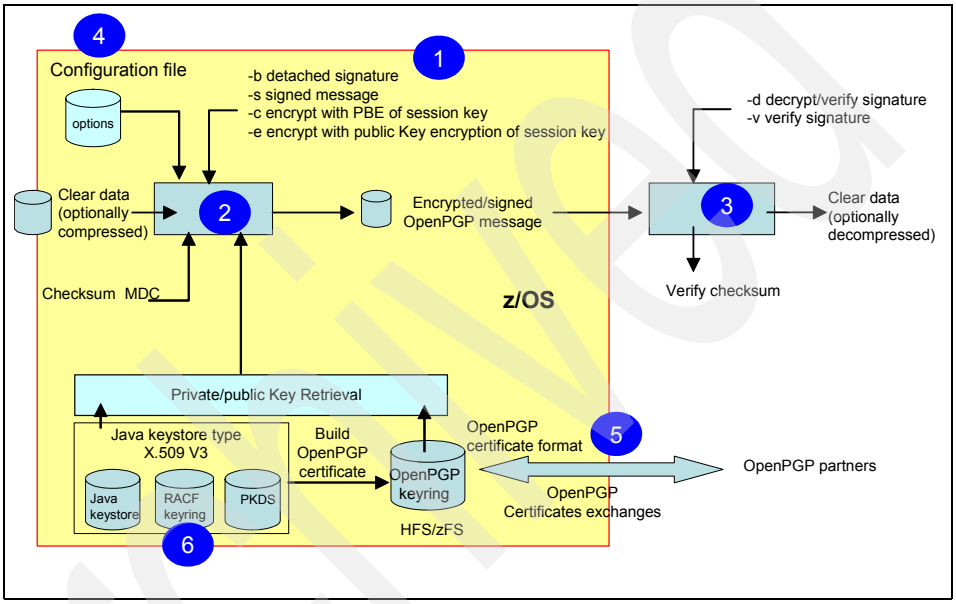


Figure 5-1 Encryption Facility for z/OS - OpenPGP support flow

1. Users issue commands to invoke the security and integrity functions provided by the OpenPGP support.
2. Data is fetched from input files or datasets, transformed and formatted according to RCF 2440, and made available in its new form in output files or datasets.
3. The output files or datasets are transmitted to the OpenPGP-compliant partner, who then proceeds with the de-formatting and reverse transformation of the data. Note that the Encryption Facility for z/OS does not provide any facility for the data transmission to the partner application.
4. The Encryption Facility for OpenPGP support proposes an optional configuration file where users can keep their own customized set of OpenPGP commands options, and use it as input to the OpenPGP functions

as an alternative to typing the options with the commands. The configuration file is further described at 5.6.4, “Options in the configuration file” on page 77.

5. OpenPGP public keys are exchanged between partners using OpenPGP certificates. The Encryption Facility for OpenPGP supports receiving OpenPGP certificates from partners and sending OpenPGP certificates to partners.
6. Internally, the OpenPGP support uses the z/OS built-in X.509 certificates management facilities and therefore operates in a hybrid environment, supporting both X.509 and OpenPGP certificates and keys. This hybrid environment is further explained at 5.5, “Encryption Facility for OpenPGP and digital certificates” on page 70.

5.2.1 Partner’s preferred algorithms

The Encryption Facility for OpenPGP supports using the algorithms preferences, if any, as stated in a data recipient certificate, according to RFC 2440. The recipient’s preferences can be overridden by specifying the algorithms to use as command options.

5.2.2 Sending OpenPGP protected data to multiple partners

The Encryption Facility for OpenPGP supports sending the same message to multiple partners. It requires having the partners’ certificates in the OpenPGP keyring, or keystores, and specifying multiple recipients in the command. Whatever processing is done for one recipient is repeated for every specified recipient, still preserving the secrecy of the data. For instance, a set of encrypted data prepared for multiple recipients with public key protection of the session key will also contain as many copies of the session key as there are recipients, each copy being encrypted with an individual recipient’s public key.

5.3 Algorithms

This section provides information on which algorithms are selectable by the users of the Encryption Facility for OpenPGP, and whether hardware assist can be provided by the system. Note, however, that the cryptographic hardware assist is mediated through the proper JCE services providers, which in turn call ICSF whenever appropriate. It also assumes that the supporting hardware is installed on the system. Refer to 4.2.2, “OpenPGP support and hardware cryptography” on page 56 for a description of the required hardware cryptographic coprocessors.

To get a list of the possible encryption algorithms you can use the OpenPGP command **-list-algo**.

5.3.1 Symmetric algorithms

Triple DES (TDES)

The TDES algorithm is supported by hardware.

AES-128

AES 128 has hardware support starting with System z9. Otherwise, ICSF provides support via its own internal software, which does not call for any hardware facility.

AES-192

ICSF provides support via its own internal software, which does not call for any hardware facility.

AES-256

ICSF provides support via its own internal code, which does not call for any hardware facility.

Blowfish

The Blowfish algorithm operates with a 128-bit key. It does not have any hardware support.

Important: The Encryption Facility for OpenPGP only supports symmetric algorithms with the CFB (Cipher Feedback) mode of operation. The fix for APAR OA19177 must be installed to have ICSF support for this mode.

5.3.2 Asymmetric algorithms

RSA

The RSA computations can be assisted by hardware; however, in that case the maximum key length is 2048 bits. If the user requires a longer key, the RSA computation will be performed by software only.

ElGamal

ElGamal is not a hardware supported algorithm and will be performed by the JCE software provider.

5.3.3 Compression algorithms

ZIP

The ZIP compression process is not assisted by hardware; it is performed by a pure Java ZIP implementation in the SDK.

ZLIB

The ZLIB compression process is not assisted by hardware; it is performed by a pure Java ZLIB implementation in the SDK.

5.3.4 Digital signature algorithms

DSA signature of a SHA-1 hash

There is no hardware support for the DSA algorithm in the z990, z890, and follow-on systems. However, SHA-1 is performed with the CPACF hardware on those systems.

RSA signature of a SHA-1 hash

There is hardware support both for the RSA algorithm (to the extent of a 2048-bit long key, maximum) and SHA-1.

RSA signature of a SHA-256 hash

This is implemented in software only as of the writing of this book.

RSA signature of a SHA-384 hash

This is implemented in software only as of the writing of this book.

RSA signature of a SHA-512 hash

This is implemented in software only as of the writing of this book.

RSA signature of an MD2 value

There is hardware support for the RSA algorithm; MD2 hashing is always performed by software in the IBMJCECCA provider.

RSA signature of an MD5 value

There is hardware support for the RSA algorithm; ICSF provides a software implementation of MD5 hashing.

5.3.5 Message Digest algorithms

MD2

This algorithm is performed by software. The IBMJCECCA hardware provider has its own software implementation of the algorithm.

MD5

Although ICSF is called to perform this algorithm, it is actually performed by the ICSF software without using hardware facilities.

SHA-1

This algorithm is supported by hardware.

SHA-256

This algorithm is supported by hardware starting with the System z9.

SHA-384

This algorithm is not supported by hardware.

SHA-512

This algorithm is not supported by hardware.

5.4 Encryption Facility for OpenPGP performance considerations

There are several options to optimize the performance of the OpenPGP support in order to meet users' requirements. Obviously one of these options is the use of hardware cryptography as a means of both improving the cryptographic throughput of the application and saving on CPU cycles. The Encryption Facility for OpenPGP also has a built-in capability of establishing parallel threads of functions, so that the user can select to have the following tasks running on their own thread, in parallel with other functions performed in the OpenPGP code:

- I/O operations
- Compression of data
- Encryption and decryption of data

This capability is further explained in Chapter 10, "Performance" on page 213.

5.5 Encryption Facility for OpenPGP and digital certificates

As already mentioned, the OpenPGP support uses a hybrid environment with X.509 and OpenPGP digital certificates. OpenPGP certificates are kept in a z/OS UNIX file that acts as an OpenPGP keyring, while X.509 private and public keys (and their certificate, if relevant) are kept in Java keystores. A Java keystore can be the z/OS RACF database (for RSA and DSA keys and certificates), or the ICSF PKDS VSAM dataset (for RSA keys only).

5.5.1 Implementation

Figure 5-2 is a representation of the hybrid environment where the Encryption Facility for OpenPGP can use both X.509 and OpenPGP keys and certificates.

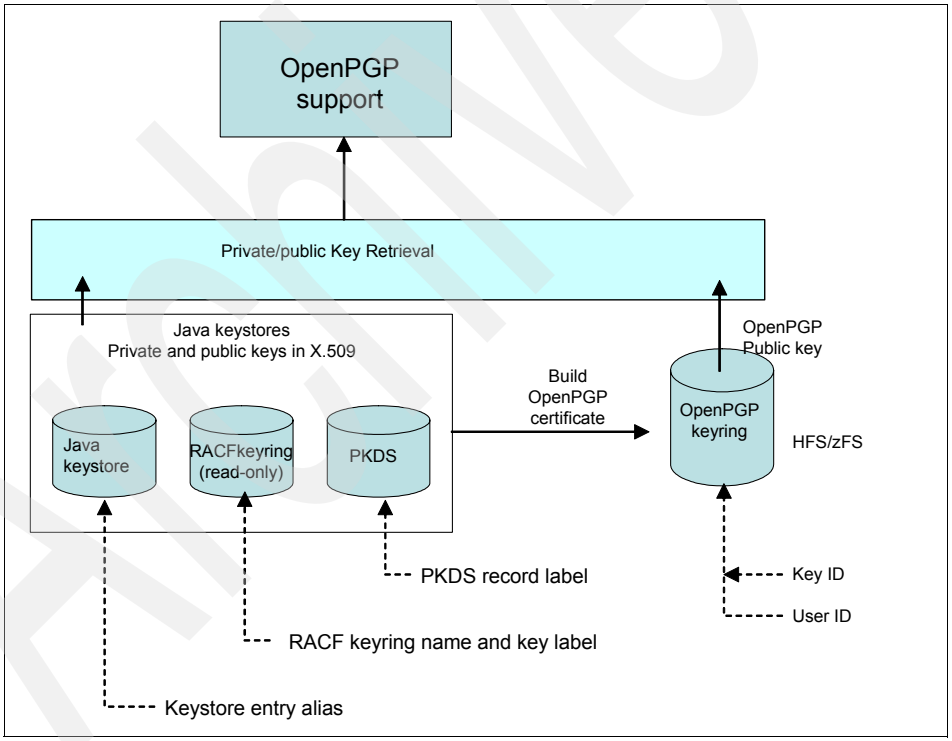


Figure 5-2 Encryption Facility for OpenPGP keys and certificates environments

X.509 keys and certificates are provided through a keystore implementation that is accessible to the Java code of the OpenPGP support. The keystore type can be:

- JKS or JCEKS, for keystores physically residing in z/OS UNIX files.
  Keys are kept in keystore entries designated with a key *alias*.
- JCECCAKS, for keys residing in the ICSF PKDS.
  Keys are kept in PKDS "tokens" that are designated by a PKDS label. Note that the JCECCAKS is accessed with key alias that is internally translated into a PKDS label.
- JCERACFKS, for keys and certificates residing in RACF keyrings.
  Keys and certificates in the RACF database are aggregated onto RACF keyrings that users own. This special type of keystore should be accessed with the RACF user ID and the RACF keyring name to retrieve keys and certificates in the RACF keyring. Note that the JCERACFKS is a read-only keystore.
- JCECCARACFKS, for private keys residing in the ICSF PKDS and certificates in the RACF keyrings.
  This keystore externally operates like the JCERACFKS by being provided with a RACF user ID and a RACF keyring name. The retrieval of private keys that reside in the PKDS is transparently done by the keystore.
  Note that the JCECCARACFKS is a read-only keystore.
- OpenPGP certificates, that is public keys only, are kept in a z/OS UNIX file that implements the OpenPGP keyring concept (see 2.5.3, "The OpenPGP keyring" on page 20). The corresponding private key portion of the OpenPGP certificate, when it exists, is actually kept in one of the keystores described previously.

  An openPGP certificate is looked for in an OpenPGP keyring with either one of these attributes:

  – A key ID, which is an 8-byte value derived from the hash of the key value. RFC 2440 provides details on how a Key ID is computed.

    Note: Because of its computation method, OpenPGP does not guaranty the uniqueness of a Key ID.

  – A user ID, which is specified when the certificate is generated or imported into the keyring. The OpenPGP user ID is made up of three parts:
    - A user real name
    - An e-mail address
    - A comment

    Note that the Encryption Facility implementation of the OpenPGP keyring uses two additional z/OS UNIX files to hold indexes to the keys in the

keyring. One file holds an index by User ID, the other file an index by Key ID. If for some reason these index files are damaged they can be rebuilt using the OpenPGP **-rebuild-key-index** command.

Note: The Encryption Facility for OpenPGP also supports retrieving keys in a Java keystore using an OpenPGP key ID. In that case the OpenPGP software scans the keystore and builds an OpenPGP key ID for all scanned keys until a matching key ID is found.

5.5.2 Certificate management

The Encryption Facility for OpenPGP users have to use different certificate management utilities depending on which repository the certificate is in.

Note: Details about the steps for exchanging OpenPGP certificates, X.509 certificates, and public/private key pairs, using the OpenPGP support, and IBM Java 5 SDK tools keytool and hwkeytool are in Appendix C, "OpenPGP key exchange and migration" on page 253.

Certificate management utilities for Java keystores

This section summarizes the material provided in 3.2.6, "Managing keys in the keystores" on page 48.

- For JKS or JCEKS types of keystores, the Java utility "keytool" is used to manage keys and certificates. Information on keytool can be found at:
  `http://www.ibm.com/developerworks/java/jdk/security/142/secguides/keytoolDocs/KeyToolUserGuide-142.html`
- For the JCECCAKS type of keystores, the Java utility "hwkeytool" is used to manage keys and certificates. Information on hwkeytool can be found at:
  `http://www.ibm.com/servers/eserver/zseries/software/java/hwkeytool.html`
- For the JCERACFKS and JCECCARACFKS types of keystore, the RACF command RACDCERT is used to manage keys and certificates.

Note: Keys in the ICSF PKDS can be managed using the RACF RACDCERT command or the ICSF ISPF panel for PKDS Key Management (available with ICSF HCR7731, or with ICSF that has received the fix for APAR OA13030). Java keystores may then need to be synchronized to the PKDS contents using the **-prepare** OpenPGP command.

Certificate management commands for the OpenPGP keyring

The OpenPGP keyring content is affected by OpenPGP commands that generate or delete keys, import or export certificates, as illustrated in Figure 5-3:

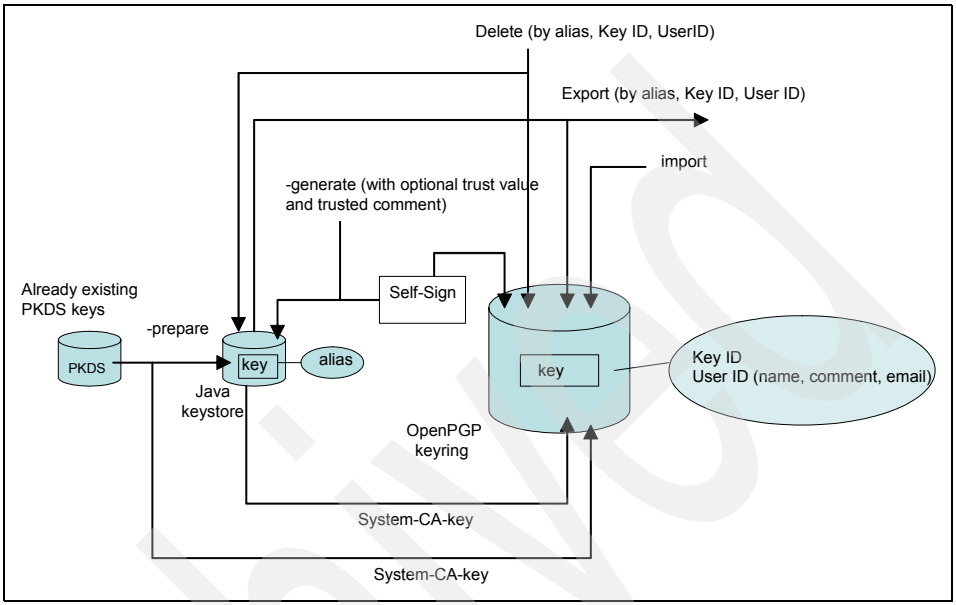


Figure 5-3 Certificate management in the OpenPGP keyring

- On the left is shown the capability of synchronizing Java keystores with the already existing contents of the PKDS using the **-prepare** command.
- Public keys can be inserted in the OpenPGP keyring as a result of the execution of the following commands:
  - The **-i** import command.
  - The **-g** command (generate). Note that this command generates a key pair and results in the following:
    - The private key and X.509 certificates are kept in a Java keystore. The certificate is signed by the key designated as the system-CA-key, if any, in the command options, or self-signed if there is no system-CA-key specified. A description of the system-CA-key-alias option is in Appendix B, “Configuration file options” on page 237.
    - An OpenPGP certificate is built around the public key, is self-signed, and is signed by the system-CA-key, if any. It is then stored in the OpenPGP keyring.
  - The **-eA** export by alias command. The export process involves fetching the public key from the designated Java keystore entry, and building an OpenPGP certificate for this key that is self-signed and signed by the

system-CA-key, if any. The process involves prompting the user for the certificate owner's identity information. The OpenPGP certificate is then made available both in the designated output file and in the OpenPGP keyring.

  - The **-prepare** command, where at the same time an entry is created in a JCECCAKS keystore, an OpenPGP certificate is built and stored in the OpenPGP keyring. An example of use of the **-prepare** command is given in 7.3.2, "Using key pairs in the ICSF PKDS" on page 114.

- Public keys can be removed from the OpenPGP keyring as a result of:
  - The deletion of a certificate by keystore alias (**-xA**) if the public key happens to already be in the OpenPGP ring
  - The deletion of a certificate by Key ID (**-xK**) or User ID (**-xP**)

Certificates can be exported from the OpenPGP keyring, that is, put in an output file intended to be shipped, by Key ID (**-eK**) or user ID (**-eP**).

Certificate verification at import

The Encryption Facility for OpenPGP attempts to verify any signature in the certificate using public keys available in the OpenPGP keyring.

It prompts the user for any special condition that is encountered, such as an expired certificate or revoked certifications.

It always asks for user confirmation before storing the certificate in the OpenPGP keyring.

You can use the option **-yes/-no** to allow for pre-answer of the prompts.

5.5.3 Generation of OpenPGP certificates - Subkeys

As already mentioned in 2.5.2, "The OpenPGP certificate" on page 18, the primary key in the OpenPGP certificate must be a signing key. The Encryption Facility for OpenPGP generates certificates only for primary keys that have the signing capability, such as DSA keys or RSA keys flagged as signature keys. When invoking the **-g** command, the user is asked what type of keys he or she wants to be generated. If the key type is ElGamal, which is an encryption-only algorithm, then the Encryption Facility for OpenPGP generates a new primary key pair (RSA or DSA) and certificate along with an ElGamal key pair, and the new ElGamal public key is added as a subkey into the certificate. This process is further discussed in 7.3.3, "ElGamal keys" on page 127.

5.6 Invocation of Encryption Facility for OpenPGP

The Encryption Facility for OpenPGP can be invoked in two ways: via a command in the z/OS UNIX shell or in a batch environment using the IBM JZOS Batch Toolkit for z/OS.

5.6.1 Encryption Facility for OpenPGP commands

OpenPGP commands are Java-based commands with options to sign messages, encrypt or decrypt messages, verify OpenPGP messages and signatures, and control or manage keys and certificates. The command is made up of three elements:

- The command verb itself, always beginning with a "**-**"
- One or more command options, also beginning with a "**-**"
- Arguments, which are plain parameter values passed in the command line

5.6.2 Encryption Facility for OpenPGP commands overview

The following list identifies the commands (in alphabetical order for simplicity) and their possible arguments. It *does not* show the possible options, which are discussed in 5.6.3, "Encryption Facility for OpenPGP commands options". Note that some commands allow the user to prepare messages for multiple recipients because several key aliases, key IDs, or user IDs can be specified for command arguments.

- -b file
  Generates a signature for OpenPGP message and yields a signature-only file (the assumption being that data is transferred out of band and partners intend to exchange the corresponding signature only).
- -c file
  Encrypts the contents of the OpenPGP message using PBE.
- -d file [file ...]
  Decrypts one or more encrypted OpenPGP messages. If the data are signed as well as encrypted, -d verifies the signature after data decryption.
- -e file
  Use public key encryption with the specified recipients; if they are invalid, PBE is defaulted to.
- -eA alias [alias ...]
  Interactively export an OpenPGP certificate.
- -eK key_ID [key_ID ...]
  Exports OpenPGP certificates using key IDs.

- -eP user_ID [user_ID ...]
  Exports OpenPGP certificates by user IDs.
- -g
  Interactively generates a system key pair used for signatures with encryption-only subkeys if required.
- -h
  Prints the Help menu to STDOUT, as mentioned previously.
- -i file [file ...]
  Imports an OpenPGP certificate into the OpenPGP keyring file.
- -list-algo
  Prints a list of algorithms to STDOUT.
- -pA [alias ...]
  Lists information about all public keys in the keystore or those public keys identified by the alias argument.
- -pK [key_ID ...]
  Lists information about all public keys in the keystore or keyring or public keys identified by the key ID argument.
- -pP [user_ID ...]
  Gives information on all public keys in the keyring file or to those keys filtered by the user_ID argument.
- -prepare key_label [key_label ...]
  Readies keystore to use existing keys in ICSF.
- -s file
  Signs the contents of the OpenPGP message using the key defined in the signers_key_alias option or as entered in the configuration file. Since encryption is not done by default, you need to specify `-c/-e` together with `-s` to sign and encrypt the message.
- -v[detached_signature] signed-file
  Verifies a signed OpenPGP message.
- -xA alias [alias ...]
  Deletes certificate and keys stored in the keystore by alias.
- -xK key_ID [key_ID ...]
  Deletes OpenPGP certificates and keys by key ID from the keystore and OpenPGP keyring.
- -xP user_ID [user_ID ...]
  Deletes OpenPGP certificates and keys by user ID from the OpenPGP keyring.

5.6.3 Encryption Facility for OpenPGP commands options

As mentioned previously, options can be entered in the command line; however, the Encryption Facility for OpenPGP offers as an alternative the use of a *configuration file* whose location is specified by the user. There might be several configuration files prepared for different usage environments and conditions; the user specifies the path to the configuration file he or she wants to use when invoking the OpenPGP command.

From a usage point of view, it is convenient for each user to build his or her own configuration file with the most commonly used configuration options for his or her needs, especially if the user is repeatedly executing the same commands. It is also recommended to have a configuration file for batch processing to create a consistent environment when invoking the OpenPGP commands in batch jobs.

The path to the configuration file is indicated in the `-homedir` command line option; the name of the file must be `ibmef.config`. If the -homedir option is not specified, the Encryption Facility for OpenPGP looks for the default configuration file at /etc/encryptionfacility/ibmef.config.

A sample configuration file, populated with all the options commented out, is available in /usr/lpp/encryptionfacility/ibmef.config.

5.6.4 Options in the configuration file

The command options as specified in the configuration file, along with a reference to the equivalent options as specified in the command line, are described in Appendix B, “Configuration file options” on page 237. The options described in the appendix are those available at the time of writing. For more current descriptions, refer to *IBM Encryption Facility for z/OS: Using Encryption Facility for OpenPGP*, SA23-2230.

5.6.5 z/OS UNIX shell commands and scripts

The command syntax to invoke Encryption Facility for OpenPGP from the z/OS UNIX shell is:

```
java -jar <path to Encryption Facility for OpenPGP jar>[-homedir name]
[command options] commands [arguments]
```

The .jar file located at /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar contains a manifest, which points to the main class of the program. Therefore, a command entered in the z/OS UNIX shell beginning with:

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar
```

is used to invoke the Encryption Facility for OpenPGP. If for some reason the main class has to be specified, it is located at:

```
com.ibm.encryptfacility.EFOpenPGP
```

You must also properly initialize the CLASSPATH and LIBPATH environment variables that you are using when in the z/OS UNIX shell. The following setup is recommended in the profile file in use for your shell session:

```
export JAVA_HOME=<JAVA_HOME> export PATH=/bin:"${JAVA_HOME}"/bin:
LIBPATH=/usr/lib/java_runtime:/lib:/usr/lib:"${JAVA_HOME}"/bin
LIBPATH="$LIBPATH":"${JAVA_HOME}"/bin/classic
export LIBPATH="$LIBPATH":
CLASSPATH=/usr/include/java_classes/ifaedjreg.jar
CLASSPATH=$CLASSPATH:/usr/lpp/encryptionfacility/CSDEncryptionFacility.
jar
```

Since the Encryption Facility for OpenPGP command can be rather lengthy, it is recommended to proceed with shell scripts invocation. A shell script is a text file that contains a command or a sequence of commands for a UNIX-based operating system and can be executed similarly to a REXX™ EXEC or a CLIST in the z/OS environment.

Shell script basics

A shell script is a text file. Use Oedit (or your favorite z/OS UNIX installed text editor) to create and edit your script. The backslash character (\) is used as a continuation character, and it usually makes the command more understandable when it spans several screen lines. See Figure 5-4 for the contents of a shell script.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/sjon/  \
-keystore /u/sjon/keystore/jceccaks_keystore \
-keystore-password stanjon   \
-keystore-type JCECCAKS \
-xP stan
```

Figure 5-4 Sample of an Encryption Facility for z/OS shell script called xp.sh

In order to execute the script file, the execute permission bit has to be set ON. Issue a UNIX list (**ls -l**) command to see the file permissions, as shown in Figure 5-5.

```
SJON:/u/sjon/openpgp: >ls -l xp.sh
-rwx------   1 SJON     SYS1         227 Mar 20 12:45 xp.sh
```

Figure 5-5 Output from list command

The script file has the owner's execute permission bit set (that is, the x in the "-rwx" sequence). If it does not, use the command **chmod 700 *filename*** to set the execute bit (for example, **chmod 700 xp.sh**).

To execute the script, just type the script's full filename (including its extension). Depending on how your z/OS UNIX user profile is set up, you may need to prefix the shell script name with a period and a forward slash (./) to invoke its execution, (for example, **./xp.sh**). You may want to ask your z/OS UNIX system programmer to modify your profile.

5.6.6 JZOS and batch environment

IBM has provided three members in SYS1.SAMPLIB for users who want to invoke Encryption Facility for z/OS in a batch environment. The three files are:

- CSDJZSVM - The started procedure in PROCLIB.
- CSDSMPEN - The shell script to use to configure environment variables.
- CSDSMJCL - The batch job JCL that calls the sample stored procedure.

All three files need to be modified according to their actual execution environment. CSDSMJCL contains three sample commands (JAVA1, JAVA2, and JAVA3) to call the Encryption Facility for OpenPGP.

Many of the Encryption Facility for OpenPGP commands require user input, such as password. Since the Encryption Facility for OpenPGP is called in a batch environment, you must specify the command options and the passwords (such as -keystore password or -key-password) in the JCL. Because doing so does not meet strict security requirements, you may want to specify the respective command options in the configuration file ibmef.config as opposed to letting them be visible in the JCL cards.

Certain commands that require user input and that have no equivalent command option cannot be issued in batch. One such command is the generation of a key pair and certificate(-g). Review the key generate scenario in 7.3.2, "Using key pairs in the ICSF PKDS" on page 114 to see the user input required for this command.

5.6.7 Encrypting and decrypting z/OS data sets

Encryption Facility for OpenPGP reads and writes to only three types of z/OS data sets. The three types are:

- Sequential
- PDS member
- PDSE member

Any other data set format (such as VSAM) cannot by used as input or output for Encryption Facility for z/OS.

All output data sets used by Encryption Facility for OpenPGP must be pre-allocated. The output data set for an encryption or signature command cannot be a fixed block (RECFM=FB) data set. Use a variable block data set for output. For a decryption command, pre-allocate a data set with the same DCB attributes as the source data set.

Encryption Facility for OpenPGP does not retain data set information in the encrypted or signed binary data. Thus, the decrypted data might not have the same attributes as the source data. The chances of getting decrypted data in the same format as the source data increase if the pre-allocated output data set has the same DCB attributes as the source.

Encryption Facility for OpenPGP accepts a fully qualified data set name or a DD name. In order for Encryption Facility for OpenPGP to reference z/OS data sets, the data set name or the DD name must be prefixed with two forward slashes (//). When referencing a PDS or PDSE data set in a shell script, you must enclose the data set name and the two forward slashes in single quotation marks. If it is a sequential data set, the single quotation marks are not needed. When invoking Encryption Facility for OpenPGP in batch, no quotation marks are needed for any file type. See Figure 5-6 for an example of how to reference a PDS member, and Figure 5-7 for an example of referencing a sequential data set.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /etc/encryptionfacility/   \
-o '//sjon.ef2.fb(dec)'  \
-d '//sjon.ef2.vb(enc)'
```

Figure 5-6 Referencing a PDS member in a shell script

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /etc/encryptionfacility/   \
-cipher-name AES_128  \
-digest-name SHA_1  \
-o //SJON.EF2.ENCRYPT.TEXT  \
-s2k-cipher-name TRIPLE_DES   \
-s2k-digest-name SHA256 \
-s2k-mode 3  \
-c //SJON.PLAIN.TEXT
```

Figure 5-7 Referencing a sequential data set in a shell script

Figure 5-8 is an example of referencing a PDS using the JZOS JCL, and Figure 5-9 is an example of referencing a sequential data set using the JZOS JCL.

```
//PROCLIB JCLLIB ORDER=SJON.EF2.LIB
//ENCPBE  EXEC PROC=CSDJZSVM,VERSION='50'
//STDENV  DD DSN=SJON.EF2.LIB(CSDSMPEN),DISP=SHR
//*
//MAINARGS  DD *
-homedir /etc/encryptionfacility/
-cipher-name triple_des
-o //SJON.EF2.VB(ENCRYPT)
-s2k-passphrase GOOd_bye_HOres
-c //SJON.EF2.LIB(PLAIN)
/*
```

Figure 5-8 Referencing a PDS member with JZOS

```
//PROCLIB JCLLIB ORDER=SJON.EF2.LIB
//ENCPBE  EXEC PROC=CSDJZSVM,VERSION='50'
//STDENV  DD DSN=SJON.EF2.LIB(CSDSMPEN),DISP=SHR
//*
//MAINARGS  DD *
-homedir /etc/encryptionfacility/
-o //SJON.EF2.ENCRYPT.TEXT
-s2k-passphrase GOOd_bye_HOres
-c //SJON.PLAIN.TEXT
/*
```

Figure 5-9 Referencing a sequential data with JZOS

Finally, Figure 5-10 illustrates the use of a DD NAME when invoking the Encryption Facility for OpenPGP.

```
//PROCLIB JCLLIB ORDER=SJON.EF2.LIB
//JAVA2  EXEC PROC=CSDJZSVM,VERSION='50'
//STDENV DD DSN=SJON.EF2.LIB(CSDSMPEN),DISP=SHR
//DECOUT DD DSN=SJON.EF2.DEC.OUT,
//        DISP=(NEW,CATLG),
//        DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160),
//        UNIT=SYSALLDA,
//        SPACE=(CYL,(5,1),RLSE)
//*
//MAINARGS  DD *
-homedir /etc/encryptionfacility/
-o //DD:DECOUT
-s2k-passphrase GOOd_bye_HOres
-d //SJON.EF2.ENC.OUT
/*
```

Figure 5-10 Referencing a data set using a DD name

5.7 Interoperability issues

This section discusses a few issues to consider when you exchange messages within different platforms.

5.7.1 Code page compatibility

Because the Encryption Facility for OpenPGP is intended to run as a z/OS application, it operates with the EBCDIC encoding for text characters. Special care must be taken when exchanging text data with other, non-EBCDIC platforms because improper character set conversion may result in unrecoverable protected data.

The Encryption Facility for z/OS supports translating character sets as described in Appendix B, “Configuration file options” on page 237.

A list of supported character sets can be obtained with the **-list-algo** command.

5.7.2 Messages empty lines

Empty records may be required in a message, such as for an empty line of text. If a z/OS data set is used to hold the message, the Encryption Facility for OpenPGP writes a record of one space because datasets cannot have empty records. Note, however, that some applications that read the message may associate a specific meaning to this space character (for example, a space in an LDIF file is interpreted as a continuation character).

If a space character is not acceptable as a substitute to an empty line, then the user should use z/OS UNIX files to hold the output message.

6

Session key protection: Passphrase-based encryption

This chapter provides examples showing how to use passphrase-based encryption (PBE) to protect the OpenPGP session key.

It begins with a short discussion of why you may want to use PBE or public key cryptography to protect the OpenPGP session key. It then provides examples of OpenPGP data exchanges, using PBE, between z/OS systems and between z/OS and non-z/OS systems.

6.1 Choosing PBE or public key cryptography

General considerations

The choice between the use of PBE and public key cryptography is mostly dictated by these three factors:

- What session key protection mechanism does the partner support?
- What is the level of end-to-end security of the exchange channel?
- What fits best considering the intended or actual usage of the Encryption Facility for OpenPGP and the answers to the first two questions.

Note: The Encryption Facility for OpenPGP does not impose any limit on the length of the passphrase. A limit can, however, be inherent in the passphrase delivery method (JCL, z/OS UNIX shell, and so forth) or with the partner's product that implements OpenPGP.

What does the partner support?

It is expected that most partners will support PBE, and there should not be any major reason not to be technically able to support public key cryptography as well. Instead, this choice would depend on organizational and management reasons such as, for instance, how much does a partner want to be involved in the asymmetric keys and Public Key Infrastructure technologies management.

On the z/OS platform the security of the RSA private keys can be greatly enhanced by keeping the keys in the PKDS and controlling their use with RACF profiles; this might be a differentiator leading mainframe users to prefer using public key cryptography.

End-to-end security of the exchange channel

The use of PBE requires that great care be taken to minimize the risk of unintended passphrase disclosure when stored, exchanged, or used during invocation of the Encryption Facility for OpenPGP or the partner's product. In the Encryption Facility for OpenPGP, this risk can be mitigated by specifying the passphrase in the user's specific configuration file only.

There is also necessary to use a very secure channel to carry over to the partner the passphrase to be shared. It is up to the involved parties to determine how secure the exchange process and media are. Most of the time the passphrase exchange requires a secure channel of its own.

On the other hand, public key encryption is inherently more secure than PBE because there is no secret ever transmitted. Private keys are notably very long

binary values with a number of possible combinations that exceeds by far the PBE's one. However, it is of utmost importance to ensure that proper measures are in place to guarantee the secrecy of the private keys, at both sides of the communication, meaning an objectively known and homogenous level of key security is achieved by both partners.

z/OS-to-z/OS partners can consider exploiting the high level of private key security provided by the use of the ICSF PKDS and RACF access control. On the other hand, z/OS-to-non-z/OS partners might have to lower their expectation regarding the private key overall protection level because protection on the non-z/OS side might not match what is achieved in z/OS.

Best fit for planned usage

Among the situations for which you can consider using Encryption Facility for OpenPGP are the following:

- Ad-hoc file transfer
  If a file is being transferred to another site and there is a need to protect the data en route, then a password protection scheme may provide adequate protection. This would apply particularly if the same person was using the data at the encryption site as at the decryption site.

  Making use of public key cryptography in an ad-hoc situation may be appropriate, but consideration should always be given to 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 session key.
- Regular file transfer
  Transferring files on a regular basis is probably best handled from a security standpoint by using public key protection of the session key, because the risk of passphrase unintended disclosure increases with the duration of the period of use.
- Archiving
  Although not primarily intended for this use, one may consider using the Encryption Facility for OpenPGP for securely archiving data. The use of a passphrase for long term archiving might not be a proper choice because adequate passphrase protection has to be guaranteed over a possibly long period of time, with quick access to the key in an emergency situation. The passphrase management processes may also get complicated as many data sets or files have to be archived.

  Public key protection of the session key is probably a better solution in that case, but still keeping in mind that the chosen processes and technologies must ensure the secrecy and usability of the private key over long periods of time.

6.2 PBE: z/OS-to-z/OS examples

6.2.1 Protecting a z/OS UNIX file

This scenario illustrates the use of passphrase-based encryption (PBE) with Encryption Facility for OpenPGP. A file is encrypted and decrypted using PBE. To make things easier, this scenario will use two fictitious companies to demonstrate the process.

Xander of the fictitious Hyena Corp wants to send encrypted data to Anne of the fictitious ITSO Magic Box Company. The passphrase is used to protect the session key.

1. Xander encrypts the data using 128 bit AES. The session key is encrypted using TDES and the digest algorithm is SHA-265. The data that will be sent to Anne is an HFS file called plain.txt. Xander uses the UNIX script in Figure 6-1 to encrypt the file. The encrypted file sent to Anne is called encrypted.dat.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /etc/encryptionfacility/   \
-cipher-name AES_128      \
-digest-name SHA_1  \
-o /u/xander/data/encrypted.dat   \
-s2k-cipher-name TRIPLE_DES   \
-s2k-digest-name SHA256 \
-s2k-mode 3  \
-c /u/xander/data/plain.txt
```

Figure 6-1 Script for z/OS UNIX file encryption using passphrase (PBE)

The results of the execution of the script are shown in Figure 6-2, where Xander is interactively prompted for a passphrase value. Note that Xander incorporates uppercase and lowercase letters, the number zero, and the underscore character for his passphrase.

```
xander:/u/xander/openpgp: >encpbe.sh
CSD1001A Enter passphrase for passphrase-based encryption:
NOrmal_is_the_WatchWOrd
CSD0000A Confirm passphrase:
NOrmal_is_the_WatchWOrd
CSD0051I Command processing has completed successfully.
```

Figure 6-2 Results of the execution of the PBE encrypt script

2. Xander sends the data to Anne. Because the data is already encrypted, the transfer method does not need a high level of security.
3. Xander sends Anne the passphrase through a secure and trusted channel.
4. Anne uses the script in Figure 6-3 to decrypt the data. The encrypted data file contains all the necessary information for the Encryption Facility for OpenPGP to decrypt the file. The encrypted OpenPGP message file is self-describing regarding options such as cipher-name or s2k-cipher-name. The results of the execution of the script can be seen in Figure 6-4, where Anne is prompted to enter the passphrase.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /etc/encryptionfacility/   \
-o /u/anne/data/decrypted.dat  \
-d /u/anne/data/encrypted.dat
```

Figure 6-3 Script to decrypt the file

```
anne:/u/anne/openpgp: >decpbe.sh
CSD1002A Enter passphrase for passphrase-based decryption:
NOrmal_is_the_WatchWOrd
CSD0051I Command processing has completed successfully.
```

Figure 6-4 Results of the execution of PBE decrypt script

6.2.2 Protecting a z/OS data sets

This scenario illustrates how the Encryption Facility for OpenPGP can encrypt and decrypt z/OS data sets. The process is demonstrated by using a passphrase to encrypt and decrypt the session key. The scenario uses two fictitious companies to demonstrate the process.

Faith from fictitious Alpha Corp plans to send sensitive data over to Kendra of the fictitious Omega Company. The data is in a PDS member.

1. Faith's data is contained in the data set FAITH.INFO(DATA). Faith uses the script in Figure 6-5 to encrypt the data.

   The input data set, FAITH.INFO(DATA), is a fixed block format data set. The output data set, FAITH.ENCRYPT.FOR.KENDRA(ENCRYPT), is pre-allocated as a variable block format data set.

   Note: The output data set for an encrypt operation cannot be fixed block format (RECFM=FB). The input data set can be of any format.

   The output from the execution of the script is in Figure 6-6.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /etc/encryptionfacility/   \
-cipher-name AES_128     \
-o '//faith.encrypt.for.kendra(encrypt)'  \
-s2k-cipher-name TRIPLE_DES   \
-s2k-digest-name SHA256 \
-s2k-mode 3  \
-c '//faith.info(data)'
```

Figure 6-5 Faith's PBE encrypt script

Note: Since the data set is a PDS, its name must be enclosed with single quotation marks. If the data set was a physically sequential data set, no quotation marks would be required.

```
CSD1001A Enter passphrase for passphrase-based encryption:
GOOdbye_HOrses
CSD0000A Confirm passphrase:
GOOdbye_HOrses
CSD0051I Command processing has completed successfully.
```

Figure 6-6 Execution of Faith's encryption script

2. Faith sends the data set to Kendra. Faith also sends the passphrase to Kendra via a secure channel.

3. Kendra uses the script in Figure 6-7 to decrypt the data. The results of the script are shown in Figure 6-8. The output data set for decryption can be of any format. Use the same format as the original data set when pre-allocating the output data set to simplify the operation.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /etc/encryptionfacility/   \
-o '//kendra.alpha.data(dec)'  \
-d '//kendra.from.faith(enc)'
```

Figure 6-7 Kendra's decryption script execution

```
CSD1002A Enter passphrase for passphrase-based decryption:
GOOdbye_HOrses
CSD0051I Command processing has completed successfully.
```

Figure 6-8 Output from Kendra's script

6.2.3 Invoking the Encryption Facility for OpenPGP with JCL

This scenario illustrates how the Encryption Facility for OpenPGP can be invoked using JCL; specifically, using the JZOS Java batch launcher.

This scenario is similar to what was described in 6.2.2, "Protecting a z/OS data sets". One party encrypts some data and sends it over to their business partner. The session key is encrypted using a passphrase and the data is in a PDS data set. Thus, we use the same scenario, that is Faith sending encrypted data to Kendra. All the data sets shown in the examples are pre-allocated.

Note: The JCL samples that we use are modified versions of the samples provided in:

- SYS1.SAMPLIB(CSDJZSVM)
- SYS1.SAMPLIB(CSDSMPEN)
- SYS1.SAMPLIB(CSDSMJCL)

The JCL statements used by Faith are shown in Figure 6-9.

```
//PROCLIB JCLLIB ORDER=FAITH.EF2.PROCLIB
//ENCPBE  EXEC PROC=CSDJZSVM,VERSION='50'
//STDENV  DD DSN=FAITH.EF2.LIB(CSDSMPEN),DISP=SHR
//*
//MAINARGS  DD *
-homedir /etc/encryptionfacility/
-cipher-name TRIPLE_DES
-o //FAITH.ENCRYPT.FOR.KENDRA(ENCRYPT)
-s2k-passphrase GOOd_bye_HOres
-c //FAITH.INFO(DATA)
/*
```

Figure 6-9 Faith's JCL used to encrypt data using PBE

The JCL statements used by Kendra to decrypt are shown in Figure 6-10.

```
//PROCLIB JCLLIB ORDER=KENDRA.EF2.PROCLIB
//DECPBE EXEC PROC=CSDJZSVM,VERSION='50'
//STDENV DD DSN=KENDRA.EF2.LIB(CSDSMPEN),DISP=SHR
//*
//MAINARGS  DD *
-homedir /etc/encryptionfacility/
-o //KENDRA.ALPHA.DATA(DEC)
-s2k-passphrase GOOd_bye_HOres
-d //KENDRA.FROM.FAITH(ENC)
/*
```

Figure 6-10 Kendra's JCL to decrypt

Note: In a production environment, it is highly recommended that both partners keep the passphrase specified only with the S2K_PASSPHRASE option in their own configuration file.

6.2.4 Troubleshooting

We encountered the following error during our testing, and felt it requires some explanation:

```
Given final block not properly padded
```

This error message occurs when there is a password problem during the execution of a command.

For example, when the keystore type is JCECCARACFKS or JCERACFKS, the keystore password and the key password must be the same. If they are not, then this error message will appear. Figure 6-11 displays details of the execution of a script with mismatched passwords.

```
+ java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar -homedir
/u/sjon/ -digest-name SHA_1 -keystore HyenaRing -keystore-type JCECCARACFKS
-racf-keyring-userid SJON -o /u/sjon/certs/racfOne.cer -eA Hyena Corp Self
Signed Cert
CSD0002A Enter keystore password for HyenaRing:
password
CSD0777I User ID SJON will be used to load RACF keyring HyenaRing.
CSD0029I Exporting an OpenPGP certificate for Hyena Corp Self Signed
Cert...
CSD0001A Enter key password for Hyena Corp Self Signed Cert:
not_the_same_password
CSD1348I Error encountered while attempting to export Hyena Corp
Self Signed Cert
        Error Message: Given final block not properly padded
CSD0050I Command processing ended abnormally:
        CSD1338I 1 errors were encountered while attempting to
export OpenPGP certificates.
```

Figure 6-11 Example of `Given final block not properly padded` error

6.3 Protection-based Encryption with a non-z/OS client

In this example we are exchanging data between a z/OS system and a Windows® platform that is running the PGP Desktop 9.6 software.

Encryption on z/OS for decryption on non-z/OS

The Encryption Facility for OpenPGP is invoked on the z/OS side using the script shown in Figure 6-12. The z/OS UNIX file movie-list.txt is encrypted with the default T-DES algorithm, the result is put in the movie-list.pgp file, and the session key is itself protected with the passphrase “paola.”

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/krogers -s2k-passphrase paola -o movie-list.pgp -t \
-c movie-list.txt
```

Figure 6-12 Data encryption in z/OS

Note that the text to be encrypted resides in EBCDIC encoding in the input z/OS UNIX file. It must be recovered as the equivalent ASCII text by the receiving Windows platform. The -t option in the script indicates that the Encryption Facility for OpenPGP should proceed with character set translation as explained in Appendix B, “Configuration file options” on page 237. In our case here, since there is no character set specified with the -t option, the Encryption Facility for OpenPGP translates the data into UTF-8 encoding before proceeding with encryption. It is then expected that the recipient system will be able to proceed with UTF-8 to local code page translation once the received data has been decrypted.

Once encrypted, the resulting file is sent using FTP to the non-z/OS system. The FTP transfer must be specified to occur in binary format because the z/OS FTP server must not proceed with a character set translation of its own.

When the PGP Desktop product is invoked and the file to decrypt specified, the decryption passphrase is required to be entered in the pop-up window, as shown in Figure 6-13.

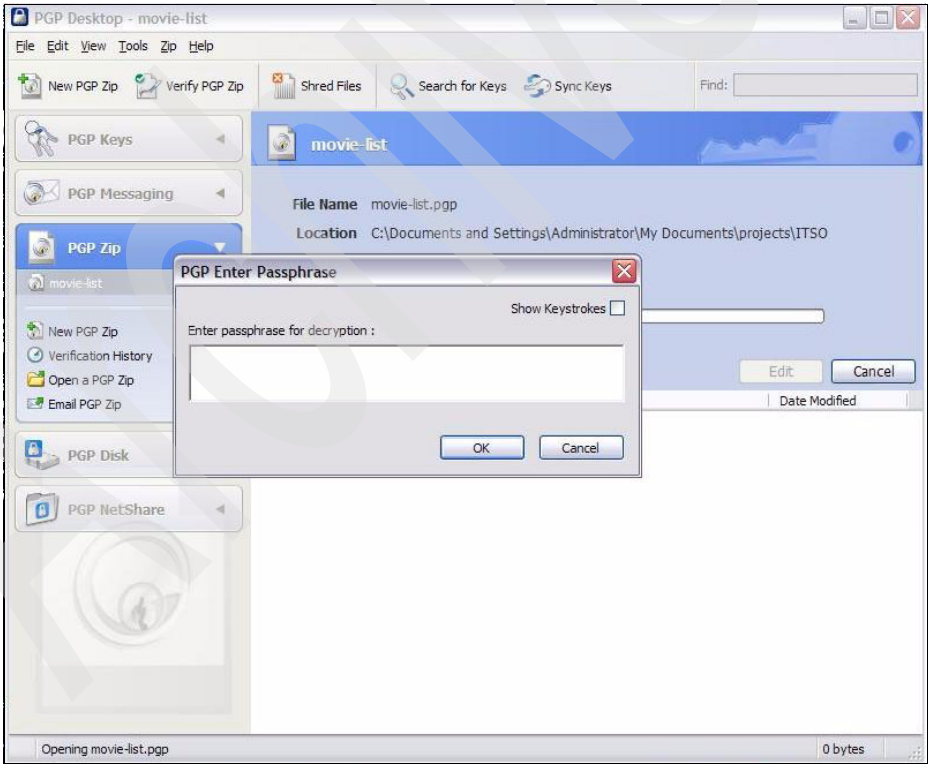


Figure 6-13 Asking for the PGP passphrase (PGP Corporation, Reprinted by Permission)

The pop-up window is to allow the passphrase to be entered. Once the valid passphrase is typed in, the file undergoes decryption. Note that the decrypted file is made available to other programs only after a PGP Desktop “extraction” step, as indicated in the pop-up message shown in Figure 6-14.

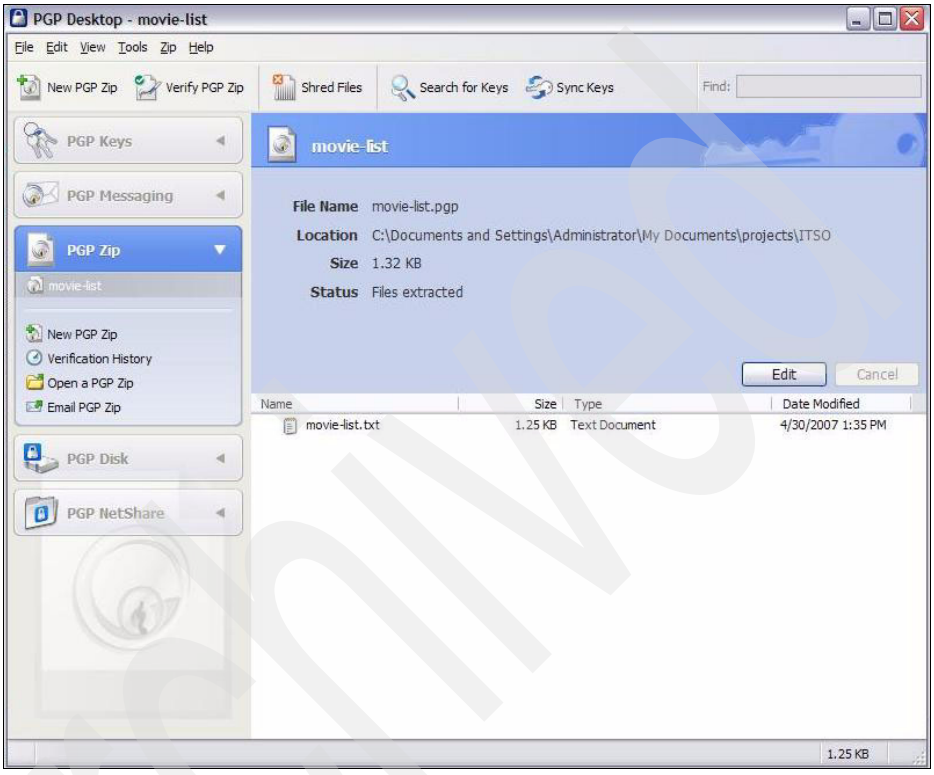


Figure 6-14 Decryption of the file produced by the Encryption Facility for OpenPGP (PGP Corporation, Reprinted by Permission)

Displaying the file after decryption shows that the initial z/OS UNIX file content was retrieved on the Windows platform.

Encryption on non-z/OS for decryption on z/OS

When encrypting a file with PGP Desktop on the Windows platform, the user is prompted for a password as shown in Figure 6-15.

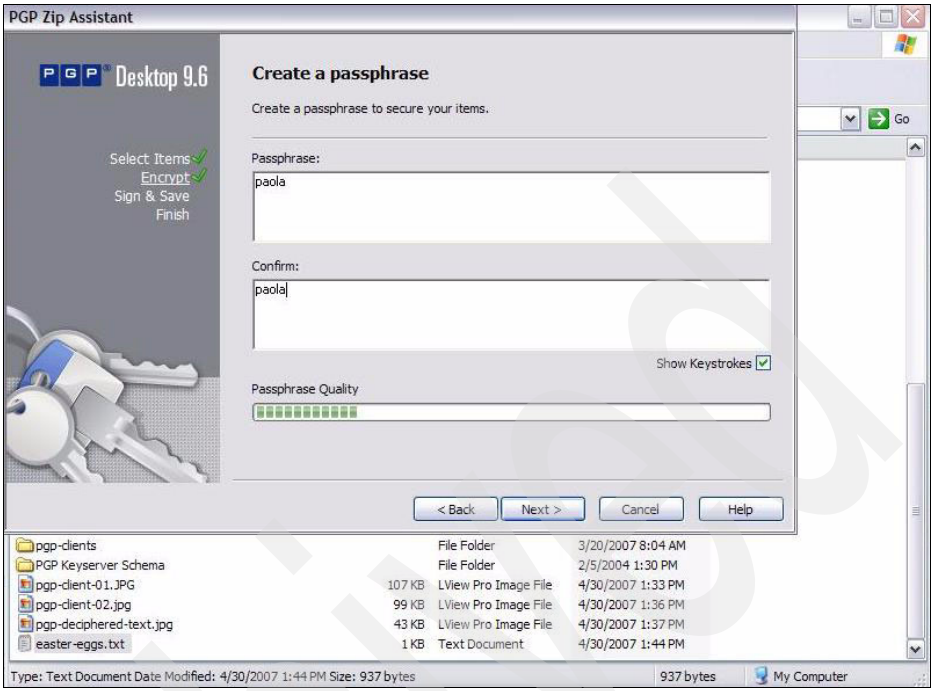


Figure 6-15 Passphrase specification for the PGP Desktop product (PGP Corporation, Reprinted by Permission)

Once the passphrase has been entered and confirmed, the PGP software proceeds with the encryption as shown in Figure 6-16.

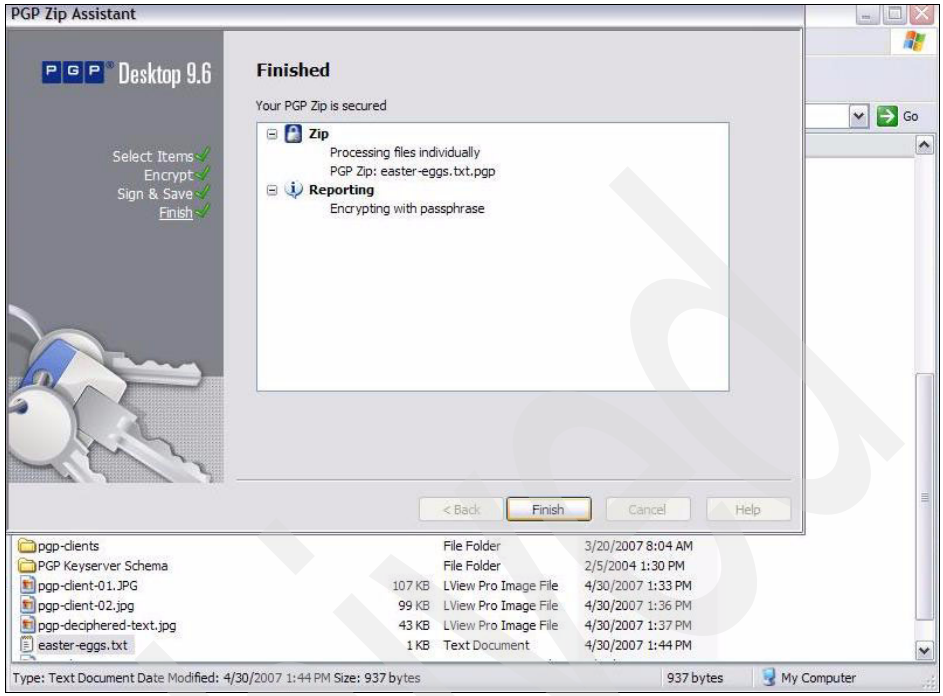


Figure 6-16 File encryption (PGP Corporation, Reprinted by Permission)

The resulting PGP file is sent to z/OS as a binary FTP transfer.

The decryption is performed with the script shown in Figure 6-17.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar\
-homedir /u/krogers \
-s2k-passphrase paola -o easter-eggs.txt -p easter-eggs.txt.pgp
```

Figure 6-17 Script to decrypt the PGP file

7

Session key protection: Public key cryptography

As already described in 5.5, “Encryption Facility for OpenPGP and digital certificates” on page 70, the Encryption Facility for OpenPGP can be used with both X.509 V3 and OpenPGP certificates. Using X.509 certificates allows you to exploit the X.509 support functions already built into z/OS, whereas the use of OpenPGP certificates is dictated by the need to inter-operate with other OpenPGP-compliant systems.

This chapter describes the facilities that are available to users of the Encryption Facility for OpenPGP for managing both X.509 and OpenPGP certificates and provides examples of the use of public key cryptography to protect the OpenPGP session key.

Refer to the discussion in 6.1, “Choosing PBE or public key cryptography” on page 86, for considerations on which type of session key protection best fits your needs.

7.1 Basic X.509 V3 certificate services in z/OS

There are three facilities in z/OS that provide certificate services that can be exploited in the context of use of the Encryption Facility for OpenPGP. This section describes these facilities, which are:

- The RACF (or equivalent vendor product) RACDCERT command
- The ICSF ISPF panels for the PKDS key management
- The Java keytool and hwkeytool utilities

Note that there are also certificate management functions provided by the Encryption Facility for OpenPGP itself. These functions are described at 7.2, “OpenPGP certificates management” on page 103.

7.1.1 Certificate services with RACDCERT

The first method of creating and managing certificates and asymmetric keys is to use the RACF command RACDCERT. RACF supports RSA keys and, beginning with z/OS V1R7, DSA keys as well.

The RACDCERT command can be used to create public and private keys, and to store them in the RACF database along with X.509 V3 certificates containing the public keys. The command can also be used to create certificates whose keys are stored in the ICSF PKDS dataset (for RSA keys only). Note that the RACDCERT command has been enhanced by APAR OA13030, so that the user can specify which PKDS label to use to store RSA private and public keys. (Prior to this enhancement the label name would be automatically generated during the execution of the RACDCERT GENCERT or RACDCERT ADD commands.)

Note: Once APAR OA13030 has been installed, a detailed description of the utility enhancements can be found in SYS1.SAMPLIB(IRR13030).

What the RACDCERT command does

Refer to *z/OS Security Server RACF Command Language Reference*, SA22-7687, for complete details regarding the RACDCERT command syntax.

RACDCERT has many functions. These are specified in the first parameter of the command. The parameters which are of importance in the context of use of the Encryption Facility for z/OS are shown in Table 7-1.

Access to these functions is protected by RACF profiles in the FACILITY class. The profile names are in the form IRR.DIGTCERT.<function_name>. Access to the function is not controlled for RACF SPECIAL users.

Table 7-1 The RACDCERT command functions

Function	Description
ADD	Add a certificate into the RACF data base
ADDRING	Create a RACF keyring
ALTER	Change the trust status of a certificate
CHECKCERT	Check whether a certificate has already been added to the RACF database
CONNECT	Connect a certificate to a keyring
DELETE	Delete a certificate from the RACF database
DELRING	Delete a keyring
EXPORT	Export a certificate as a PKCS#7 package or a PKCS#12 package, or as a binary DER or base64 encoded X.509 V3 certificate
GENCERT	Generate a key pair and certificate
GENREQ	Generate a PKCS#10 certificate request
LIST	List the contents of a certificate
LISTRING	List the contents of a keyring
REMOVE	Remove a certificate from a keyring

7.1.2 Certificate services with ICSF panels (RSA keys only)

The second method of generating RSA keys is to use an enhancement to the ICSF panels. This enhancement is supplied by the fix to APAR OA15156. 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 then the user must be permitted in RACF to the chosen key label as defined with a profile in the CSFKEYS class of resources.

Other options on the panel allow the deletion of keys, the exporting of an RSA public key to a self-signed X.509 V3 certificate, and also the ability to import a public key that is contained in an existing X.509 V3 certificate file.

Certificates used in this panel should be stored in sequential datasets with RECFM=VB. The certificates are DER (Distinguished Encoding Rule) encoded

and contain binary data. When transporting such certificates they should not undergo any character translation, that is, the binary pattern must remain as it is since it has been generated. The datasets used by these panels are similar to the datasets created by the RACDCERT EXPORT command when the parameter CERTDER parameter is used.

Complete details about the enhancement can be found at:

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

Note: The following restrictions apply to X.509 certificate management when performed using the ICSF PKDS Key Management panel:

- The certificates are not verified when imported in the PKDS.
- Exported certificates are always self-signed certificates, with the subject's, and issuer's identifiers composed of the **Common Name** (CN) attribute only.

7.1.3 Java keytool and hwkeytool

The IBM SDK comes with two utilities:

- The *keytool* utility, in the IBMJCE provider: This utility is intended to manage keys in the JCEKS, that is, it operates by software only. More details about the externals of keytool are provided at:

  `http://www.ibm.com/developerworks/java/jdk/security/142/secguides/keytoolDocs/KeyToolUserGuide-142.html`

- The *hwkeytool* utility, in the IBMJCECCA provider: This utility provides services similar to keytool but is intended to operate on ICSF key tokens. It is used to manage keys in the JCECCAKS keystore. Information on hwkeytool can be found at:

  `http://www.ibm.com/servers/eserver/zseries/software/java/hwkeytool.html`

Information about the Java keystores is provided in 3.2.5, "Keystores that can be used with the IBM JCE providers" on page 42.

Following are specific conditions pertaining to the use of keytool and hwkeytool:

- None of the utilities can write keys in the RACF database. From the Java usage standpoint the RACF keyrings are read-only keystores. If one needs to write into the RACF keyrings then one must get proper authorization to use the RACDCERT command.

- The Java hwkeytool can be used to generate RSA key pairs in the ICSF PKDS. Il also writes the corresponding X.509 certificate in the mapping JCECCAKS entry.
- The Java keytool can be used to generate and manage key pairs for the following algorithms, in addition to the RSA algorithm:
  - DSA
  - ElGamal

7.2 OpenPGP certificates management

The Encryption Facility for OpenPGP provides the following functions for OpenPGP asymmetric keys and certificates management:

- OpenPGP certificates are kept in an OpenPGP keyring, which is automatically built if not already existing and required by the OpenPGP function. The Open PGP keyring is a set of three z/OS UNIX files:
  - The keyring itself, containing OpenPGP certificates only
  - An index file to retrieve keyring entries by user ID
  - An index file to retrieve keyring entries by Key ID

  If required, the two index files can be rebuilt with the -rebuild-key-index command.

  Important: There are no private keys stored in the OpenPGP keyring. Private keys are kept in the Java keystores.

- Generation of key pairs and certificates (-g command)

  The command interactively prompts the user for the key type to be generated. It generates an asymmetric key pair with the private key and X.509 certificate in the designated Java keystore, and the corresponding OpenPGP certificate in the designated OpenPGP keyring.

  If the Java keystore is the JCECCAKS, the key pair is RSA only and it can be optionally stored in the ICSF PKDS.

  Note: The -g command proposes as an option, when using the JCEKS, to generate an ElGamal key pair and certificate. In that case the corresponding X.509 certificate is kept in the JCEKS, whereas the OpenPGP certificate is built with an RSA or DSA primary key and an ElGamal subkey.

- Mapping of existing PKDS key tokens to entries in the JCECCAKS, with the optional creation of an OpenPGP certificate (-prepare command).

  Optionally the command can build an OpenPGP certificate in the designated OpenPGP keyring.

- Importation of an OpenPGP certificate into an OpenPGP keyring (-i command). During the execution of the keyring the *fingerprint* of the certificate is displayed.

  Note: Fingerprint is a common way for OpenPGP partners to verify certificates they receive, in a context where centralized certificate authorities are seldom used. The fingerprint has to be exchanged via a trusted channel.

- Exportation of OpenPGP certificates from a Java keystore or an OpenPGP keyring:
  - By alias (-eA). This retrieves the Java keystore entry, builds the corresponding Open PGP certificate if it is not already available in the OpenPGP keyring, and makes it available in the specified z/OS UNIX file.

    Note: The Java keystore alias is case-insensitive unless it designates a certificate or key label in a RACF keyring.

  - By Key ID (-eK). This also retrieves both the Java keystore and the corresponding OpenPGP, if any, keyring entries. It builds the corresponding Open PGP certificate if it is not already available in the OpenPGP keyring, and makes it available in the specified z/OS UNIX file.
  - By User ID (-eP). This command retrieves certificates in the OpenPGP keyring and makes them available in the specified z/OS UNIX file.
- List OpenPGP, and whenever applicable, Java keystores keys and certificates information by the specified pointer:
  - By alias (-pA). This retrieves both the Java keystore and the corresponding Open PGP keyring entries.
  - By Key ID (-pK). This also retrieves both the Java keystore and the corresponding OpenPGP keyring entries. Because the Key ID is not implemented as a pointer in Java keystores, the Encryption Facility for OpenPGP computes the key ID as it scans keys in Java keystores.
  - By User ID (-pP). This command retrieves keys and certificates in the OpenPGP keyring.

- Delete OpenPGP, and whenever applicable, Java keystores keys and certificates, by the specified pointer:
  - By alias (-xA). This command also deletes entries in the Java keystore.
  - By Key ID (-xK). This command also deletes entries, if found, in the keystore.
  - By user ID (-xP). This command deletes entries in the openPGP keyring only.
- Provide a local Certificate Authority signature key via the -system-CA-key-alias, which points to a key pair in a Java keystore. The corresponding OpenPGP certificate can be shipped to partners, to be installed in their OpenPGP keyrings so that it can be used to verify the signature of the OpenPGP certificates they receive.

Attention: OpenPGP certificates generated by the Encryption Facility for OpenPGP contain the default user preferences, that is, an Encryption facility user cannot specify his or her own preferences in the certificate. The default preferences are:

- Cipher name: T-DES (data or passphrase protection)
- Digest name: SHA-1 (data integrity or passphrase string to key derivation)
- Compress name: ZIP

7.3 Public key protection scenarios

This section provides simple examples of the use of public key protection of the session keys. A more sophisticated example of public key infrastructure setup between OpenPGP partners is provided in Chapter 8, “Certificate Authority: X.509 and OpenPGP coexistence” on page 141.

7.3.1 z/OS to z/OS using RACF self-signed certificates

This scenario illustrates how RACF self-signed certificates can be used with the Encryption Facility for OpenPGP. One self-signed certificate will have its private key stored in the JCECCAKS, that is, the ICSF PKDS; the other self-signed certificate will have its private key stored in the JCERACFKS, that is, in the RACF database. Figure 7-1 is a graphical description of the flow of interactions.

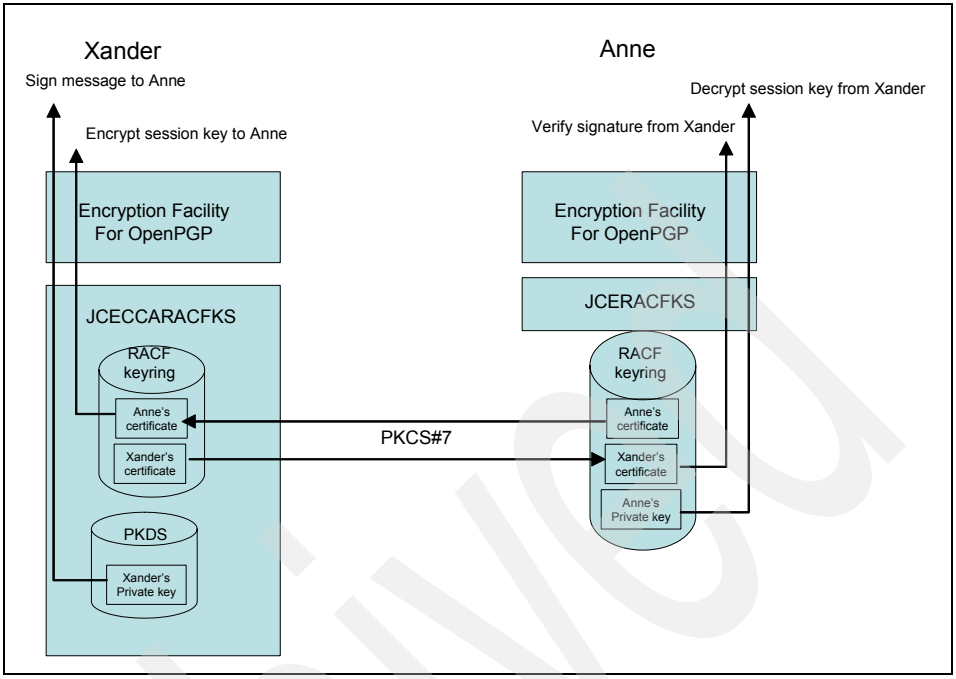


Figure 7-1 z/OS to z/OS using RACF self signed certificates

As in the PBE scenario, Xander of the fictitious Hyena Corp will be sending encrypted data to Anne of the fictitious ITSO Magic Box Company. However, this time, both parties will be using public key encryption to protect the session key. The public and private key pair will be managed by RACF in the form of an X.509 certificate. Using X.509 certificates may be the preferred option for organizations already familiar with the Public Key Infrastructure technologies and the use of X.509 entities.

Both Xander and Anne have the hardware JCE provider com.ibm.crypto.hdwrCCA.provider.IBMJCECCA set as the preferred provider in their configuration file (ibmef.config).

All of the following RACF RACDCERT commands are issued in batch mode with the JCL shown in Example 7-1.

Example 7-1 JCL used to issue RACDCERT commands

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

```
<RACDCERT LIST ID (XANDER)>
/*
```

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.

The scenario proceeds as follows:

1. Xander creates a self-signed certificate using RACF. The certificate has the label Hyena Corp Self Signed Cert. All certificate labels are case sensitive. The RSA keys are 2048 bit keys and are managed and kept by ICSF in the PKDS, with a PKDS label of HYENA.KEY. The RACDCERT command used is shown in Figure 7-2.

```
RACDCERT ID(XANDER) -
         GENCERT  -
         SUBJECTSDN(CN('Hyena Corp Self Signed Cert') -
                    O('Hyena Corp') -
                    L('Sunny') -
                    SP('CA') -
                    C('US')) -
         SIZE(2048) -
         WITHLABEL('Hyena Corp Self Signed Cert') -
         PCICC(HYENA.KEY)
```

Figure 7-2 Xander's RACDCERT command to create self-signed certificate

Note: When using the RACDCERT PCICC parameter, the generated private key never appears in clear, it is always provided encrypted with the coprocessor Master Key. Installations must be aware that this makes it impossible to set up a key restoration or backup scheme based on the knowledge of the clear value of the key.

2. Xander exports his self-signed certificate into a PKCS#7 certificate package that is to be sent to Anne. Xander does not need to export his certificate as an OpenPGP certificate since Anne will be using RACF to manage her certificates. The RACDCERT command used by Xander is shown in Figure 7-3.

```
RACDCERT ID(XANDER) EXPORT -
         (LABEL('Hyena Corp Self Signed Cert')) -
         DSN('XANDER.HYENA.CERT') -
         FORMAT(PKCS7B64)
```

Figure 7-3 Xander's RACDCERT export command

3. Now Xander needs to create a RACF keyring using the RACDCERT command shown in Figure 7-4.

```
RACDCERT ID(XANDER) -
         ADDRING(HyenaRing)
```

Figure 7-4 Xander's RACDCERT command to create a keyring

4. The next step for Xander is to connect his certificate to his keyring so that it can be made available to the Encryption Facility for OpenPGP. The USAGE of the certificate has to be PERSONAL since he needs access to the private keys. The RACDCERT command used is shown in Figure 7-5.

```
RACDCERT ID(XANDER) -
         CONNECT(ID(XANDER) -
         LABEL('Hyena Corp Self Signed Cert') -
         RING(HyenaRing) -
         USAGE(PERSONAL))
```

Figure 7-5 Xander's RACDCERT command to connect his self-signed certificate to his keyring

5. Anne also creates a self-signed certificate, but she will use the RSA key pair residing in the RACF database. Her certificate is called Magic Box Self Signed Cert, with a 1024 bit RSA key pair. Her RACDCERT command is shown in Figure 7-6.

```
RACDCERT ID(ANNE) -
         GENCERT  -
         SUBJECTSDN(CN('Magic Box Self Signed Cert') -
                    O('Magic Box') -
                    L('Sunny') -
                    SP('CA') -
                    C('US')) -
         SIZE(1024) -
         WITHLABEL('Magic Box Self Signed Cert')
```

Figure 7-6 Anne's RACDCERT GENCERT command

6. Anne exports her self-signed certificate to a PKCS#7 certificate package. She creates a keyring called MagicBoxRing and connects her self-signed certificate to it. The USAGE has to be PERSONAL since she needs access to her private keys. The syntax for the RACDCERT commands is similar to Xander's RACDCERT commands outlined in previous steps.
7. Both Xander and Anne send each other their PKCS#7 files. The exchange method does not have to be secure since the PKCS#7 file only contains the public key.
8. Anne imports Xander's certificate into her RACF database using the RACDCERT command shown in Figure 7-7. Hyena's certificate will be called Public Hyena Cert.

-

```
RACDCERT ID(ANNE) -
         ADD('ANNE.HYENA.CER') -
         WITHLABEL('Public Hyena Cert')
```

Figure 7-7 Anne's RACDCERT ADD command

9. Currently, in order for Java to access a RACF keyring, a PERSONAL and at least one CERTAUTH certificate must be in the RACF keyring, as discussed in the following note. To meet this requirement, Anne has to connect Public Hyena Cert, that is, Xander's certificate, to her keyring with a USAGE of CERTAUTH. The RACDCERT command used is shown in Figure 7-8.

```
RACDCERT ID(ANNE) -
         CONNECT(ID(ANNE) -
         LABEL('Public Hyena Cert') -
         RING(MagicBoxRing) -
         USAGE(CERTAUTH))
```

Figure 7-8 Anne's RACDCERT CONNECT command

Restriction: Currently, the RACF keyring support requires that both a PERSONAL certificate and a CERTAUTH certificate be connected to the KEYRING in order for you to use it through the Java services. When using self-signed certificates in Encryption Facility, it does not make sense that the receiver's certificate be marked as CERTAUTH since you will be using the receiver's public key to encrypt the session key. This is a known issue to Java development.

There are two ways to bypass this issue. You can connect any CA root certificate to the keyring and connect both self-signed certificates with USAGE of PERSONAL, or you can connect the receiver's certificate to the sender's keyring with USAGE of CERTAUTH.

10. Xander does the same with Anne's self-signed certificate that Anne sent him. He imports the certificate with the RACDCERT ADD command. He calls Anne's certificate Public Magic Box Cert. He connects the certificate to his keyring with USAGE of CERTAUTH.
11. Both Xander and Anne issue the RACDCERT LISTRING command to check that everything is in order. Xander's keyring is shown in Figure 7-9. Anne's keyring is shown in Figure 7-10.

```
Digital ring information for user XANDER:

Ring:
     >HyenaRing<
Certificate Label Name              Cert Owner     USAGE      DEFAULT
------------------------------      ------------   --------   -------
Hyena Corp Self Signed Cert         ID(XANDER)     PERSONAL     NO

Public Magic Box Cert               ID(XANDER)     CERTAUTH     NO
```

Figure 7-9 Output from Xander's LISTRING

```
Digital ring information for user ANNE:

Ring:
     >MagicBoxRing<
Certificate Label Name              Cert Owner     USAGE      DEFAULT
------------------------------      ------------   --------   -------
Magic Box Self Signed Cert          ID(ANNE)       PERSONAL     NO

Public Hyena Cert                   ID(ANNE)       CERTAUTH     NO
```

Figure 7-10 Output from Anne's LISTRING

12. Xander goes into the z/OS UNIX shell. He lists the contents of his keyring with the Encryption Facility command -pA. The script Xander used is in Figure 7-11. Note that since HyenaRing holds a certificate with a private key managed by ICSF, -keystore-type is set to JCECCARACFKS.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/xander/   \
-keystore HyenaRing \
-keystore-type JCECCARACFKS  \
-racf-keyring-userid xander  \
-pA
```

Figure 7-11 Xander's script for -pA

The output from Xander's -pA script is in Figure 7-12. Note that you can enter anything as the keystore password when the keystore is a RACF keyring.

Note also that no OpenPGP keyring has been created and, since the -PA command lists entries in a Java keystore, it will list the contents of the specified RACF keyring because they are entries in the JCECCARACFKS.

```
CSDO002A Enter keystore password for HyenaRing:
any_password
CSDO777I User ID xander will be used to load RACF keyring HyenaRing.
CSD1352I Displaying certificate with alias: hyena corp self signed cert
dn: CN=Hyena Corp Self Signed Cert, O=Hyena Corp, L=Sunny, ST=CA, C=US
primary: RSA 2,048  key ID: 92F7354CB470E624  fingerprint: D2DB5EA19D00951AA4BE3
A8F92F7354CB470E624  created: 3/26/07 "expired: 3/26/08"

CSD1352I Displaying certificate with alias: magic box self signed cert
dn: CN=Magic Box Self Signed Cert, O=Magic Box, L=Sunny, ST=CA, C=US
primary: RSA 1,024  key ID: 9F13CBF161203951  fingerprint: 0236B06F299C010FBA6FD
EF79F13CBF161203951  created: 3/26/07 "expired: 3/26/08"

CSDO051I Command processing has completed successfully.
```

Figure 7-12 Output from Xander's -pA script

Note: The -pK command will display the contents of both the keystore and the OpenPGP keyring. In this scenario, since the OpenPGP keyring is empty, the same output can be obtained with the command -pK.

If the -pP command is used, it will output nothing. The command -pP will only display the contents in the OpenPGP keyring.

13. The data Xander is sending to Anne is a z/OS UNIX file. Xander will compress, encrypt, and sign the data using the script shown in Figure 7-13.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/xander/     \
-cipher-name TRIPLE_DES  \
-digest-name SHA_1  \
-compress-name ZLIB  \
-keystore HyenaRing \
-keystore-type JCECCARACFKS \
-racf-keyring-userid xander  \
-o /u/xander/data/forAnne_signed_compress.enc  \
-rA "magic box self signed cert" \
-signers-key-alias "Hyena Corp Self Signed Cert" \
-s -e /u/xander/data/forAnne.txt
```

Figure 7-13 Xander's encrypt and sign script

The output of the script can be seen in Figure 7-14. The keystore password and the key password have to be the same in this execution of the script. The passwords can be anything you want.

Xander's private key has been used to sign the message. Anne's public key is used to encrypt the dynamically generated T-DES session key.

```
CSDO002A Enter keystore password for HyenaRing:
same_password
CSDO777I User ID xander will be used to load RACF keyring
HyenaRing.
CSDO768I Output data can be exchanged with the owner of the key
with key ID 9F13
CBF161203951.
CSDO001A Enter key password for hyena corp self signed cert:
same_password
CSDO051I Command processing has completed successfully.
```

Figure 7-14 Output from the encrypt and sign script

14. Xander sends the file over to Anne.
15. Anne decrypts it with the script in Figure 7-15. Since Anne's certificates do not have any keys managed by ICSF, the -keystore-type is JCERACFKS. Since the encrypted file sent by Xander contains all the cipher information, Anne does not need to specify anything in her decrypt script. The output from the script can be seen in Figure 7-16. As long as Anne has Xander's certificate in her keyring, the signature can be verified.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/anne/  \
-keystore MagicBoxRing  \
-keystore-type JCERACFKS  \
-racf-keyring-userid anne  \
-o /u/anne/data/forAnne.dec.txt \
-d /u/anne/data/forAnne_signed_compress.enc
```

Figure 7-15 Anne's decrypt script

During the execution of the script Anne's private key is used to decrypt the session key and Xander's public key is used to verify the signature.

```
CSDO002A Enter keystore password for MagicBoxRing:
same_password
CSDO777I User ID anne will be used to load RACF keyring
MagicBoxRing.
CSDO001A Enter key password for magic box self signed cert:
same_password
CSDO779I Signature by key ID 92F7354CB470E624 is valid.
CSDO051I Command processing has completed successfully.
```

Figure 7-16 Output from Anne's script

Note: The decrypt process also performs a verification of the signature. A separate verify step (using the -verify command) is not needed.

7.3.2 Using key pairs in the ICSF PKDS

This scenario illustrates how to make use of already existing RSA key pairs located in the PKDS and how to generate a new hardware RSA key pair with the Encryption Facility for OpenPGP commands. The scenario will again use fictitious companies to demonstrate the process.

Frank and Joe own separate security firms, and they need to exchange sensitive information with each other. They both have ICSF up and running on their processors. They each will exploit the added private key security ICSF provides by storing their RSA keys in the PKDS. Figure 7-17is a graphical description of the interactions between Frank's and Joe's systems.

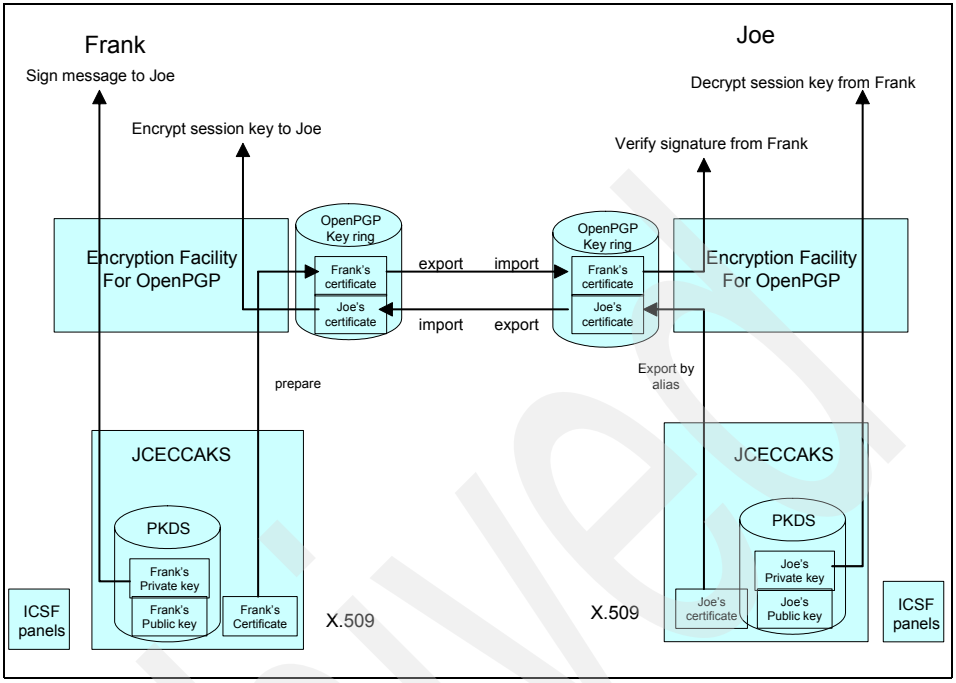


Figure 7-17 Using keys in the PKDS and certificates in the OpenPGP keyring

Both Frank and Joe have the hardware JCE provider set in their configuration file (ibmef.config).

The exchange between Frank and Joe proceeds as follows:

1. Frank creates his RSA keys with the ICSF panels. From the main ICSF panel, he uses option 5.6, as shown in Figure 7-18. He is successful, as shown in Figure 7-19.

```
--------------------------- ICSF - PKDS Keys ------------------
COMMAND ===>
Enter the PKDS record's label for the actions below
    ==>  FRANK.S2048.PREPARE

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

s  Generate a new PKDS key pair record
   Enter the key length  ===> 2048     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 7-18 Frank creates a RSA key pair using ICSF option 5.6

```
------------------ICSF - PKDS Key Request Successful----------
COMMAND ===>



  Label ==> FRANK.S2048.PREPARE




  Key function completed successfully

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

Figure 7-19 Frank successfully creates RSA keys

2. Frank needs to “prepare” his key to be used by the Java keystore and Encryption Facility for OpenPGP. He uses the script in Figure 7-20 to do this. Note that the keystore-type is JCECCAKS.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/frank/                                        \
-keystore /u/frank/keystore/jceccaks_keystore             \
-keystore-password frankharker                            \
-keystore-type JCECCAKS    \
-prepare frank.s2048.prepare
```

Figure 7-20 Frank’s prepare script

The output resulting from the prepare script is shown in Figure 7-21. Note that the prepare command essentially maps the RSA key pair in the PKDS to an entry in the JCECCAKS. It also builds a corresponding X.509 certificate in the Java keystore entry. In this case, the keystore is a z/OS UNIX file called jceccaks_keystore. It also creates an OpenPGP certificate and puts it into the OpenPGP keyring.

Note: Whenever the entry is deleted from the keystore, the RSA key pair will still remain in the PKDS. You must remove the key from the PKDS using ICSF services.

```
CSDO077A Preparing RSA key for keystore type JCECCAKS. Do you want to continue? (yes/no)
y
CSDO018A Enter alias for RSA key:
frank
CSDO044A For how many days should the X.509 certificate be valid (Maximum value 9999):
"365"
CSDO01OA What is your first and last name? "Unknown"
frank harker
CSDO011A What is the name of your organizational unit? "Unknown"
TS
CSDO012A What is the name of your organization? "Unknown"
IBM
CSDO013A What is the name of your city or locality?  "Unknown"
MKM
CSDO014A What is the name of your state or province?  "Unknown"
ON
CSDO015A What is the two-letter country code for this unit? "Unknown"
CA
CSDO016A Is <CN=stanley jon,OU=TS,O=IBM,L=MKM,ST=ON,C=CA> correct? (yes/no)
Y
CSDO001A Enter key password for frank:
frankharker
CSDO017A Confirm password:
frankharker
CSD1364I This key will be self-signed with alias: frank
CSDO040A Importing key into the OpenPGP key ring...
CSDO026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email address(optional).
CSDO02OA Real Name:
Frank Harker
CSDO024A Comment:
CSDO022A Email address:
CSDO025A You specified user ID: "Frank Harker"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSDO027A Add another user ID? (yes/no)
n
CSDO019A For how many days should this OpenPGP certificate be valid (0 for always valid):
"365"
CSDO038I Importing OpenPGP certificate with primary RSA public key ID: A4F426B53
D4503F8...
CSD1372I X.509 certificate summary for key alias frank
        Key ID: A4F426B53D4503F8
        Key Type: RSA
        Key Size: 2,048
        Keyring User ID(s): ""Frank Harker""
CSDO039I Total number processed successfully: 1
CSDO051I Command processing has completed successfully.
```

Figure 7-21 Output from Frank's prepare script

3. Frank displays his certificate information using the -pK command. He wants to see the contents of the keystore and the OpenPGP keyring. The script is in Figure 7-22. The results of the command are in Figure 7-23.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/frank/  \
-keystore /u/frank/keystore/jceccaks_keystore   \
-keystore-password frankharker   \
-keystore-type JCECCAKS \
-pK
```

Figure 7-22 Frank's -pK script

```
CSD1352I Displaying certificate with alias: frank
dn: CN=frank harker, OU=TS, O=IBM, L=MKM, ST=ON, C=CA
primary: RSA 2,048  key ID: A4F426B53D4503F8  fingerprint:
45084E24C76F30F33A130
110A4F426B53D4503F8  created: 3/23/07 "expired: 3/22/08"

CSD1353I Displaying OpenPGP certificate whose user IDs match:
"Frank Harker"
primary: RSA 2,048  key ID: A4F426B53D4503F8  fingerprint:
45084E24C76F30F33A130
110A4F426B53D4503F8  created: 3/23/07 "expired: 3/22/08"
uid:           "Frank Harker"
trust:   10 Trusted

CSD0051I Command processing has completed successfully.
```

Figure 7-23 Output of Frank's -pK script

Note: The -pK command is a very useful command to display certificates because it displays keys and certificates in both the Java keystore and the OpenPGP keyring.

4. Frank needs to export his certificate in order to give it to Joe. He will export using the key ID. The script he uses is in Figure 7-24. The output of the script is in Figure 7-25.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/frank/   \
-keystore /u/sjon/keystore/jceccaks_keystore \
-keystore-password frankharker    \
-keystore-type JCECCAKS  \
-o /u/frank/certs/frank.cer     \
-eK A4F426B53D4503F8
```

Figure 7-24 Frank's export script

```
CSD0029I Exporting an OpenPGP certificate for A4F426B53D4503F8...
CSD1347I 1 OpenPGP certificate(s) were exported successfully to
  /u/frank/certs/frank.cer.
CSD0051I Command processing has completed successfully.
```

Figure 7-25 Output of Frank's export script

5. Frank sends the certificate to Joe. The transportation method does not have to be too secure since this certificate only contains Frank's public key.
6. Joe imports it to his OpenPGP keyring. The script is in Figure 7-26 and the output from the script execution is in Figure 7-27.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/joe/   \
-i /u/joe/certs/frank.cer
```

Figure 7-26 Joe's import script

```
CSD0038I Importing OpenPGP certificate with primary RSA public key ID:
A4F426B53D4503F8...
CSD0073I Total number of certificates imported successfully: 1
CSD0051I Command processing has completed successfully.
```

Figure 7-27 Output from Joe's import script

7. Joe issues a -pK command to display Frank's certificate. The output from the -pK command is shown in Figure 7-28. Since Joe currently does not have any certificates in his keystore, all we see is Frank's certificate in the OpenPGP keyring.

```
CSD1353I Displaying OpenPGP certificate whose user IDs match:
"Frank Harker"
primary: RSA 2,048  key ID: A4F426B53D4503F8  fingerprint:
45084E24C76F30F33A130
110A4F426B53D4503F8  created: 3/23/07 "expired: 3/22/08"
uid:            "Frank Harker"
trust:   10 Trusted

CSD0051I Command processing has completed successfully.
```

Figure 7-28 Joe's -pK output

8. Joe generates a key pair using the -g command. The key pair is stored in the PKDS. The script he uses is in Figure 7-29. The -g command is interactive. Figure shows the full process to generate the key pair.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/joe/                                           \
-keystore /u/joe/keystore/jceccaks_keystore                \
-keystore-password joeharker                               \
-keystore-type JCECCAKS                                     \
-yes                                                        \
-g
```

Figure 7-29 Joe's script to generate key pair

```
CSD0046A You specified a keystore type JCECCAKS. Generating RSA key. Do you want to
continue? (yes/no)
Yes
CSD0018A Enter alias for RSA key:
Joe
CSD0004A Enter key size:"1024"
1024
CSD0044A For how many days should the X.509 certificate be valid (Maximum value 9999):"365"
365
CSD0010A What is your first and last name? "Unknown"
Joe Harker
CSD0011A What is the name of your organizational unit? "Unknown"
IGS
```

Figure continued on following page.

```
CSDO012A What is the name of your organization? "Unknown"
IBM
CSDO013A What is the name of your city or locality?  "Unknown"
MKM
CSDO014A What is the name of your state or province?  "Unknown"
CA
CSDO015A What is the two-letter country code for this unit? "Unknown"
ON
CSDO016A Is <CN=Joe Hardy,OU=IGS,O=IBM,L=MKM,ST=CA,C=ON> correct? (yes/no)
Yes
CSDO001A Enter key password for Joe:
joeharker
CSDO017A Confirm password:
joeharker
CSD1363I Generating RSA key...
CSDO048A Please select what type of hardware key you want:"1"
        (1) PKDS (2) CLEAR
1
CSD1331I An acceptable hash algorithm name was not found. Using: SHA_1
CSD1364I This key will be self-signed with alias: Joe
CSDO040A Importing key into the OpenPGP key ring...
CSDO026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email add
ress(optional).
CSDO020A Real Name:
Joe Harker
CSDO024A Comment:
CSDO022A Email address:
CSDO025A You specified user ID: "Joe Harker"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSDO027A Add another user ID? (yes/no)
n
CSD1331I An acceptable hash algorithm name was not found. Using: SHA_1
CSDO019A For how many days should this OpenPGP certificate be valid (0 for always
valid):"365"
365
CSDO038I Importing OpenPGP certificate with primary RSA public key ID: 76931F986BC5A2BC...
CSDO039I Total number processed successfully: 1
CSD1372I X.509 certificate summary for key alias Joe
        Key ID: 76931F986BC5A2BC
        Key Type: RSA
        Key Size: 1,024
        Keyring User ID(s): ""Joe Harker""
CSDO051I Command processing has completed successfully.
```

Figure 7-30 Output from key pair generate

The -g command uses the IBMJCECCA APIs to generate the RSA key pair and stores it in the PKDS. There is no documentation as to how the key label name is generated.

If you use IDCAMS PRINT to punch out the PKDS after the generate, you can see the key label looks something like:
IBM47589.11166712.85673465.88814750.17094274.28299572.7946

Important: Currently, when you delete the certificate that has its key generated by the -g command, the key will not be deleted from the PKDS. The IBMJCECCA APIs do not have the capability to delete the key from the PKDS. You will essentially have orphaned keys in the PKDS.

You will need to delete them manually as long as you know what the key label is. You can use ICSF option 5.6 or create a program that calls the ICSF APIs to delete a key record from the PKDS.

The only way to find the name of the key label is to dump out the contents of the PKDS. We recommend dumping out the PKDS contents before and after the -g command is issued to make searching for the new key label easier.

9. Joe displays the contents of his keystore and OpenPGP keyring with the -pK command. The output from the command is shown in Figure 7-31.

```
CSD0002A Enter keystore password for
/u/joe/keystore/jceccaks_keystore:
joeharker

CSD1352I Displaying certificate with alias: joe
dn: CN=Joe Harker, OU=IGS, O=IBM, L=MKM, ST=CA, C=ON
primary: RSA 1,024  key ID: 76931F986BC5A2BC  fingerprint:
67D924F10E965BBB74F41
BCA76931F986BC5A2BC created: 3/23/07 "expired: 3/22/08"

CSD1353I Displaying OpenPGP certificate whose user IDs match:
"Joe Harker"
primary: RSA 1,024  key ID: 76931F986BC5A2BC  fingerprint:
67D924F10E965BBB74F41
BCA76931F986BC5A2BC  created: 3/23/07 "expired: 3/22/08"
uid:            "Joe Harker"
trust:   10 Trusted

CSD1353I Displaying OpenPGP certificate whose user IDs match:
"Frank Harker"
primary: RSA 2,048  key ID: A4F426B53D4503F8  fingerprint:
45084E24C76F30F33A130
110A4F426B53D4503F8  created: 3/23/07 "expired: 3/22/08"
uid:            "Frank Harker"
trust:   10 Trusted

CSD0051I Command processing has completed successfully.
```

Figure 7-31 Output from Joe's -pK command

10. Joe has to export his public key into an OpenPGP certificate and send it to Frank. The script Joe uses to export his key is in Figure 7-32.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/joe/  \
-digest-name SHA_1  \
-keystore /u/joe/keystore/jceccaks_keystore  \
-keystore-type JCECCAKS   \
-o /u/joe/certs/joe.cer   \
-eA joe
```

Figure 7-32 Joe's export script

The export command is interactive. The export process can be seen in Figure 7-33

```
CSD0002A Enter keystore password for /u/joe/keystore/jceccaks_keystore:
joeharker
CSD0029I Exporting an OpenPGP certificate for joe...
CSD0001A Enter key password for joe:
joeharker
CSD0026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email address
(optional)
CSD0020A Real Name:
Joe Harker
CSD0024A Comment:

CSD0022A Email address:

CSD0025A You specified user ID: "Joe Harker"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSD0027A Add another user ID? (yes/no)
n
CSD0019A For how many days should this OpenPGP certificate be valid (0 for always valid):"0"

CSD0032A Do you want to add the exported OpenPGP Certificate to your OpenPGP keyring?
(yes/no)
y
CSD0075A Certificate <76931F986BC5A2BC> already exists in the key ring. Would you like to
replace it? (yes/no)
n
CSD1347I 1 OpenPGP certificate(s) were exported successfully to /u/joe/certs/joe.cer.
CSD0051I Command processing has completed successfully.
```

Figure 7-33 Output from Joe's export command

11. Joe sends the certificate file to Frank. Frank successfully imports Joe's certificate into his OpenPGP keyring.
12. Joe is now ready to encrypt the data he wants to send to Frank. He will use Frank's public key to encrypt the session key. Joe will also sign the data. The script he uses is shown in Figure 7-34.

    Joe uses the command option -rK to reference Frank's public key. The key IDs can be seen in the previous listings (Figure 7-31) of the keystore and OpenPGP keyring.

    The output of the execution of the command is shown in Figure 7-35.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/joe/  \
-cipher-name TRIPLE_DES  \
-compress-name ZLIB  \
-digest-name SHA_1  \
-keystore /u/joe/keystore/jceccaks_keystore  \
-keystore-type JCECCAKS \
-o /u/joe/data/forFrank_signed.enc  \
-rK A4F426B53D4503F8 \
-signers-key-alias joe  \
-s -e /u/joe/data/forFrank.txt
```

Figure 7-34 Joe's encrypt and sign script

Joe's private key is used to sign the message and Frank's public key is used to encrypt the session key.

```
CSDO002A Enter keystore password for /u/stan/keystore/jceccaks_keystore:
joeharker
CSDO768I Output data can be exchanged with the owner of the key with key ID A4F4
26B53D4503F8.
CSDO001A Enter key password for joe:
joeharker
CSDO051I Command processing has completed successfully.
```

Figure 7-35 Output of Joe's script

13. Joe sends the file to Frank. The transport channel does not have to be very secure since the data is encrypted.
14. Frank decrypts the file with the script in Figure 7-36. The output from the decrypt script is shown in Figure 7-37.

```
java  \
-jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/frank/
-keystore /u/frank/keystore/jceccaks_keystore  \
-keystore-type JCECCAKS  \
-o /u/frank/data/forFrank.dec.txt \
-d /u/frank/data/forFrank_signed.enc
```

Figure 7-36 Frank's decrypt script

```
CSDO002A Enter keystore password for /u/frank/keystore/jceccaks_keystore:
frankharker
CSDO001A Enter key password for frank:
frankharker
CSDO779I Signature by key ID 76931F986BC5A2BC is valid.
CSDO051I Command processing has completed successfully.
```

Figure 7-37 Output from Frank's decrypt script

7.3.3 ElGamal keys

The Encryption Facility for OpenPGP can use the ElGamal keys and algorithm to encrypt and decrypt the session key. Because the ElGamal algorithm can only be used for encryption, the ElGamal public key must appear as a subkey in a certificate that holds a signature-only primary key (refer to 2.5.2, "The OpenPGP certificate" on page 18 for an explanation of subkeys).

The JCE provider, IBMJCE, generates and manages ElGamal keys and X.509 certificates. However, we recommend that you use the -g command of the Encryption Facility for OpenPGP to trigger the generation of ElGamal keys and certificates. The dialog of the -g command proposes to the user a combination of an ElGamal key pair with an RSA or DSA key pair. When the user has selected an option, then both the RSA or DSA key pair and the ElGamal key pair are generated in the JCEKS. (Be aware that the ElGamal key generation can actually last for *minutes*). Two X.509 certificates are also created that wrap each public key of the chosen combination. Then the user can elect that the Encryption Facility for OpenPGP creates an OpenPGP certificate with the just generated RSA or DSA public key as the primary key and the ElGamal public key as the subkey.

Users using only X.509 entities will refer to the JCEKS alias of the ElGamal key and certificate. Users who prefer using OpenPGP certificates will refer to the JCEKS alias of the Elgamal private key and the OpenPGP certificate that contains the ElGamal public subkey.

7.3.4 Troubleshooting while running these scenarios

This section describes the meaning of several errors we encountered in the lab.

Information not available for protected private keys

This error message occurs during a decryption operation when the Java environment has not been set up correctly. Specifically, the unrestricted policy

files have not been installed. See the following web site for details:

http://www.ibm.com/servers/eserver/zseries/software/java/j5jcecca.html

Also, see “Setting up the list of providers” on page 40 for more details on Java setup.

Figure 7-38 shows an example of the error when it occurs during decryption.

```
KAPPELE:/u/kappele/openpgp: >rsaenc.sh
CSDO700A File </u/kappele/openpgp/data_out> already exists.  Do you
want to overwrite? (yes/no)
yes
CSD1311I An acceptable symmetric cipher algorithm name was not
found. Using: TRIPLE_DES
CSDO768I Output data can be exchanged with the owner of the key with
key ID BA2F68258E71F5C8.
CSDO051I Command processing has completed successfully.
KAPPELE:/u/kappele/openpgp: >rsadec.sh
CSDO050I Command processing ended abnormally:
        Information not available for protected private keys.
```

Figure 7-38 Example of “Information not available” error

Inappropriate key specification

This error was encountered during an export with alias (-eA) and decrypt commands. The keystore type was JCEKS and the key type was RSA. A PMR has been opened with the support center and they are investigating. See Figure 7-39 and Figure 7-40 for examples of the error message.

```
+ java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar -homedir
/u/sjon/ -digest-name SHA_1 -keystore /u/sjon/keystore/jceks_keystore
-keystore-type JCEKS -o /u/sjon/certs/stanley.cer2 -eA RSA key
CSD0002A Enter keystore password for /u/sjon/keystore/jceks_keystore:
stanjon
CSD0029I Exporting an OpenPGP certificate for RSA key...
CSD0001A Enter key password for RSA key:
stanjon
CSD1348I Error encountered while attempting to export RSA key
        Error Message: Inappropriate key specification:
DerInputStream.getLength
(): lengthTag=127, too big.
CSD0050I Command processing ended abnormally:
        CSD1338I 1 errors were encountered while attempting to
export OpenPGP certificates.
```

Figure 7-39 First example of “Inappropriate key specification” error

```
+ java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar -homedir
/u/sjon/ -keystore /u/sjon/keystore/jceks_keystore -keystore-password
stanjon -keystore-type JCEKS -o /u/sjon/data/forStanley.dec.txt -d
/u/sjon/data/forStanley.enc
CSD0001A Enter key password for rsa key:
stanjon
CSD0050I Command processing ended abnormally:
        Inappropriate key specification: DerInputStream.getLength():
lengthTag=127, too big.
```

Figure 7-40 Second example of “Inappropriate key specification” error

7.4 Using public key protection with non-z/OS systems

This section shows an exchange of encrypted messages between the Encryption Facility for openPGP and a Windows system that runs the PGP desktop product. The encryption session key is protected with the partner’s public key.

Note that we are specifying the following options in the configuration file of the Encryption facility:

```
KEY_RING_FILENAME /u/krogers/ibmpkring.ikr
JAVA_KEY_STORE_NAME /u/krogers/ibmef.jks
```

7.4.1 Generation of an RSA key pair on z/OS

The -g command of the Encryption Facility for OpenPGP is invoked using the shell script shown in Figure 7-41.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar\
-homedir /u/krogers/ \
-keystore /u/krogers/jceks_keystore \
-keystore-password rogersken \
-keystore-type JCEKS \
-yes \
-g
```

Figure 7-41 Script to generate the RSA key pair

As already seen with other examples of -g, a JCEKS entry is created for the key pair with an alias specified by the user. An X.509 certificate is also created to wrap the public key.

An OpenPGP certificate is also created and imported in the OpenPGP keyring.

The -g dialog with the user to display the characteristics of the created certificates is shown in Figure 7-42. Note the OpenPGP user ID, which has been specified by the user.

```
CSD1372I X.509 certificate summary for key alias rogers-key
        Key ID: 6FDD20654B49A620
        Key Type: RSA
        Key Size: 1,024
        Keyring User ID(s): "ken rogers <rogerske@us.ibm.com>""

CSD0051I Command processing has completed successfully.
```

Figure 7-42 Completion of the RSA key pair generation by the Encryption Facility

7.4.2 Generation of an RSA key pair by the PGP desktop

The key generation options are actually selected in the sequence of menus shown in Figure 7-43 and Figure 7-44.

.

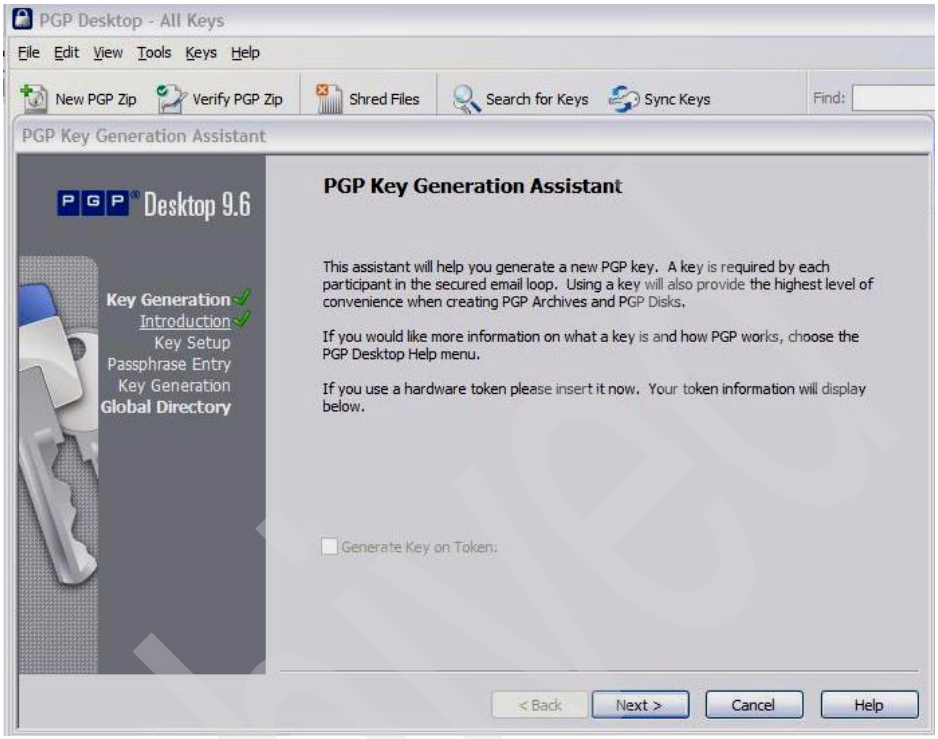


Figure 7-43 Calling PGP key generation assistant (PGP Corporation, Reprinted by Permission)

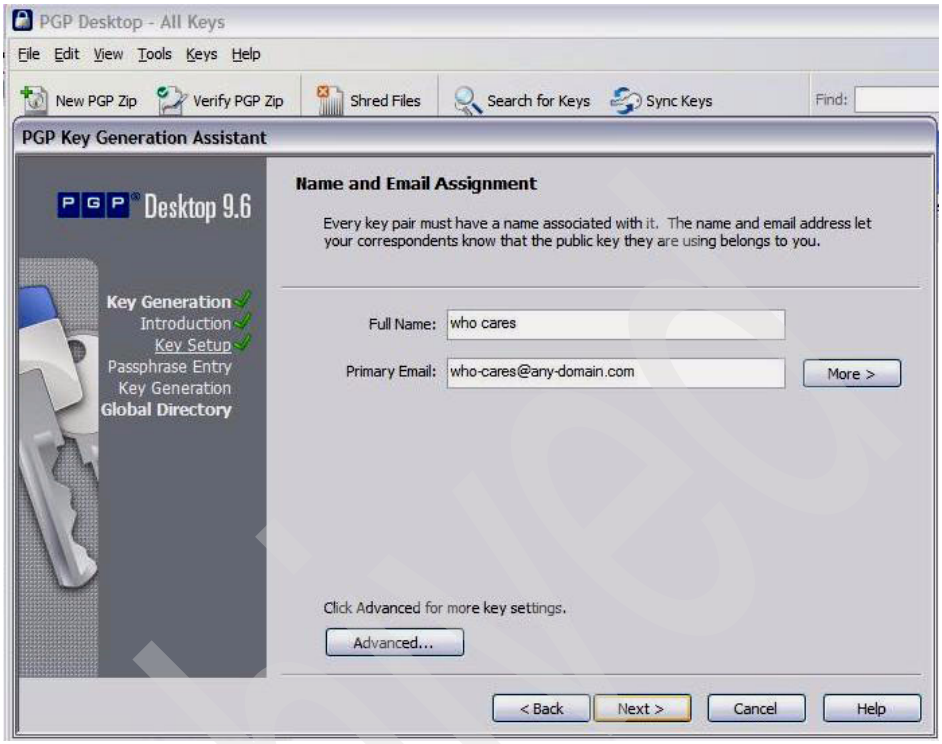


Figure 7-44 Specifying the PGP user ID (PGP Corporation, Reprinted by Permission)

7.4.3 Exporting the RSA public key from z/OS to non-z/OS

The public key is exported by key ID using the shell script shown in Figure 7-45.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/krogers/ \
-keystore  /u/krogers/jceks_keystore \
-keystore-password rogersken \
-keystore-typeJCEKS \
-o  /u/krogers/ken-rogers-z-keyid.asc \
-eK 6FDD20654B49A620
```

Figure 7-45 Exporting the public key

7.4.4 Key preparation for exportation on the PGP system

The PGP user invokes the menu shown in Figure 7-46 to select the key to export.

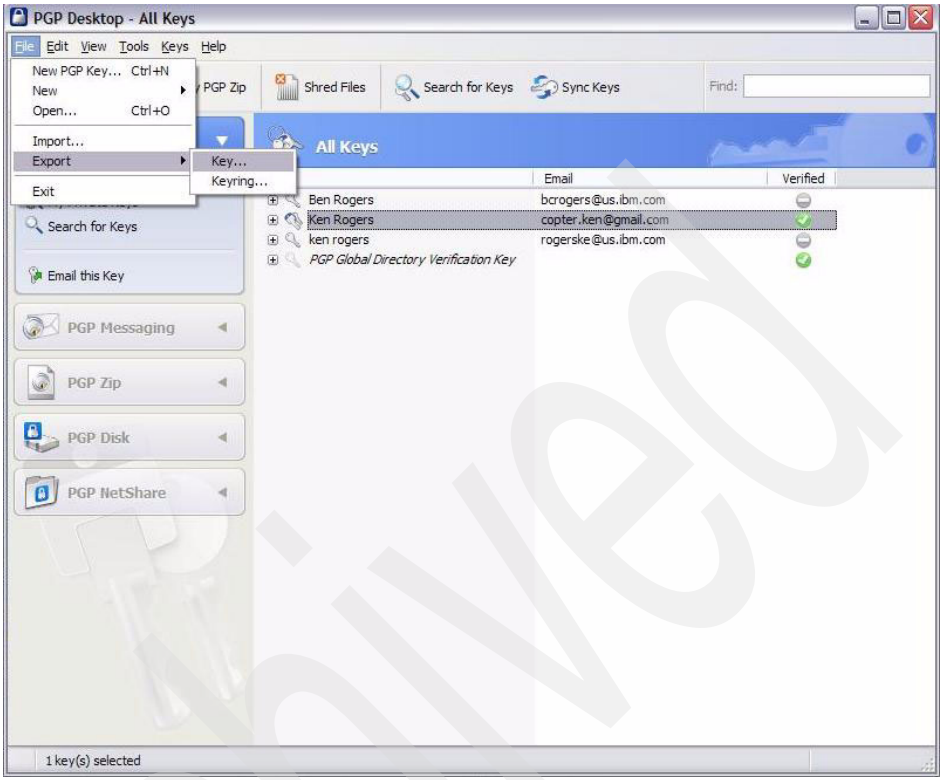


Figure 7-46 Selection of the key to be exported (PGP Corporation, Reprinted by Permission)

Then the user specifies the file to put the key into. Note that the user also specified here to wrap the certificate in an ASCII armor.

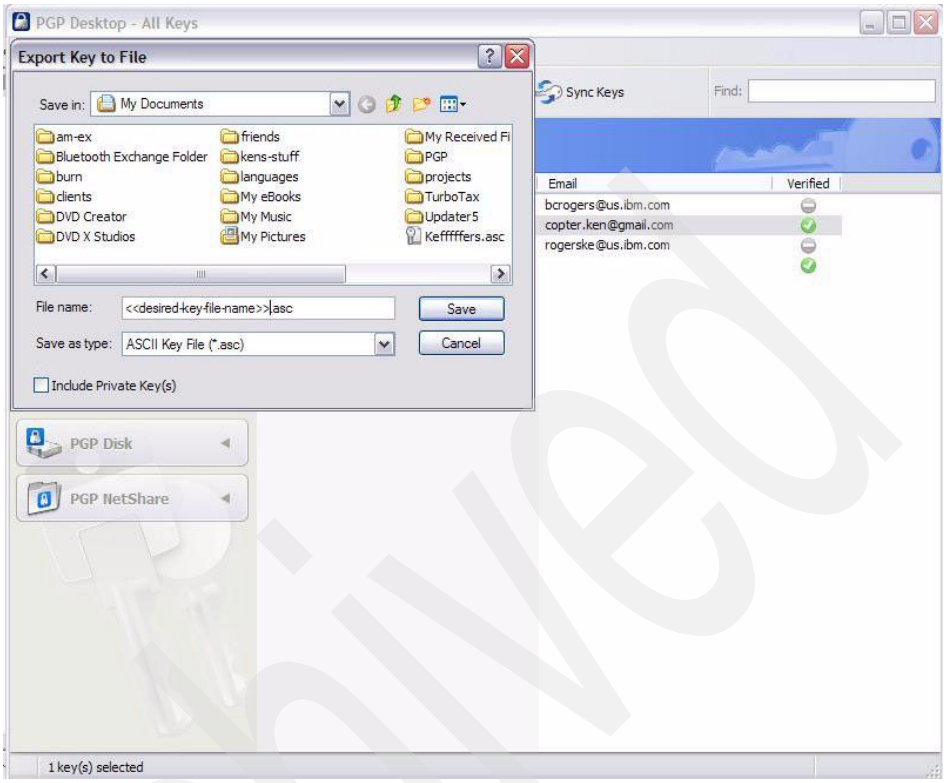


Figure 7-47 Storing the certificate in a file for exportation (PGP Corporation, Reprinted by Permission)

7.4.5 Exchanging the OpenPGP certificates

The certificate files are exchanged using FTP. Remember that ASCII armorized certificates must be transferred in text mode (that is, not binary mode).

7.4.6 Importing the partner's PGP certificate on z/OS

Once the PGP certificate has been received it can be imported in the OpenPGP keyring with the script shown in Figure 7-48.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/krogers/ \
-i /u/krogers/ken-rogers.asc
```

Figure 7-48 Script to import the PGP certificate

The certificate is then imported as shown in Figure 7-49.

```
CSD0038I Importing OpenPGP certificate with primary RSA public key
ID: C9B09BF199981408...
CSD0073I Total number of certificates imported successfully: 1
CSD0051I Command processing has completed successfully.
```

Figure 7-49 Importing the PGP certificate

7.4.7 Importing the partner's OpenPGP certificate on PGP Desktop

The menu shown in Figure 7-50 is used to import the OpenPGP certificate into the PGP keyring.

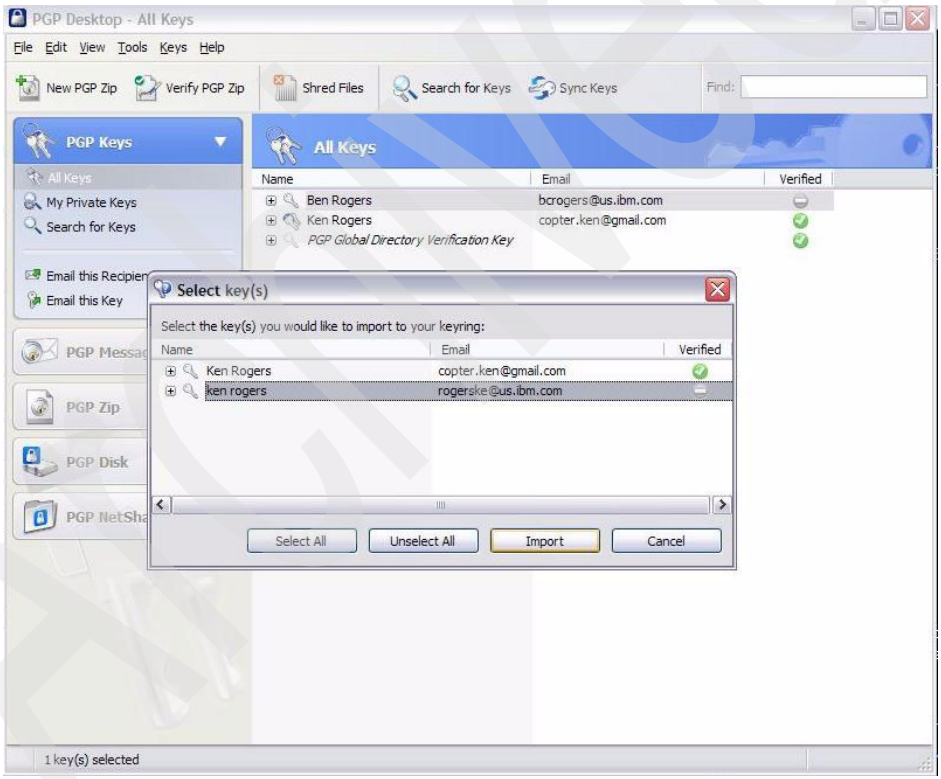


Figure 7-50 Selecting the certificate to import (PGP Corporation, Reprinted by Permission)

After completion of the import function, the menu in Figure 7-51shows the keys connected to the PGP keyring.

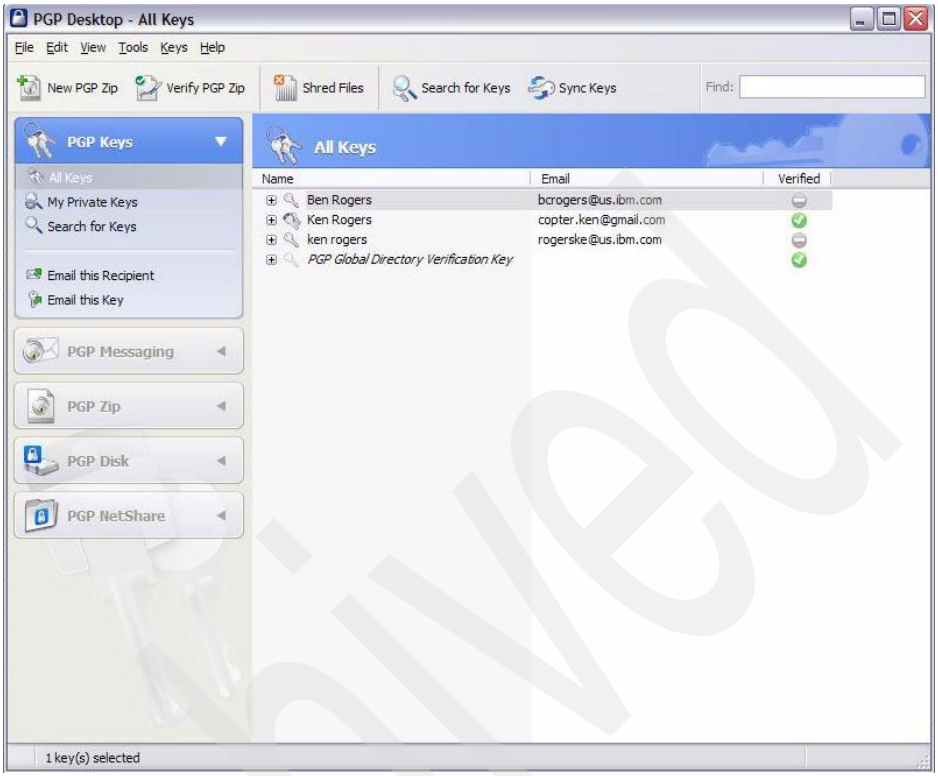


Figure 7-51 Keys in the PGP keyring (PGP Corporation, Reprinted by Permission)

7.4.8 Exchanging an encrypted file: z/OS to non-z/OS

Now that both partner's have each other's certificates, we show how the exchange of an encrypted file can be performed. We start with a file encrypted by the Encryption Facility for OpenPGP on z/OS and sent to the PGP Desktop.

We use the script in Figure 7-52, where the public key to use to protect the session key is designated by a user ID (copter.ken@gmail.com). Note also that a text character set translation is specified from whatever is the local code page to ASCII.

The file is sent to the PGP Desktop with FTP in binary mode. We describe the decryption of the file later.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/krogers/ \
-cipher-name TRIPLE_DES \
-o /u/krogers/educate.pgp \
-rP copter.ken@gmail.com \
-t US-ASCII \
-e /u/krogers/educate.txt
```

Figure 7-52 Encryption script

7.4.9 Exchanging an encrypted file: non-z/OS to z/OS

The PGP Desktop user invokes the menu shown in Figure 7-53 and selects the key to use to encrypt the session key.

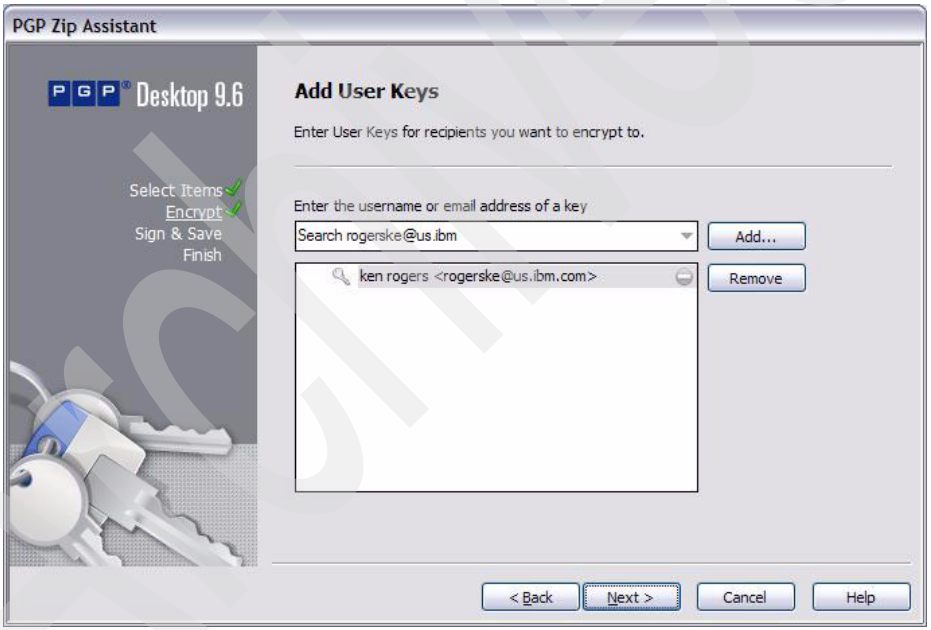


Figure 7-53 Selecting the public key to encrypt the session key (PGP Corporation, Reprinted by Permission)

It then triggers the file encryption as shown in Figure 7-54.

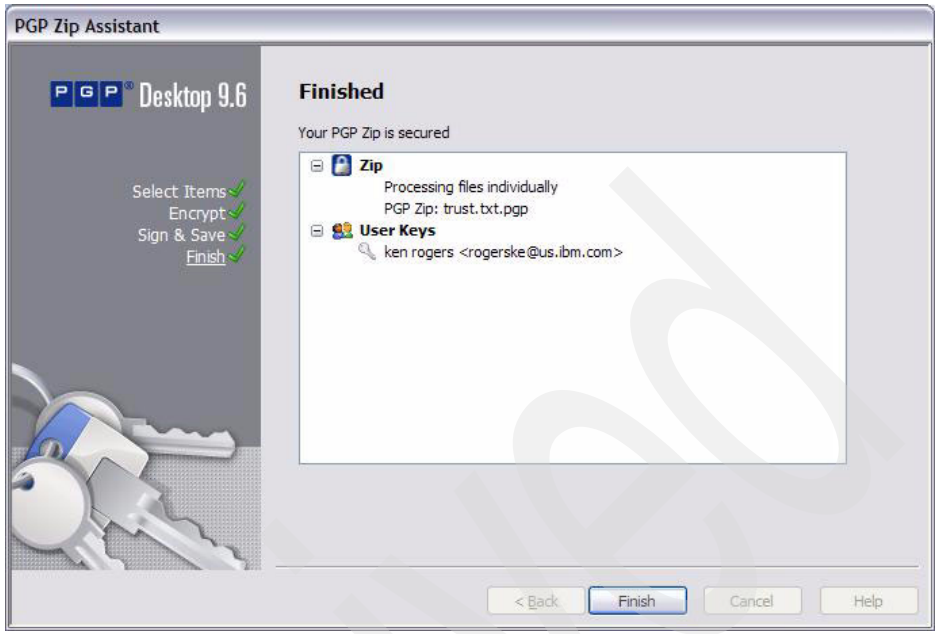


Figure 7-54 Encrypting the file (PGP Corporation, Reprinted by Permission)

The encrypted file is then transmitted to the z/OS system.

7.4.10 Decryption on the z/OS system

The Encryption Facility for OpenPGP is invoked with the script shown in Figure 7-55. Note that the public key ID that was selected at the PGP Desktop has been shipped with the file and will be used by the Encryption Facility to retrieve the corresponding private key in its keyring.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/krogers/ \
-keystore /u/krogers/jceks_keystore \
-o /u/krogers/educate.em.txt -t \
-d /u/krogers/educate.email.pgp
```

Figure 7-55 Decryption script

7.4.11 Decryption on the PGP Desktop

Invoking the decryption of the received file results in display of the panel shown in Figure 7-56. Note that the public key user ID has been carried over in the file. Now the PGP Desktop is going to use the corresponding private key to decrypt the session key.

Figure 7-56 Unlocking the private key (PGP Corporation, Reprinted by Permission)

Unlocking the private key allows the decryption of the file to proceed. When it is complete, the panel in Figure 7-57 is displayed, showing the decrypted file ready to be “extracted.”

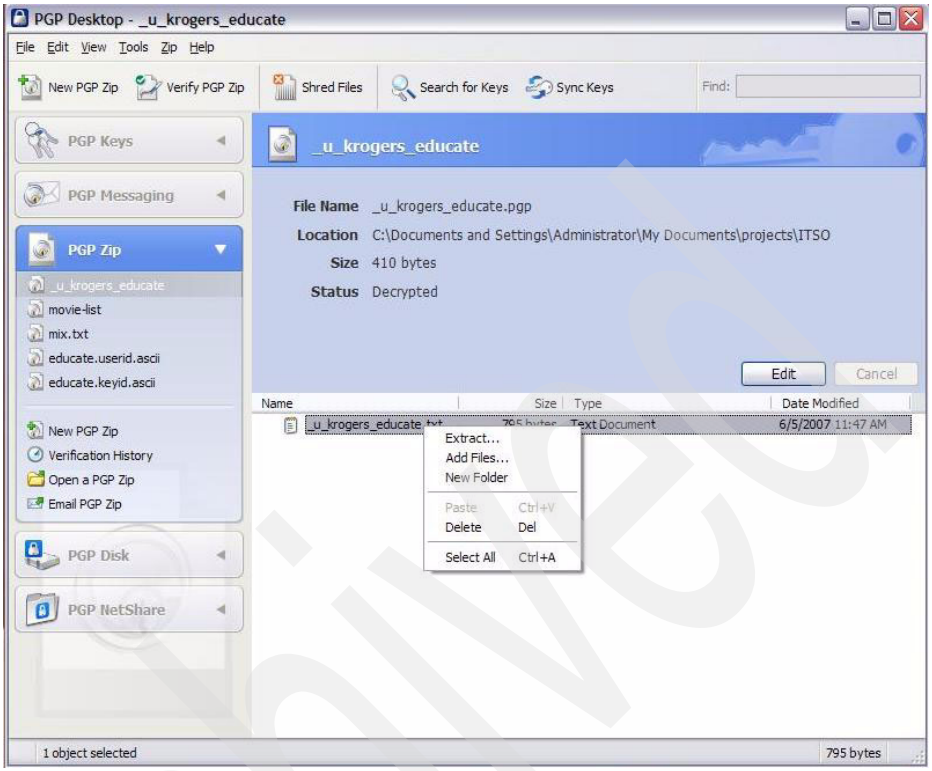


Figure 7-57 The decryption of the file is done (PGP Corporation, Reprinted by Permission)

8

Certificate Authority: X.509 and OpenPGP coexistence

This chapter discusses a possible certificate exchange infrastructure that stresses the capability of using both X.509 and OpenPGP certificate with the Encryption Facility for OpenPGP. It demonstrates the capability for OpenPGP partners to use a Certificate Authority so that received certificates can have their signatures verified.

8.1 ITSO scenario

Figure 8-1 describes the infrastructure layout that we used as an example of a hybrid environment, certificate-format wise. Building this layout and getting it to work is described in detail in the next section, where we refer to the step numbers shown in the figure.

A brief description of the overall process performed using this layout:

- User1 has a preference for dealing with X.509 certificates, using the RACF database, the PKDS, and the RACDCERT command.

  User1 personal certificate has been signed by OPGP CA and both OPGP CA and user1 X.509 certificates are transmitted to user2 as OpenPGP certificates.

  User1 Encryption Facility for z/OS makes the proper transformation from the X.509 to the OpenPGP format when exporting the certificates.

  User1 OpenPGP certificate is verified, when imported into user2 OpenPGP keyring, by the OPGP CA OpenPGP certificate.

- User2 has a preference for dealing with OpenPGP certificates and does not wish to be involved with z/OS built-in certificates facilities such as those provided by RACF or ICSF.

  User2 interacts with the Encryption Facility for z/OS to transparently generate X.509 certificates (one Certificate Authority and one personal certificate). These certificates are exported as X.509 certificates using the Java hwkeytool to User1.

  User2 personal certificate can be verified using the user2 CA certificate when imported in user1 RACF keyring.

  Still transparently, user2 uses the ICSF PKDS to securely keep the RSA private keys.

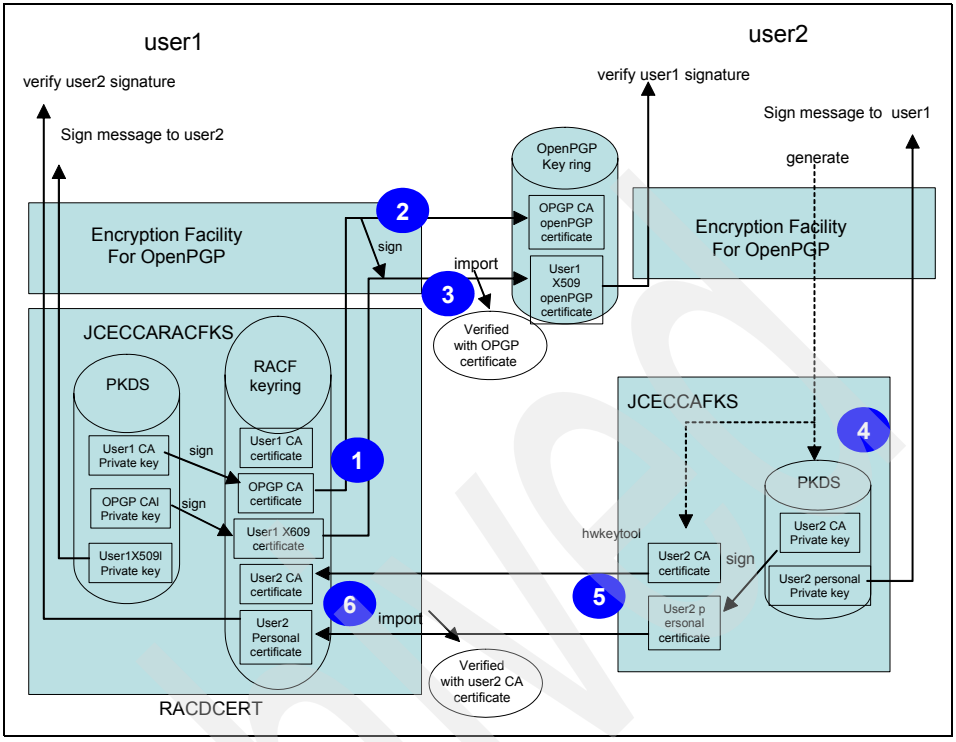


Figure 8-1 Scenario for the coexistence of X.509 and OpenPGP certificates

8.2 Establishing CAs and personal certificates

The following procedures show how to establish Certificate Authorities and personal certificates for both users.

8.2.1 User 1 key materials and certificates creation

1 User1 - Generating the RSA keys in the RACF database

The RACDCERT commands shown in Example 8-1 are used to generate two RSA key pairs in the RACF database:

- The key pair labeled ‘user 1 CA’, which is a local Certificate Authority (CA) to sign user1 X.509 certificates.
- The key pair labeled ‘OPGP CA’. The private key will be used to sign user1 OpenPGP certificate. The public key is exported to user2 as a self-signed OpenPGP certificate.

- The key pair labeled ‘user1 X509’ that belongs to user1. The private key will be used to sign the OpenPGP messages. The public key is exported to user2 as an OpenPGP certificate, also self-signed by the ‘OPGP CA’ private key.

The reference documentation for the use and syntax of the RACDCERT command is *z/OS Security Server RACF Command Language Reference*, SA22-7687.

Example 8-1 CA key pair and certificate creation

```
RACDCERT +
    GENCERT CERTAUTH +
    SUBJECTSDN(CN('user1 CA')) +
    SIZE(2048) +
    PCICC(USER1.CA) +
    WITHLABEL('user1 CA')

RACDCERT +
   GENCERT +
   SUBJECTSDN(CN('OPGP CA')) +
   SIZE(2048) +
   PCICC(OPGP.CA) +
   WITHLABEL('OPGP CA') +
SIGNWITH(CERTAUTH LABEL('user1 CA'))

RACDCERT +
  GENCERT +
  SUBJECTSDN(CN('user1 X509')) +
  SIZE(1024) +
  PCICC(USER1.X509) +
  WITHLABEL('user1 X509') +
  SIGNWITH(CERTAUTHLABEL('user1 CA'))
```

The RACDCERT commands perform the following functions:

- The first RACDCERT command generates a Certificate Authority RSA public/private key pair and stores them in the PKDS with the label as specified. The size of the key is 2048 bits (alternatively this could have specified values of 512, 768, or 1024 bits). See the note in 7.3.1, “z/OS to z/OS using RACF self-signed certificates” on page 105, for important considerations regarding the use of the ICSF or PCICC parameters with the RACDCERT GENCERT command.

  This command also generates a self-signed X.509 V3 certificate which 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 on a z890, z990, or z9 system, the “PCICC” keyword actually invokes the PCIXCC or the CEX2C coprocessor.

- The second and third RACDCERT commands generate the OPGP CA and user1’s X509 public and private key pairs and store them in the PKDS with labels as specified. These two commands also store the certificates of user1 and OPGP CA in the RACF database.

Note: We had to do this as two staged Certificate Authorities to overcome a known problem that we have today with the JCECCARACFKS and the JCERACFKS. This is explained in further detail in the next section.

User1 - Creating the RACF keyring

The sequence of the RACDCERT commands necessary to create a RACF keyring are shown in Example 8-2.

Example 8-2 Keyring creation and connection of the certificates

```
RACDCERT +
      ADDRING(USER1.KEYRING)

 RACDCERT +
     CONNECT +
     (CERTAUTH +
     LABEL('user1 CA') +
     RING(USER1.KEYRING) +
     USAGE(CERTAUTH))

 RACDCERT +
    CONNECT +
    (LABEL('OPGP CA') +
    RING(USER1.KEYRING) +
    USAGE(PERSONAL))

 RACDCERT +
    CONNECT +
    (LABEL('user1 X509') +
    RING(USER1.KEYRING) +
    USAGE(PERSONAL))
```

The RACDCERT commands perform the following functions:

1. The first RACDCERT command creates the RACF keyring with the name USER1.KEYRING.
2. The second and third RACDCERT commands connect the CA certificate and the user certificate to the just created keyring.

You can view the contents of the just created keyring by entering the RACF command shown in Example 8-3.

Example 8-3 Displaying the RACF keyring contents

```
RACDCERT LISTRING(USER1.KEYRING)

Ring:
>USER1.KEYRING<
 Certificate Label Name              Cert Owner     USAGE      DEFAULT
 ------------------------------      ------------   --------   -------
 user1 CA                            CERTAUTH       CERTAUTH     NO
 user1 X509                          ID(KAPPELE)    PERSONAL     NO
 OPGP CA                             ID(KAPPELE)    PERSONAL     NO
```

Explanation of the two staged Certificate Authorities: As of this writing, when dealing with a RACF keyring as a Java keystore:

- There must be at least one certificate flagged as CERTAUH connected to the keyring.
- However, the keystore considers that certificates flagged as “personal” can also have a private key.
- The keystore does not recognize the SITE type of certificates in RACF.

It seemed to us more consistent, in regard to what may happen in a real configuration, to accommodate these restrictions by keeping a root CA certificate actually flagged as CERTAUTH in the RACF keyring, and an intermediate CA certificate (here OPGP CA) that would be flagged as “personal” in the keyring (so that we can use its private key) and shipped as a CA certificate to the partner.

2 User1 - Exporting the certificates as OpenPGP certificates

We now invoke the export by X.509 alias command (-eA), which retrieves the designated public key from a Java keystore and wraps it in an OpenPGP certificate. The OpenPGP certificate is then stored in a file, ready to be exported, and optionally stored into a local OpenPGP keyring. We are not using a local OpenPGP keyring here.

First, we export the X.509 certificate with the label “OPGP CA”. It will be self signed only because we do not specify any signer key by omitting the option -system-CA-key-alias. The script we used is shown in Figure 8-2.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-jce-providers com.ibm.crypto.hdwrCCA.provider.IBMJCECCA \
-o /u/kappele/openpgp/exportCA.opgp \
-keystore-type JCECCARACFKS \
-keystore USER1.KEYRING \
-racf-keyring-userid KAPPELE \
-digest-name SHA_1 \
-eA "OPGP CA"
```

Figure 8-2 Shell script to create and export the OpenPGP CA certificate from the RACF keyring

The execution resulted in the interactions in the z/OS UNIX shell as shown in Figure 8-3.

```
KAPPELE:/u/kappele/openpgp: >exportpgpca.sh
CSD0002A Enter keystore password for USER1.KEYRING:
any password
CSD0777I User ID KAPPELE will be used to load RACF keyring USER1.KEYRING.
CSD0029I Exporting an OpenPGP certificate for OPGP CA...
CSD0001A Enter key password for OPGP CA:
any password
CSD0026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email add
ress(optional).
CSD0020A Real Name:
OpenPGP_CA
CSD0024A Comment:
this is the OpenPGP system CA certificate
CSD0022A Email address:
OPGPCA@itso.com
CSD0025A You specified user ID: "OpenPGP_CA <this is the OpenPGP system CA
certificate> <OPGPCA@itso.com>"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSD0027A Add another user ID? (yes/no)
no
CSD0019A For how many days should this OpenPGP certificate be valid (0 for alway
s valid):[0]
0
CSD0032A Do you want to add the exported OpenPGP Certificate to your OpenPGP key
 ring? (yes/no)
no
CSD1347I 1 OpenPGP certificate(s) were exported successfully to /u/kappele/openp
gp/exportCA.opgp.
CSD0051I Command processing has completed successfully.
```

Figure 8-3 Exporting the OpenPGP CA certificate

Then we exported the “user1 X509” certificate as an OpenPGP certificate using the script shown in Figure 8-4. Notice the specification of the private key with label “OPGP CA” in the RACF keyring as the signer of the exported OpenPGP certificates (-system-CA-key-alias).

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-jce-providers com.ibm.crypto.hdwrCCA.provider.IBMJCECCA \
-o /u/kappele/openpgp/exportX509.opgp \
-keystore-type JCECCARACFKS \
-keystore USER1.KEYRING \
-racf-keyring-userid KAPPELE \
-digest-name SHA_1 \
-system-CA-key-alias "OPGP CA" \
-eA "user1 X509"
```

Figure 8-4 Shell script to create and export user1 OpenPGP certificate from the RACF keyring

The results of the execution of this script are shown in Figure 8-5.

```
KAPPELE:/u/kappele/openpgp: >exportpgpuser1.sh
CSDO002A Enter keystore password for USER1.KEYRING:
any password
CSDO777I User ID KAPPELE will be used to load RACF keyring USER1.KEYRING.
CSDO029I Exporting an OpenPGP certificate for user1 X509...
CSDO001A Enter key password for user1 X509:
any password
CSDO026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email
address(optional).
CSDO020A Real Name:
user1
CSDO024A Comment:
this is user1's OpenPGP certificate - signed by OPGP CA
CSDO022A Email address:
user1@itso.ibm
CSDO025A You specified user ID: "user1 <this is user1's OpenPGP certificate - signed by OPGP
CA> <user1@itso.ibm>"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSDO027A Add another user ID? (yes/no)
no
CSDO019A For how many days should this OpenPGP certificate be valid (O for alway s
valid):[O]
O
CSDO001A Enter key password for OPGP CA:
any password
CSDO764I System CA OPGP CA will be used to sign generated certificate.
CSDO032A Do you want to add the exported OpenPGP Certificate to your OpenPGP key ring?
(yes/no)
no
CSD1347I 1 OpenPGP certificate(s) were exported successfully to /u/kappele/openp
gp/exportX509.opgp.
CSDO051I Command processing has completed successfully.
```

Figure 8-5 Exporting user1 OpenPGP certificate

We now have to send both certificates to user2, so that user2 can install them in his or her OpenPGP keyring. We are assuming that the transmission is done in a way that preserves the binary format of the certificate files because ASCII Armor was not specified for the export function.

3 User2 - Importing OpenPGP certificates

For this demonstration, we import the user1 OpenPGP certificate in the user2 keyring without having imported the OPGP CA OpenPGP certificate first. This is achieved by executing the shell script in Figure 8-6. We set up a configuration file for user2 where the OpenPGP keyring path and name are indicated.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/kappele/openpgp/user2config \
-i /u/kappele/openpgp/exportX509.opgp
```

Figure 8-6 Script to import user1 OpenPGP certificate.

The execution results, shown in Figure 8-7, demonstrate that the OpenPGP import function looks for certificates in the keyring that could be used to verify the certificate to import. We declined to import the OpenPGP certificate because we were warned that no key was available to verify it.

```
KAPPELE:/u/kappele/openpgp: >importuser1.sh
CSDO038I Importing OpenPGP certificate with primary RSA public key ID:
57B41B37D
7F64AEF...
CSDO708A Could not find the key with key ID 5CBCCF52C47DA3AO to verify user
ID "
user1 <this is user1's OpenPGP certificate - signed by OPGP CA>
<user1@itso.ibm>
".  Do you want to continue? (yes/no)
no
CSDO072I Error encountered while attempting to import
/u/kappele/openpgp/exportX
509.opgp.
       Error Message: CSDO052I User KAPPELE has ended the operation.
CSDO050I Command processing ended abnormally:
        CSDO074I Total number of certificates not imported successfully: 1
```

Figure 8-7 Importation of user1 OpenPGP certificate into user2 keyring with no CA certificate.

We then imported the PGP CA openPGP certificate in the user2 keyring using the script shown in Figure 8-8.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/kappele/openpgp/user2config \
-i /u/kappele/openpgp/exportCA.opgp
```

Figure 8-8 Script to import the OPGP CA certificate into user2 OpenPGP keyring.

The function was performed successfully, as shown in Figure 8-9.

```
KAPPELE:/u/kappele/openpgp: >importopgpca.sh
CSD0038I Importing OpenPGP certificate with primary RSA public key
ID: 5CBCCF52C
47DA3A0...
CSD0073I Total number of certificates imported successfully: 1
CSD0051I Command processing has completed successfully.
KAPPELE:/u/kappele/openpgp: >
```

Figure 8-9 Importing the OPGP CA certificate into user2 keyring

A good practice is for user2 to check the authenticity of user1's fingerprint in the self-signed certificate. Users can see the fingerprint of the certificate, since it is in their keyring, by invoking the -pK command and specifying the key ID 5C BC CF 52 C4 7D A3 A0.

We then can try again to import the user1 OpenPGP certificate, which now completes successfully as shown in Figure 8-10.

```
KAPPELE:/u/kappele/openpgp: >importuser1.sh
CSD0038I Importing OpenPGP certificate with primary RSA public key
ID: 57B41B37D
7F64AEF...
CSD0073I Total number of certificates imported successfully: 1
CSD0051I Command processing has completed successfully.
```

Figure 8-10 Importing user1 OpenPGP certificate with the CA certificate already in the keyring

We can now list the openPGP certificates in the user2 keyring using the list per user ID command (-pP) without specifying any user ID. This yields the display shown in Figure 8-11.

```
KAPPELE:/u/kappele/openpgp: >listuser2.sh
CSD1353I Displaying OpenPGP certificate whose user IDs match:
"OpenPGP_CA <this
is the OpenPGP system CA certificate> <OPGPCA@itso.com>"
primary: RSA 2,048  key ID: 5CBCCF52C47DA3A0  fingerprint:
33E7BFA1042321313B1B0
A1C5CBCCF52C47DA3A0  created: 5/6/07
uid:            "OpenPGP_CA <this is the OpenPGP system CA
certificate> <OPGPCA@
itso.com>"
trust:   10 Trusted

CSD1353I Displaying OpenPGP certificate whose user IDs match:
"kappele <no comme
nt> <kappele@itso.com>"
primary: RSA 1,024  key ID: A3CB0C6AF07ABE0F  fingerprint:
FCFF68125FBEE74C88981
63EA3CB0C6AF07ABE0F  created: 3/17/07
uid:         "kappele <no comment> <kappele@itso.com>"
trust:   10 Trusted

CSD1353I Displaying OpenPGP certificate whose user IDs match: "user1
<this is us
er1's OpenPGP certificate - signed by OPGP CA> <user1@itso.ibm>"
primary: RSA 1,024  key ID: 57B41B37D7F64AEF  fingerprint:
745538A2DA10A75329F19
F5C57B41B37D7F64AEF  created: 5/6/07
uid:            "user1 <this is user1's OpenPGP certificate - signed
by OPGP CA>
 <user1@itso.ibm>"
trust:   10 Trusted

CSD0051I Command processing has completed successfully.
```

Figure 8-11 Certificates in user2 OpenPGP keyring

8.2.2 User2 key materials and certificates creation

4 User2 - Generating RSA key pairs in the ICSF PKDS

User2 is to interact with user1 using X.509 certificates. We are therefore going to generate two RSA key pairs:

- A Certificate Authority key pair, labeled “USER2.CA”, to be used to sign user2’s X.509 certificate. The public key will be exported to user1.
- A user2 personal key pair, labeled USER2.KEY. The public key will be exported to user1 as well.

Generation of the CA key pair

We used the generate function (-g) of the Encryption Facility for OpenPGP to generate the USER2.CA RSA key pair. The shell script that we used is shown in Figure 8-12.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/kappele/openpgp/user2config \
-jce-providers com.ibm.crypto.hdwrCCA.provider.IBMJCECCA \
-keystore USER2.KS  \
-keystore-type JCECCAKS \
-keystore-password password \
-g
```

Figure 8-12 User2 CA key pair generation

Executing this script results in the interactions shown in Figure 8-13.

```
KAPPELE:/u/kappele/openpgp: >    generate_ca.sh
CSDO046A You specified a keystore type JCECCAKS. Generating RSA key. Do you want to continue? (yes/no)
yes
CSDO018A Enter alias for RSA key:
USER2.CA
CSDO004A Enter key size:[1024]
2048
CSDO044A For how many days should the X.509 certificate be valid (Maximum value 9999):[365]
9999
CSDO010A What is your first and last name? [Unknown]
user2 CA
CSDO011A What is the name of your organizational unit? [Unknown]
itso
CSDO012A What is the name of your organization? [Unknown]
ibm
CSDO013A What is the name of your city or locality?  [Unknown]
pok
CSDO014A What is the name of your state or province?  [Unknown]
ny
CSDO015A What is the two-letter country code for this unit? [Unknown]
us
CSDO016A Is <CN=user2 CA,OU=itso,O=ibm,L=pok,ST=ny,C=us> correct? (yes/no)
yes
CSDO001A Enter key password for USER2.CA:
password
CSDO017A Confirm password:
password
CSD1363I Generating RSA key...
CSDO048A Please select what type of hardware key you want:[1]
        (1) PKDS
        (2) CLEAR
1
CSD1331I An acceptable hash algorithm name was not found. Using: SHA_1
CSD1364I This key will be self-signed with alias: USER2.CA
CSDO040A Importing key into the OpenPGP key ring...
CSDO026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email address(optional).
CSDO020A Real Name:
user2 CA
CSDO024A Comment:
This is the CA RSA key pair that is going to sign user2 personal certificate
CSDO022A Email address:
user2_CA@itso.com
CSDO025A You specified user ID: "user2 CA <This is the CA RSA key pair that is going to sign user2 personal
certificate> <user2_CA@itso.com>"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSDO027A Add another user ID? (yes/no)
no
CSD1331I An acceptable hash algorithm name was not found. Using: SHA_1
CSDO019A For how many days should this OpenPGP certificate be valid (0 for always valid):[9,999]
CSDO038I Importing OpenPGP certificate with primary RSA public key ID: 58C5083427152351...
CSDO039I Total number processed successfully: 1
CSD1372I X.509 certificate summary for key alias USER2.CA
        Key ID: 58C5083427152351
        Key Type: RSA
        Key Size: 2,048
        Keyring User ID(s): "[user2 CA <This is the CA RSA key pair that is going to sign user2 personal certificate>
<user2_CA@itso.com>]"
CSDO051I Command processing has completed successfully.
```

Figure 8-13 Generation of user2 CA key pair

We are done now with the generation of the user2 CA RSA key pair. The keys are kept in the PKDS, with an X.509 certificate available in the JCECCAKS. We can then display this certificate with the -pA command, as shown in Figure 8-14.

```
CSD1352I Displaying certificate with alias: USER2.CA
dn: CN=user2 CA, OU=itso, O=ibm, L=pok, ST=ny, C=us
primary: RSA 1,024  key ID: B6DE2E05C54B338C  fingerprint:
F7A73C505A918EFD31516
7CDB6DE2E05C54B338C created: 5/9/07 [expired: 5/8/08]

CSD0051I Command processing has completed successfully.
```

Figure 8-14 The user2 CA certificate

Generation of user2 personal RSA key pair

The personal RSA key pair of user2 is generated using the -g command of the Encryption Facility for OpenPGP, as shown in Figure 8-15.

```
KAPPELE:/u/kappele/openpgp: >generate_rsa.sh
CSDO046A You specified a keystore type JCECCAKS. Generating RSA key. Do you want to continue? (yes/no)
yes
CSDO018A Enter alias for RSA key:
USER2.KEY
CSDO004A Enter key size:[1024]
CSDO044A For how many days should the X.509 certificate be valid (Maximum value 9999):[365]
CSDO010A What is your first and last name? [Unknown]
user2 key
CSDO011A What is the name of your organizational unit? [Unknown]
itso
CSDO012A What is the name of your organization? [Unknown]
ibm
CSDO013A What is the name of your city or locality?  [Unknown]
pok
CSDO014A What is the name of your state or province?  [Unknown]
ny
CSDO015A What is the two-letter country code for this unit? [Unknown]
us
CSDO016A Is <CN=user2 key,OU=itso,O=ibm,L=pok,ST=ny,C=us> correct? (yes/no)
yes
CSDO001A Enter key password for USER2.KEY:
password
CSDO017A Confirm password:
password
CSD1363I Generating RSA key...
CSDO048A Please select what type of hardware key you want:[1]
        (1) PKDS
        (2) CLEAR
1
CSD1331I An acceptable hash algorithm name was not found. Using: SHA_1
CSDO001A Enter key password for USER2.CA:
password
CSDO033I This key will be signed with system CA alias: USER2.CA
CSDO040A Importing key into the OpenPGP key ring...
CSDO026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email address(optional).
CSDO020A Real Name:
user2 key
CSDO024A Comment:
this is user2 personal key pair and certificate. The public key is signed by USER2.CA
CSDO022A Email address:
user2_key@itso.com
CSDO025A You specified user ID: "user2 key <this is user2 personal key pair and certificate. The public key is signed
by USER2.CA> <user2_key@itso.com>"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSDO027A Add another user ID? (yes/no)
no
CSD1331I An acceptable hash algorithm name was not found. Using: SHA_1
CSDO019A For how many days should this OpenPGP certificate be valid (0 for always valid):[365]
CSDO764I System CA USER2.CA will be used to sign generated certificate.
CSDO038I Importing OpenPGP certificate with primary RSA public key ID: 576AB6093E12CBA9...
CSDO039I Total number processed successfully: 1
CSD1372I X.509 certificate summary for key alias USER2.KEY
        Key ID: 576AB6093E12CBA9
        Key Type: RSA
        Key Size: 1,024
        Keyring User ID(s): "[user2 key <this is user2 personal key pair and certificate. The public key is signed by
USER2.CA> <user2_ca@itso.com>]"
CSDO051I Command processing has completed successfully.
```

Figure 8-15 Generating user2 personal RSA key pair and certificate

The user2 personal key pair and certificate generation has been performed successfully.

5 User2 - Exporting user2 certificates

According to the scenario we described earlier, the certificate prepared with the label USER2.CA in the JCECCAKS has to be exported to user1. We used the export function of hwkeytool so that we can keep handling this certificate as an X.509 certificate.

The hwkeytool command that we used is shown in Figure 8-16.

```
hwkeytool -export -alias USER2.CA -file user2.ca \
-keystore USER2.KS \
-storetype JCECCAKS \
-storepass password \
-provider com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
```

Figure 8-16 Example of hwkeytool command

The user2 CA certificate is now stored in the z/OS UNIX file user2.ca in binary. We assume that we transmit the file, preserving its binary contents, to user1 so that user2 certificate can be added to user1's RACF keyring. We need to put the certificate in a z/OS data set so that we can use the RACDCERT ADD command. This can be done with the TSO OGET command as follows:

```
oget 'kappele/openpgp/user2.ca' USER2.CA binary
```

We will return later to the importation of the user2 CA certificate into user1's RACF keyring.

As we did for the USER2.CA certificate, we use hwkeytool to export the user2 personal X.509 certificate and the OGET command to get the certificate into a dataset, ready to be added to user1 keyring.

User1 now has two certificates to add to the keyring, contained in:

- KAPPELE.USER2.CA
- KAPPELE.USER2.KEY

6 Importing user2 CA and personal certificates

To demonstrate that RACF verifies a certificate added to the database against the existing CA certificate, we try first to add the user2 personal certificate prior to adding the user2 CA certificate. This yields the result shown in Figure 8-17.

```
racdcert add('kappele.user2.key') withlabel('user2 key') trust

Certificate Authority not defined to RACF.  Certificate added with
NOTRUST status
```

Figure 8-17 Trying to add user2 personal certificate before user2 CA

However, if the user2 CA certificate has been installed in the RACF database then there is no warning message when adding the user2 personal certificate, meaning that the user2 personal certificate signature has been verified.

Once the certificates have been added to the RACF database they must be connected to user1's RACF keyring. This is done with the commands:

```
RACDCERT connect (certauth label('user2 ca') ring(USER1.KEYRING)
usage(certauth))
RACDCERT connect (label('user2 key') ring(USER1.KEYRING)
usage(certauth))
```

We can now check the contents of user1's RACF keyring as shown in Figure 8-18.

```
RACDCERT listring(USER1.KEYRING)
Digital ring information for user KAPPELE:

   Ring:
>USER1.KEYRING<
 Certificate Label Name              Cert Owner     USAGE
DEFAULT
 ------------------------------   ------------   -------- -------
user1 CA                          CERTAUTH       CERTAUTH     NO
user1 X509                        ID(KAPPELE)    PERSONAL     YES
OPGP CA                           ID(KAPPELE)    PERSONAL     NO
user2 ca                          CERTAUTH       CERTAUTH     NO
user2 key                         ID(KAPPELE)    CERTAUTH     NO
```

Figure 8-18 Checking the contents of USER1.KEYRING

8.3 User1 sends user2 a signed message

User1 signs the message

The message will be signed by the private key of user1, which is kept in the PKDS and pointed at by the certificate label “user1 X509” connected to the RACF keyring USER1.KEYRING.

The shell script with the proper Encryption Facility for OpenPGP commands is shown in Figure 8-19, and the execution results are shown in Figure 8-20. In this example, encryption has not been used.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-jce-providers com.ibm.crypto.hdwrCCA.provider.IBMJCECCA \
-o /u/kappele/openpgp/signed_data \
-keystore-type JCECCARACFKS \
-keystore USER1.KEYRING \
-racf-keyring-userid KAPPELE \
-signers-key-alias "user1 X509" \
-signers-key-password password \
-keystore-password password \
-digest-name SHA_1 \
-s /u/kappele/openpgp/data_to_sign
```

Figure 8-19 SIgnature script

```
KAPPELE:/u/kappele/openpgp: >rsasignkring.sh
CSDO777I User ID KAPPELE will be used to load RACF keyring
USER1.KEYRING.
CSDOO51I Command processing has completed successfully.
```

Figure 8-20 Results of the signature command

User2 verifies the message

We assume that the signed_data file has been transferred to user2. User2 verifies the signature using user1’s public key, which is kept in user2’s OpenPGP keyring. The user1 certificate is kept with the name “kappele” in the OpenPGP keyring.

The verify command is in the script verifypgpring.sh shown in Figure 8-21. The option -v does not unpackage the data. If you need to perform unpackaging, you need to use option -d.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-homedir /u/kappele/openpgp/user2config \
-v /u/kappele/openpgp/signed_data
```

Figure 8-21 The verifypgpring.sh script

Attention: It is required that the -v command be the last command specified. If not, as for instance -rP being after -v, Encryption Facility for z/OS R2 issues an error message stating that -v requires a final argument.

The results of the signature verification are shown in Figure 8-22.

```
KAPPELE:/u/kappele/openpgp: >verifypgpring.sh
CSDO779I Signature by key ID 57B41B37D7F64AEF is valid.
CSDO051I Command processing has completed successfully.
```

Figure 8-22 Verify signature execution results

8.3.1 User 2 sends user1 a signed message

User2 signs the message

User2 signs the message with the private key alias USER2.KEY in the JCECCAKS USER2.KS, which is actually a private key kept in the PKDS.

We used the shell script rsasignsks.sh shown in Example 8-23.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-jce-providers com.ibm.crypto.hdwrCCA.provider.IBMJCECCA \
-o /u/kappele/openpgp/signed_data \
-keystore-type JCECCAKS \
-keystore USER2.KS \
-signers-key-alias USER2.KEY \
-signers-key-password password \
-keystore-password password \
-digest-name SHA_1 \
-s /u/kappele/openpgp/data_to_sign
```

Figure 8-23 rsasignks.sh

The results of the execution of rsasignks.sh are shown in Figure 8-24.

```
KAPPELE:/u/kappele/openpgp: >rsasignks.sh
CSDO051I Command processing has completed successfully.
```

Figure 8-24 user2 signs with rsasignks.sh

8.3.2 User1 verifies the message

We assume that user2 sent the file signed_data to user1.

User1 is to verify the message using user2 X.509 certificate in RACF, with a label “user2 key” connected to the USER1.KEYRING RACF keyring.

We used the shell script verifykring.sh shown in Figure 8-25. The results of the execution are shown in Figure 8-26.

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar \
-jce-providers com.ibm.crypto.hdwrCCA.provider.IBMJCECCA \
-keystore-type JCECCARACFKS \
-keystore USER1.KEYRING \
-racf-keyring-userid KAPPELE \
-v /u/kappele/openpgp/signed_data
```

Figure 8-25 verifykring.sh

```
KAPPELE:/u/kappele/openpgp: >verifykring.sh
CSDO002A Enter keystore password for USER1.KEYRING:
password
CSDO777I User ID KAPPELE will be used to load RACF keyring
USER1.KEYRING.
CSDO779I Signature by key ID 576AB6093E12CBA9 is valid.
CSDO051I Command processing has completed successfully.
```

Figure 8-26 Execution of verifykring.sh

9

Sample code for an OpenPGP Certificate Server

This chapter focuses on two items of concern organizations have when dealing with PGP keys: certificate authenticity and certificate management. This chapter discusses some practical aspects of these concerns and provides a sample application that yields some of the functionality discussed.

For this chapter, the terms *OpenPGP certificate* and *OpenPGP key* are used interchangeably. Although these terms do not always represent precisely the same thing, they are close enough for this discussion. We are concerned here with the public portion of an asymmetric key, most often an RSA key, as well as the information that helps to identify the key material, the usage preferences, key owner, and other information found in certificates.

9.1 OpenPGP key authenticity

As discussed previously in this book, the OpenPGP paradigm has a peculiar model of trust referred to as a “web of trust.” In this model, individuals can assign a trust rating to other user’s OpenPGP keys. Through some networking algorithms, or through manual inspection, individuals or organizations can determine a level of trust for a particular certificate. This model of trust works very well for individuals – for example, if I want to share information with someone I have not directly met, but somebody whom I know vouches for this person and therefore I have some amount of confidence that signed information I am receiving does not come from a malicious third party. In the corporate or governmental space, this notion of personal relationship-based and somehow “weighted” trust may make security practitioners uneasy; they would rather deal with certificates that are known in a very binary way to either be fully trusted or, on the contrary, be not trusted, based on an organization-wide agreed upon strict trust policy.

When it comes to distributing keys between parties or entities, a challenge mechanism is often used to acquire some level of confidence that the senders of keys are actually who they claim to be. In today’s environment, the most common challenge mechanism is that of a user ID and password. In this model, the receiving organization will create the userid and password and distribute it to the sending organization. The sending organization will then use this information to authenticate to the service and send their certificate. Assuming the ID and password have been disseminated in a prudent manner, the receiving organization will have a reasonably high degree of confidence that the certificate submitted is really from the sending organization.

When the sending organization goes to submit their key, they would also like some assurances that they are sending their key to the correct organization, and not a spoofed web site. To cover this, the server that is receiving the OpenPGP key should be protected by SSL or TLS and a certificate signed by a well known X.509 Certificate Authority. This X.509 certificate should be signed by a certificate authority that both the sending and receiving organizations agree on. In this situation, both parties will have a reasonably high enough degree of confidence that the key sent has not been intercepted or altered in transit.

These are two common practices for determining authenticity of both the sending and receiving organizations in today’s business environment. This chapter shows how this can be implemented with a sample application that can manage OpenPGP keys.

The level of trust of such a client-server kind of communication could be enhanced using, for instance, SSL/TLS client authentication. However, many

organizations may still ask for a user password after the client workstation has been authenticated with a client X.509 certificate. For the sake of simplicity here we consider that SSL/TLS client authentication is not deemed to be a mandatory practice.

9.2 OpenPGP key distribution and management

Organizations that must manage more than a handful of OpenPGP keys should consider implementing a Public Key Server that is controlled in house. Public key servers have been around for quite some time, with the X.509 PKIX variant being the prevailing one today. The OpenPGP HTTP Keyserver Protocol (HKP) defines a standard for interacting with a key server and can be found at:

`http://ietfreport.isoc.org/idref/draft-shaw-openpgp-hkp/`

This draft also briefly discusses some of the same issues that are covered in this chapter, including the notion that key users need to be sure that they are in fact using an authentic and valid key.

The HKP describes the basic operations that need to be supported by an implementation of a key server, but recognizes that a more robust set of features may be desired. The remainder of this chapter covers these basic features as implemented in the sample application and notes where additional capabilities may be worthwhile.

9.3 Our scenario for using OpenPGP key server

We came to the high-level assumption that users of OpenPGP in an organization, or in a network of partnering organizations, would need the capability to retrieve the OpenPGP public keys, therefore the certificates, of individuals they have to deal with in such cases as sending an encrypted message of verifying a signed message.

Furthermore, the certificate retrieval capability should also be augmented with simple, but trusted, ways for members of the organizations to make their OpenPGP certificates available to other members. Finally, because OpenPGP certificate revocation might prove to be a complex process to deal with in large environments, a simple way of “disabling” certificates should also be provided.

All these functions would be built around a certificate repository as shown in Figure 9-1 and Figure 9-2. In Figure 9-1 Stephanie is expecting to receive an OpenPGP encrypted message with the symmetric key protected by her

OpenPGP key. She therefore submits her OpenPGP certificate to the key server so that it can be stored and later retrieved by an authorized person such as Ben. When the time comes for Ben to send an encrypted message to Stephanie, he can search for Stephanie's certificate in the key server and use Stephanie's public key to protect the encryption key.

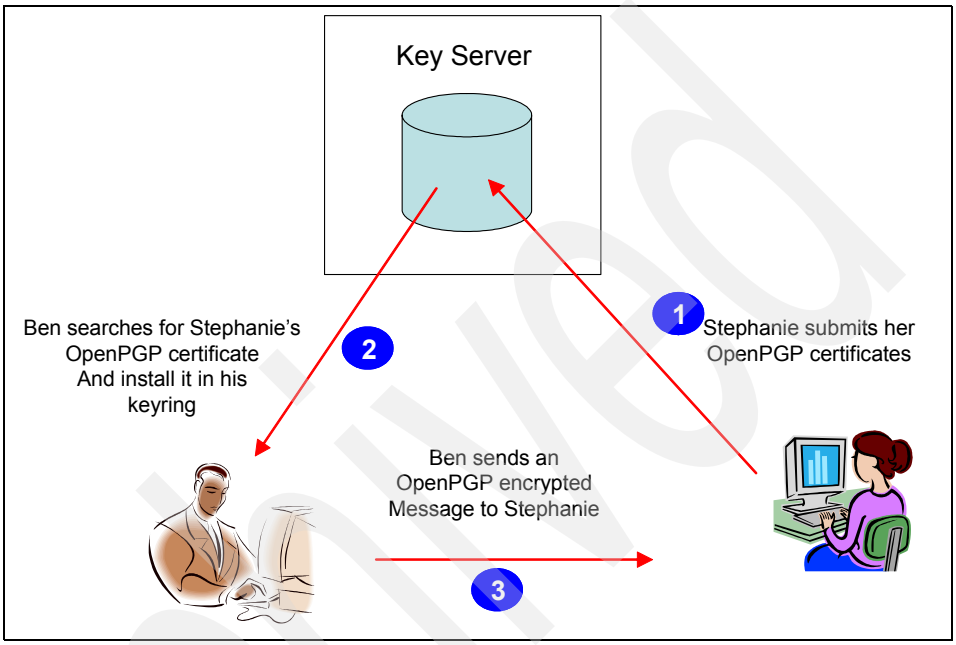


Figure 9-1 Stephanie expects to receive OpenPGP encrypted messages from Ben

In Figure 9-2 Stephanie is expecting to receive signed messages from Ben. Ben has submitted his OpenPGP certificate to the key server and Stephanie can now search Ben's certificate in order to be able to verify Ben's signature.

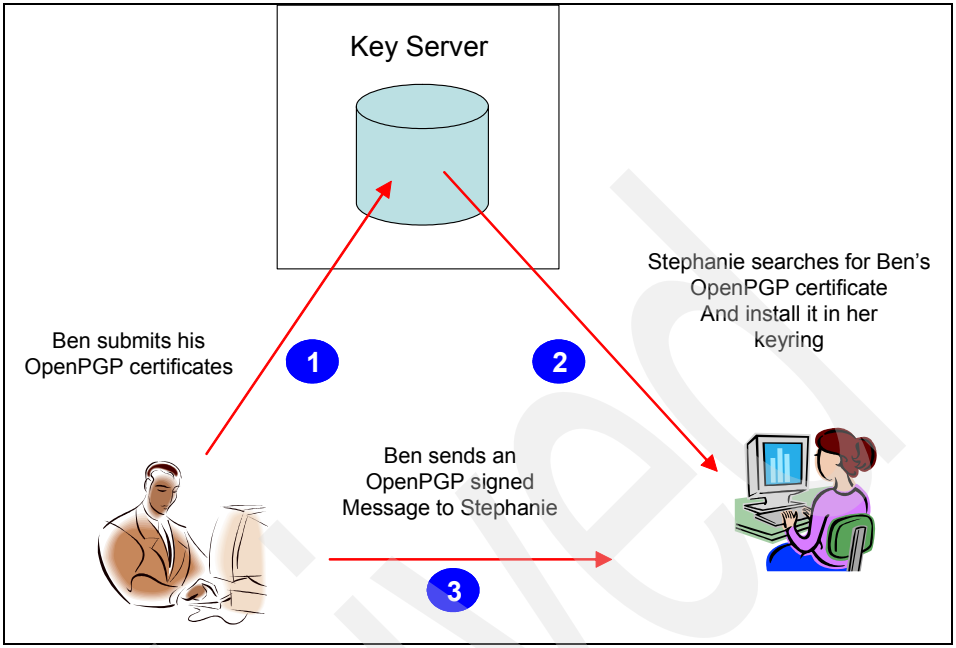


Figure 9-2 Stephanie expects to receive signed messages from Ben

9.4 OpenPGP Public Key Server sample application design

For the remainder of this chapter, the sample application will be referred to as the IBM ITSO OpenPGP Public Key Server, or IIPKS.

It is based on the OpenPGP HTTP Keyserver Protocol (HKP) and has some additional built-in features to address some of the concerns organizations may have about certificate authenticity when dealing with OpenPGP certificates. Refer to Appendix D, "Additional material" on page 267 for instruction on how to download the sample code.

When considering alternative solutions for this endeavour, the fact that the Encryption Facility OpenPGP code is written in Java leads to the decision that the key server should be written in Java as well, so that part of the Encryption Facility OpenPGP code could be re-used for quick implementation of the required OpenPGP format handling functions.

Attention: This sample application *requires* that the Encryption Facility OpenPGP code be available to the WebSphere runtime environment. This sample application *will not* function correctly or in any meaningful way without the Encryption Facility OpenPGP jar file.

Since the HKP protocol specifies sending and retrieving OpenPGP keys over the HTTP protocol, and since the Encryption Facility OpenPGP support is written in Java, it made sense to look at the various Java-based web containers that can run in the z/OS environment. When considering the containers that could be used, WebSphere 6.1 on z/OS emerged as the winner.

There were a number of reasons why WebSphere 6.1 was chosen as the application container. First, WebSphere 6.1 uses Java 5.0, which is a requirement of the Encryption Facility OpenPGP feature. Previous versions of WebSphere do not use Java 5.0 and are not candidates for this application. Second, WebSphere 6.1 has a feature called “federated repository support,” which helps to make authentication to multiple unique sources transparent to the application.

Note: Although the WebSphere 6.1 runtime has a less recent level of Java than what is required by the Encryption Facility OpenPGP feature, the code paths exercised with this sample application do not seem to need the newer version of Java.

One of the design problems encountered with this application is the notion of authenticating an internal user, organization-wise, versus an external user, and where that authentication information should be kept. Many organizations today have a centralized user repository, like an LDAP directory, that can be used by many applications. Since this OpenPGP application should be accessible to all or nearly all people within the organization, this type of centralized repository is something that would be great to use. However, this only addresses one portion of the user community requirement. The other portion of the user community will be from outside the organization, such as from a business partner, and in most cases should never appear in the organization’s centralized user repository. Most organizations would have a fit if an application required them to update their central user registry and add external users to it so that these users could get access to this service. The goal of this design challenge is to allow both “internal” and “external” users access to the same service without having to make special application changes.

To address this issue, we took advantage of a new feature in WebSphere 6.1 called “Federated Repositories.” This feature allows the WebSphere administrator to configure multiple user registries that are searched transparently

to the user or application. For instance, internal users are authenticated against the organization's centralized LDAP server and "external" users are authenticated against a secondary LDAP server.

Figure 9-3 shows the overall architecture of the sample OpenPGP Public Key Server application.

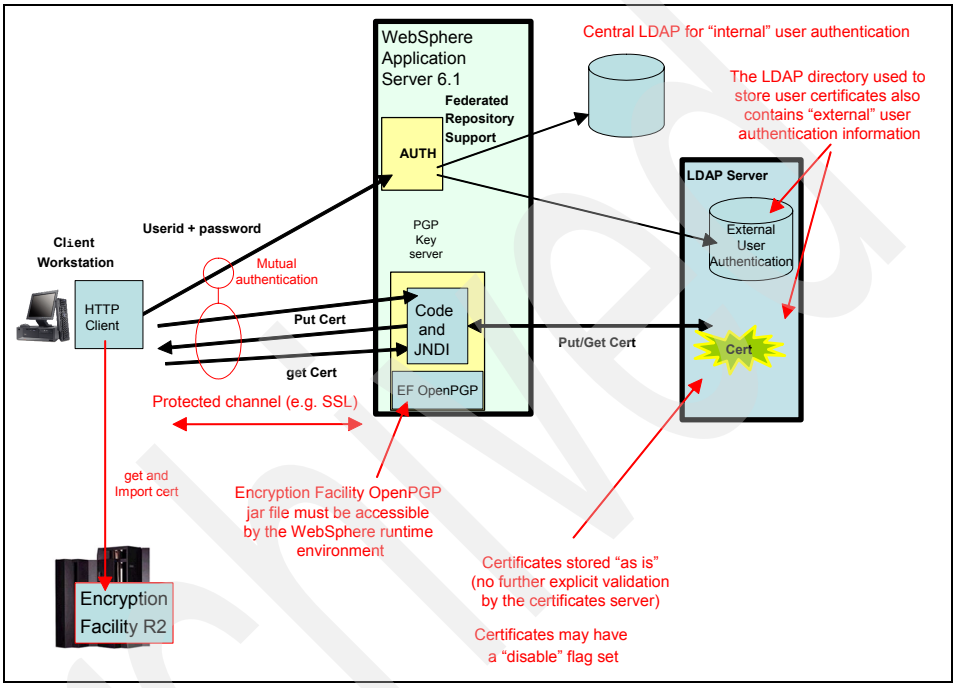


Figure 9-3 OpenPGP Public Key Server architecture

9.4.1 User registries

As mentioned previously, there are two broad classes of users that have an interest in using a key server: users who are part of the organization that owns or manages the key server, and users who are outside this organization. In this context, an organization could be a subset of a company, or it could be an entire company. For this proof of concept, the "owning" organization is comprised of all IBM employees, and users outside the organization might be IBM business partners. Another assumption that was made is that many organizations already have some kind of centralized user authentication mechanism like an LDAP server. Within IBM, we call this "Bluepages." This directory is something that applications can use to authenticate users, but typically, the applications do not modify this directory. This is why the federated user repository support is so critical. It allowed us to plug in an existing repository so that any valid user within

the company can access this service, and we can create another stand-alone repository for “external” users such as business partners.

In this solution, the LDAP server that holds user entries also holds all of the OpenPGP keys. For this proof of concept, this LDAP is running on z/OS in the same LPAR as the z/OS WebSphere instance. Although this muddies the waters slightly between a container that holds key information and a container that holds user registry information, it provides a solution with fewer moving parts than having a separate directory for external users and keys.

9.4.2 Thwarting identity theft

One common practice for verifying the identity of a server or application is to use an X.509 certificate that has been signed by a well known Certificate Authority. This certificate provides two functions. The first function is that it lets the users of the server know that a third party has vouched for the identity of the certificate, and in some sense, the identity of the organization and even the server.

The other important thing this does is to help provide some of the basic building blocks of an SSL session between the user and the server. The public key associated with the X.509 certificate is used in the SSL handshake to exchange the underlying encryption key. This certificate is critical in proving the identity and providing the cryptographic security of this service.

9.5 Installing and configuring the IIPKS

The following sections assume that a functional z/OS WebSphere 6.1 environment is already available and that the Encryption Facility OpenPGP code is available on the same system. For this application to work, the Encryption Facility OpenPGP code must be available on the system hosting the application. It also assumes that an LDAP server exists that can be used to store OpenPGP key information and external user identity information.

The download referred to in Appendix D, “Additional material” on page 267 contains two types of information: the LDAP schema .ldif files used to help store the OpenPGP key information in LDAP, and the application file. The application is delivered as an .ear file and is installable into WebSphere with very few steps.

9.5.1 Load the LDAP schema

There are actually two packages of .ldif files for this project. One package is the original version of the files, which do not load cleanly into the z/OS LDAP server.

The other package has been modified so that the files will load into the z/OS LDAP server. The primary difference between the files is the removal of the X-ORIGIN attributes in the z/OS version.

There are three ldif files that need to be loaded into LDAP:

pgp-remte-prefs.zos.schema

pgp-recon.zos.schema

pgp-keyserver.zos.schema

These are sample `ldapmodify` commands that can be used to add the schema into the z/OS LDAP:

```
ldapmodify -D cn=admin -w secret -f pgp-remte-prefs.zos.schema
ldapmodify -D cn=admin -w secret -f pgp-recon.zos.schema
ldapmodify -D cn=admin -w secret -f pgp-keyserver.zos.schema
```

Ldif schema files are provided courtesy of PGP Corporation. Reprinted by permission.

After successfully loading the pgp schema files, the base LDAP objects must be loaded into the directory. Locate the file IBM.OpenPGP.Public.Key.Server.ldif and open it for editing. Replace all instances of <SUFFIX> with the suffix that will be used to store the OpenPGP keys and the local user information. After modifying this file, use the following command to update the LDAP directory with this ldif file:

```
ldapmodify -D cn=admin -w secret -f IBM.OpenPGP.Public.Key.Server.ldif
```

9.5.2 Install the EAR file

After downloading the .ear file to your workstation, log onto the WebSphere administration console, select **Applications** → **Enterprise Applications**, and select the **Install** option. This displays the Preparing for the application installation panel (Figure 9-4).

Select the .ear file from your computer and select **Next** (Figure 9-4).

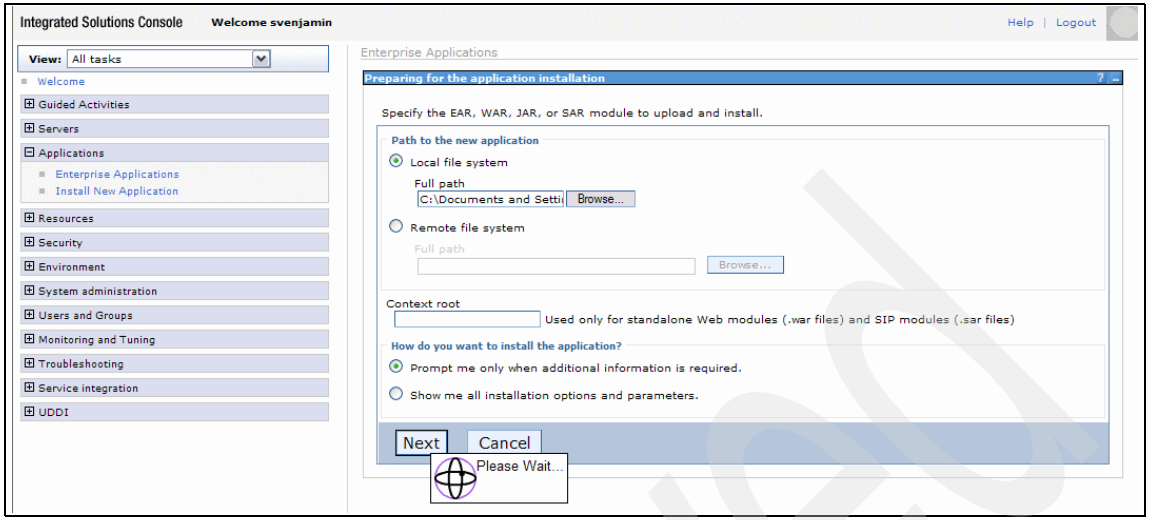


Figure 9-4 Installing the sample code - ear

Select any options appropriate for your environment and select **Next** (Figure 9-5).

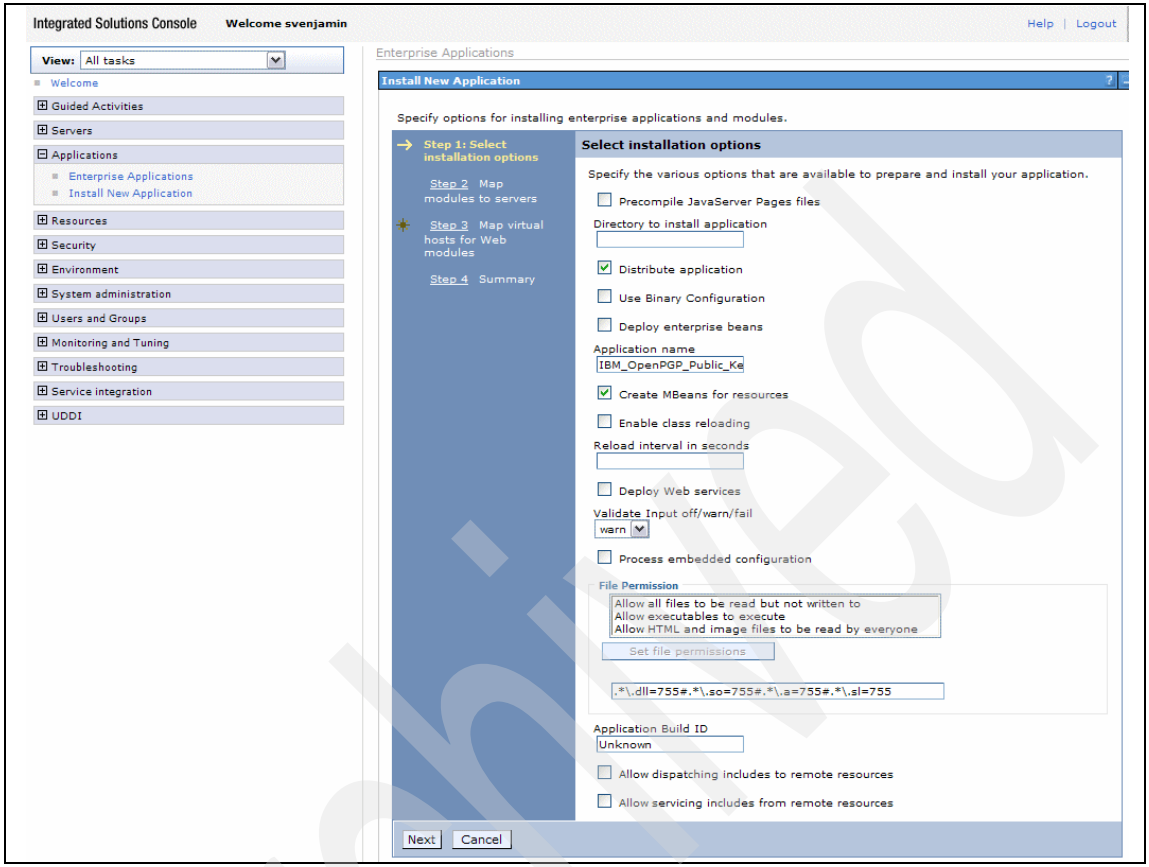


Figure 9-5 Installation options selection

Select your Clusters and Servers and click **Next** after selecting the appropriate information (Figure 9-6).

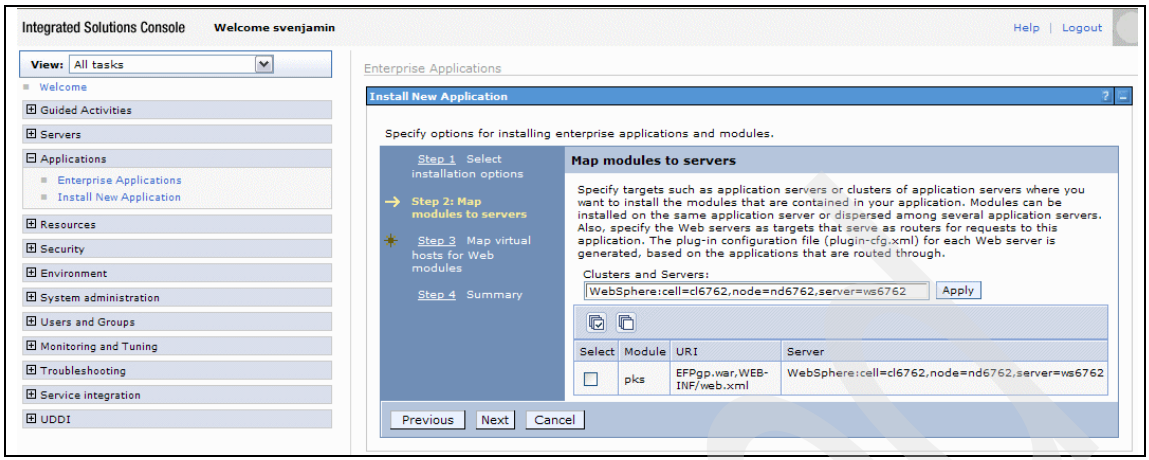


Figure 9-6 Mapping to servers

Make sure the correct virtual host mapping is done and select **Next** (Figure 9-7).

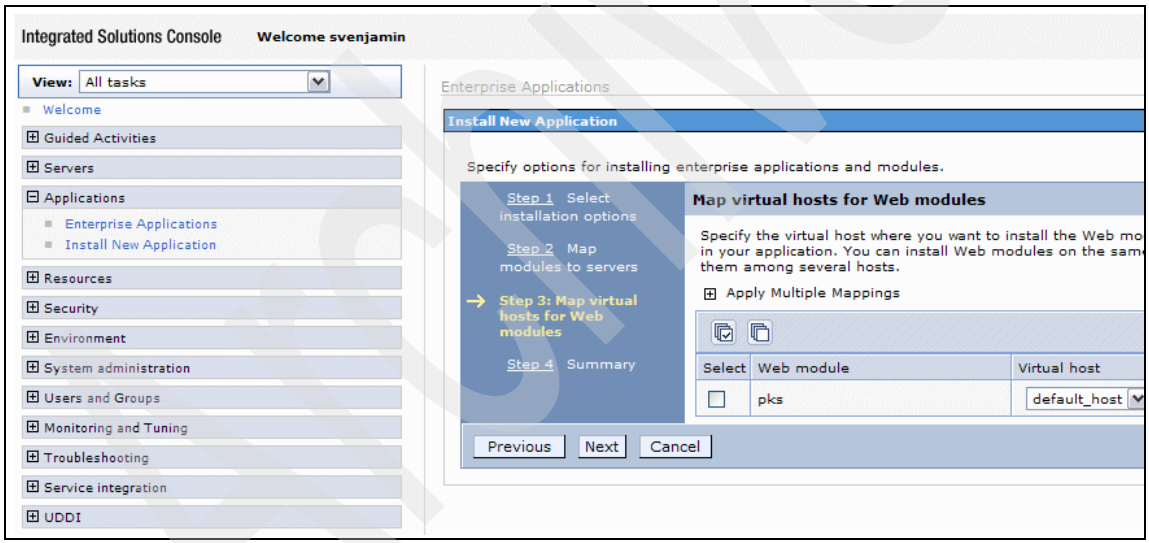


Figure 9-7 Virtual host definition

Confirm all of the selections you made (Figure 9-8) and click **Finish**.

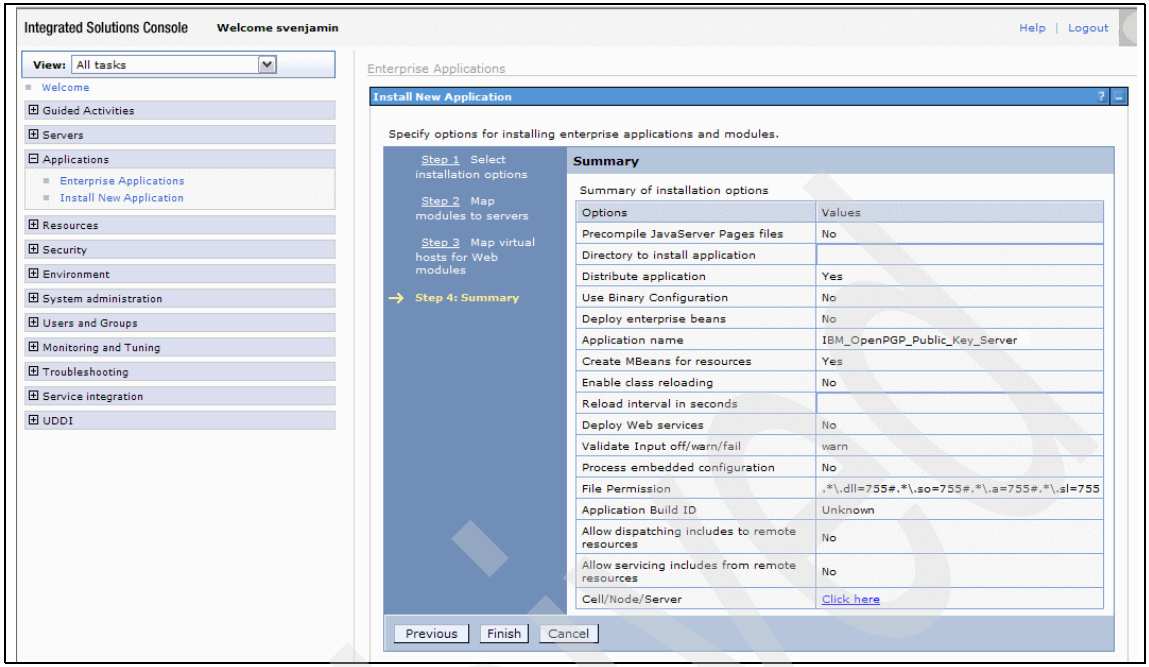


Figure 9-8 Completing the installation

You should now see informational messages about the install. At the end, you should see a message indicating success. Click **Save** to save the changes to the master configuration file.

At this point, the application has been successfully installed and can now be started. Select the application and click **Start** to start the application. The application name is IBM_OpenPGP_Public_Key_Server.

A green arrow is displayed when the application starts successfully, as shown in Figure 9-9.

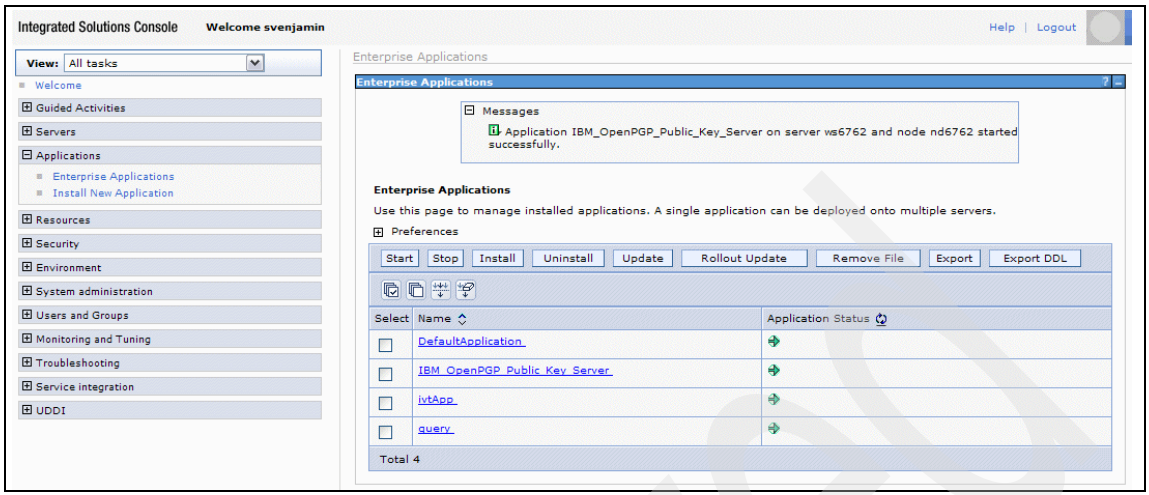


Figure 9-9 Application started successfully

At this point, the application has been successfully installed. We can now proceed to configure the WebSphere Application Server environment.

9.5.3 Configure WebSphere Application Server

This section describes the modifications we need to make to the WebSphere Application Server environment.

The first step is to configure two WebSphere Application Server environment variables. To configure environment variables, follow the links **Environment** → **WebSphere Variables**. One variable that must be changed is LIBPATH and the second is special to this application and should be called OPENPGP_HOME.

The LIBPATH environment variable should have a scope of Node and must contain the following directory:

```
/usr/lib/java_runtime
```

This is shown in Figure 9-10.

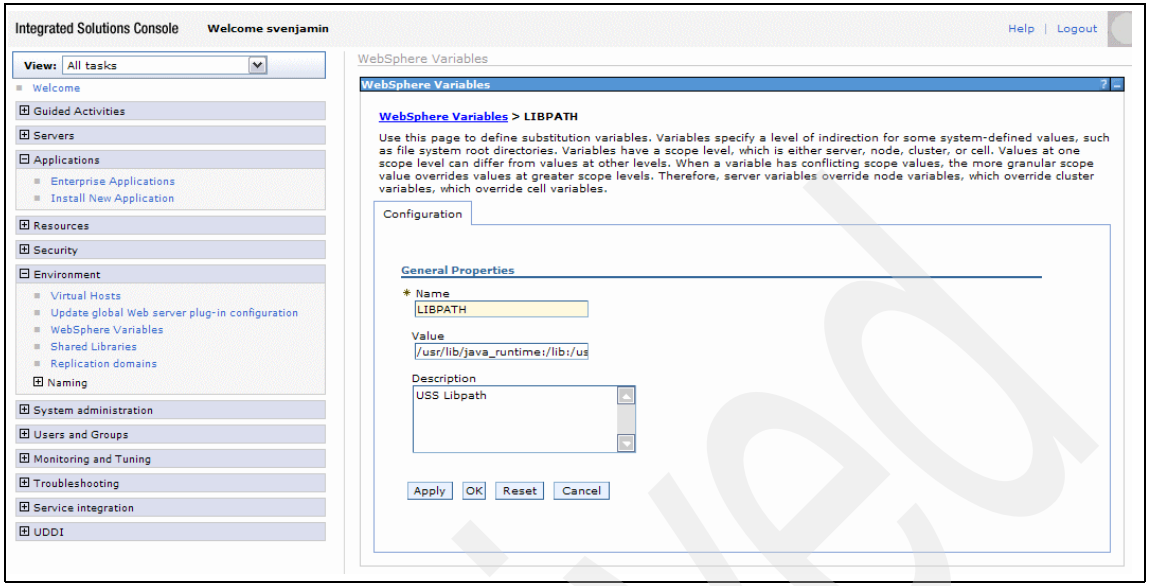


Figure 9-10 LIBPATH variable set up

The next variable must be created. It should be called OPENPGP_HOME and should have a scope of Node, as shown on Figure 9-11. From **Environment** → **WebSphere Variables**, select Node=<*yournode*>l from the Scope drop-down list, and click **New**.

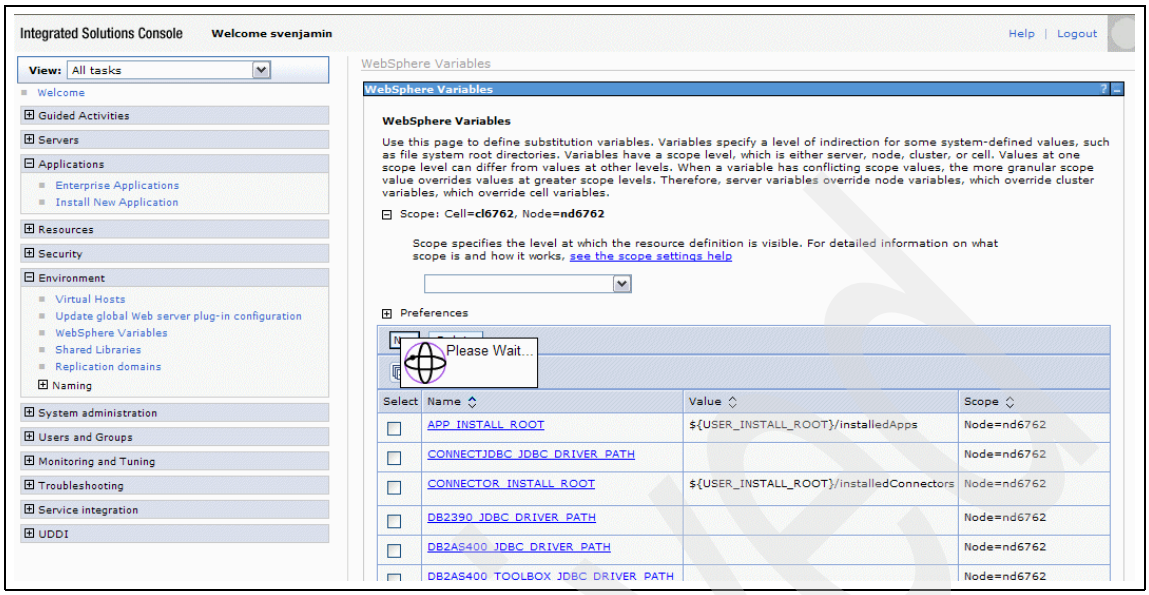


Figure 9-11 Variables definition

On the next panel (Figure 9-12) enter the name of the environment variable as `OPENPGP_HOME`; the value for this will be a directory that will contain pkipgpserver.properties.

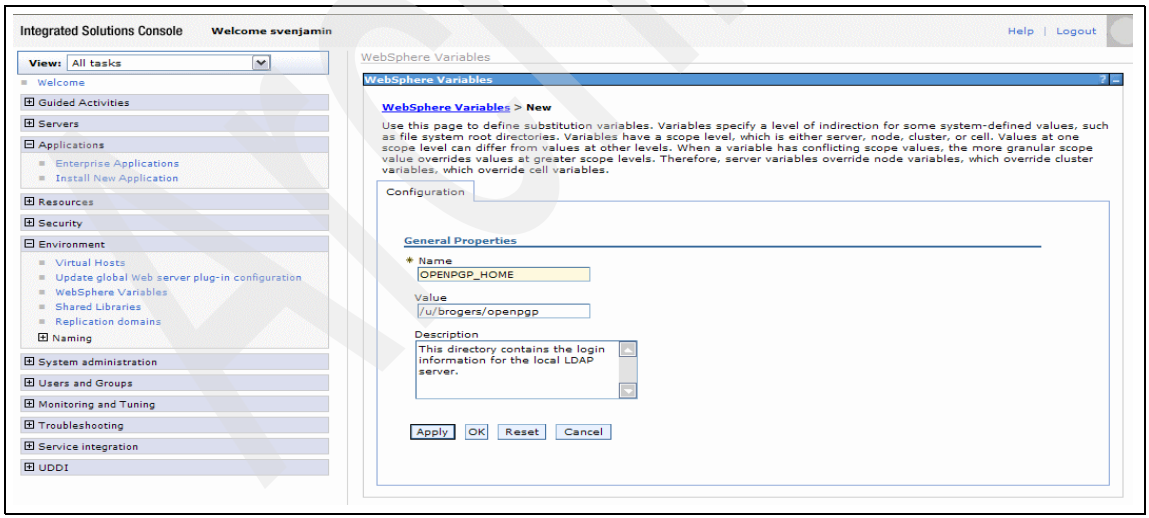


Figure 9-12 Adding WebSphere variable

Configure the Servant region's classpath. Do this by selecting **Servers** → **Application Servers** → ***<Your Server Name>*** (Figure 9-13).

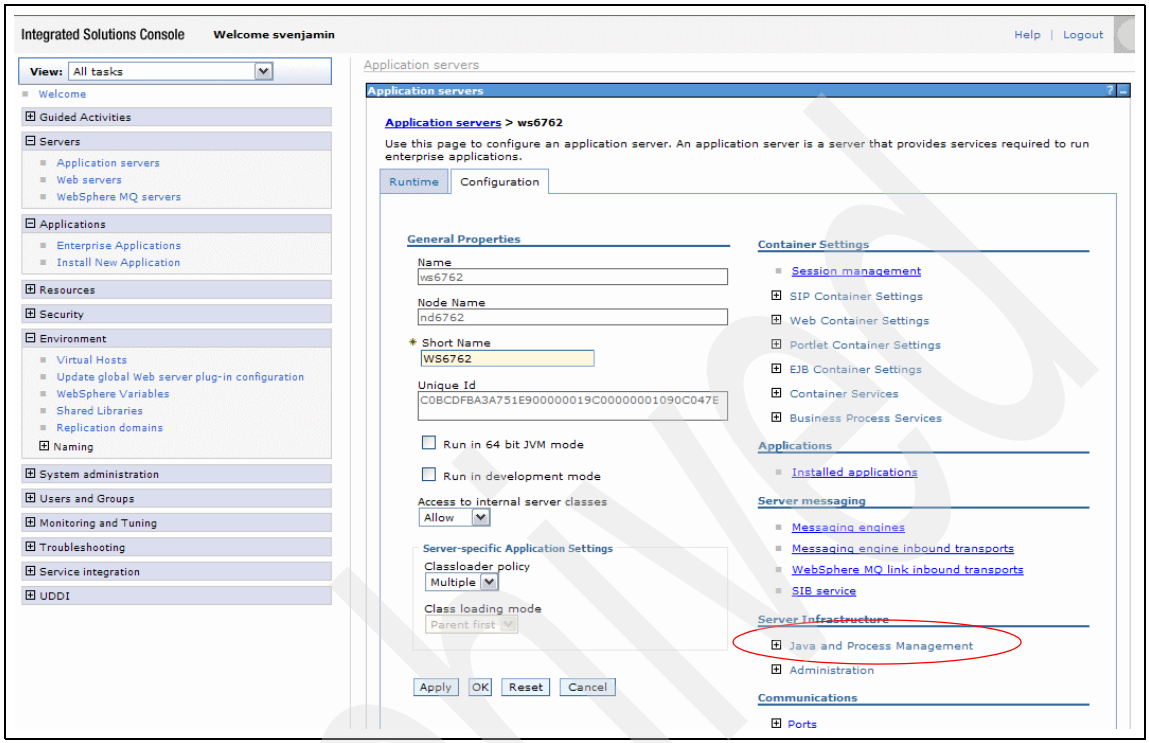


Figure 9-13 CLASSPATH definition

Expand the Java and Process Management section and select **Process Definition** → **Servant** (Figure 9-14).

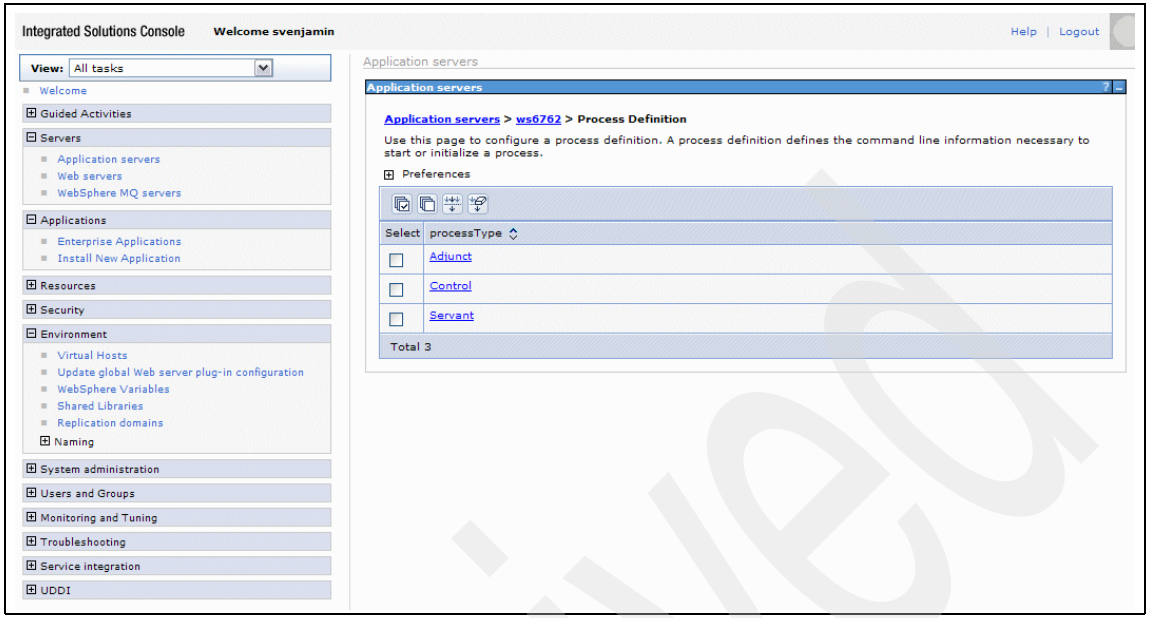


Figure 9-14 Variable definition

Select **Java Virtual Machine**, which brings you to the panel shown in Figure 9-15. In the box for the classpath, enter the full path to the CSDEncryptionFacility.jar. In this example, the full path is:

```
/usr/lpp/encryptionfacility/CSDEncryptionFacility.jar
```

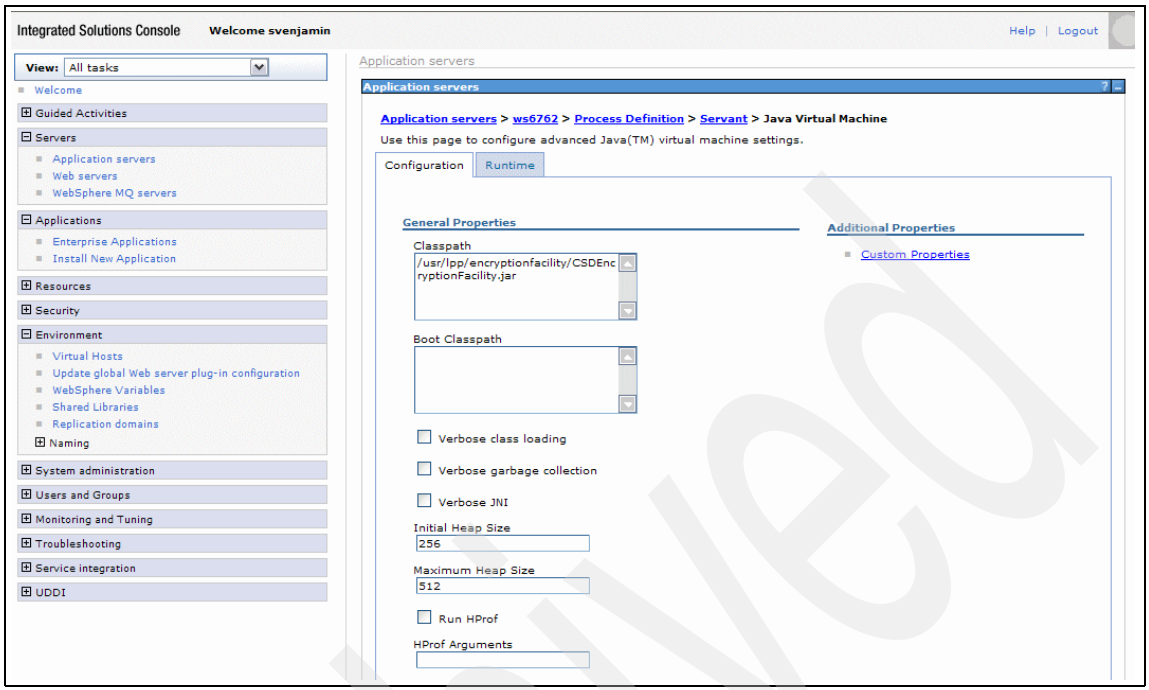


Figure 9-15 CLASSPATH definition

Click **OK**, then save the master configuration file. At this point, the WebSphere server can be restarted so that these changes take effect.

9.5.4 Configure security and the user registries

Navigate to **Security** → **Secure administration, applications, and infrastructure**. Select “Enable administrative security” and “Enable application security” if they are not already selected. Click **Apply** (Figure 9-16), then save the configuration file.

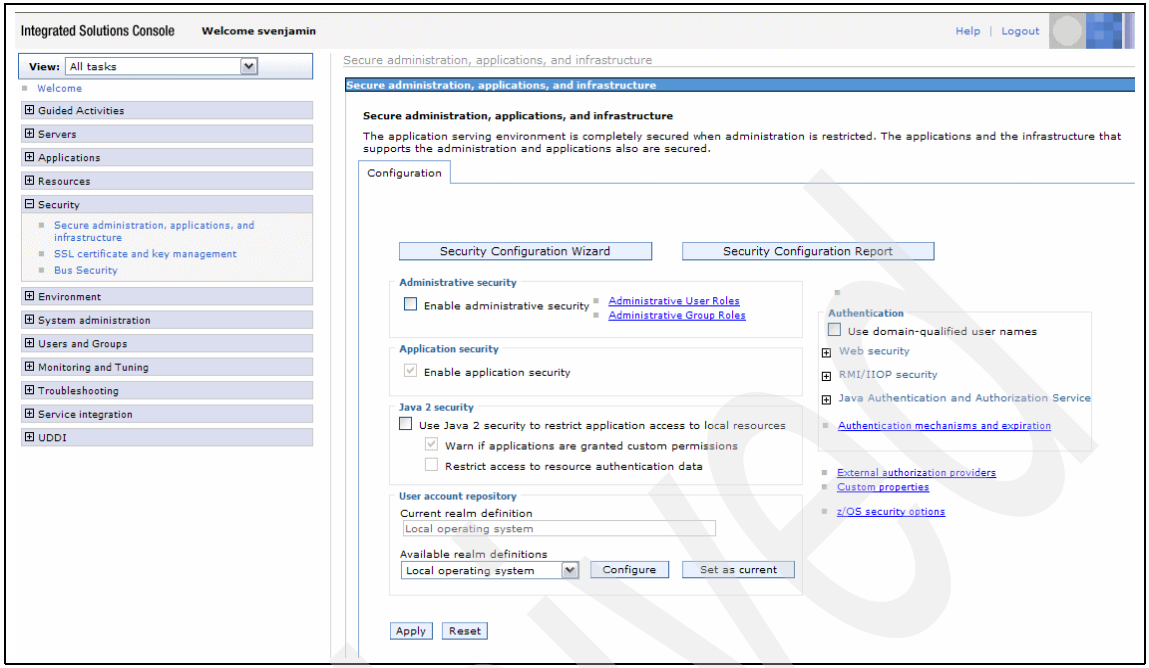


Figure 9-16 Configuring security

In the “User account repository” section, select **Federated repositories** from the drop-down list and click **Configure**. This will take you to the Federated repositories configuration page shown in Figure 9-17.

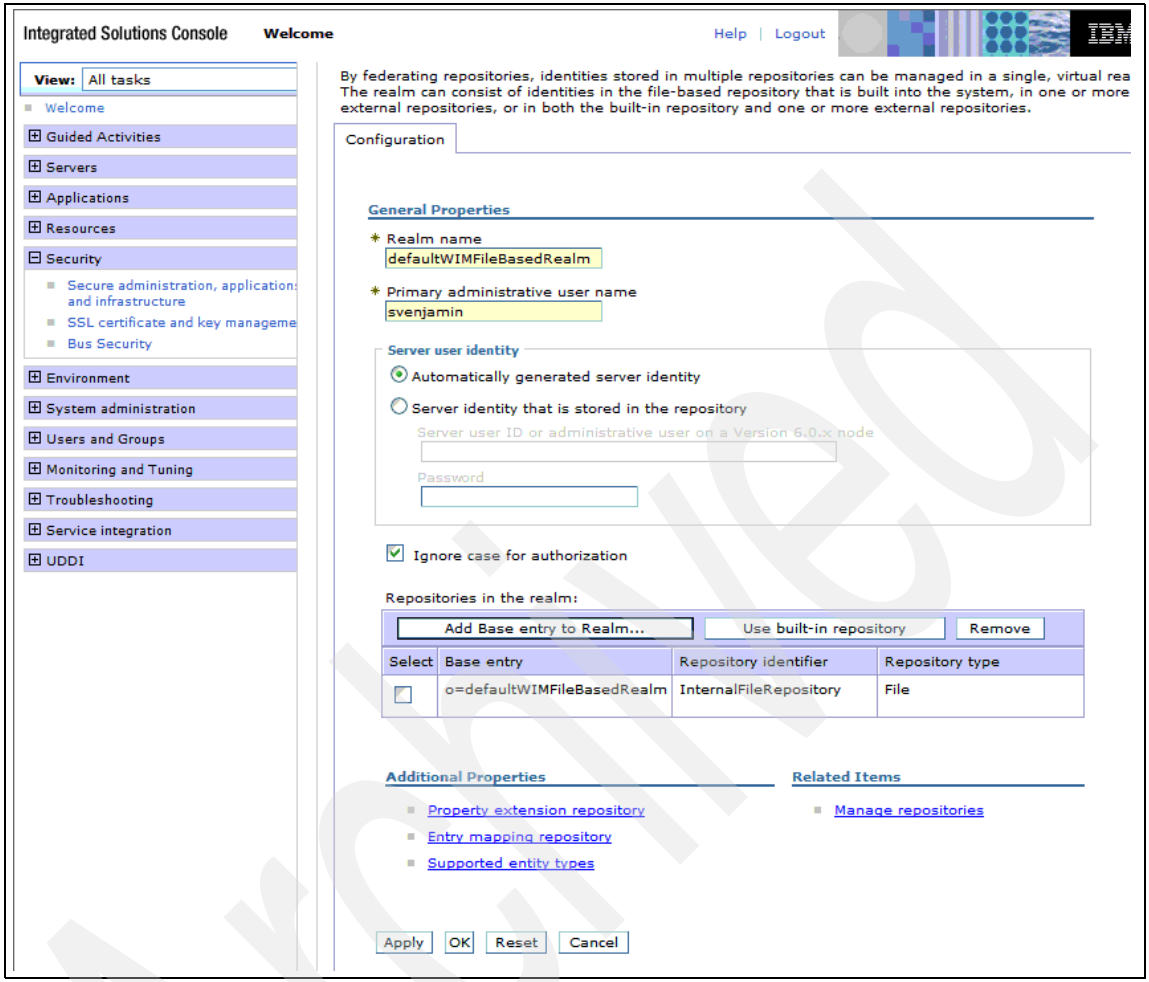


Figure 9-17 Configuring security

On this panel, click **Add Base entry to Realm**. The Repository reference page shown in Figure 9-18 is displayed. Click the **Add Repository** button.

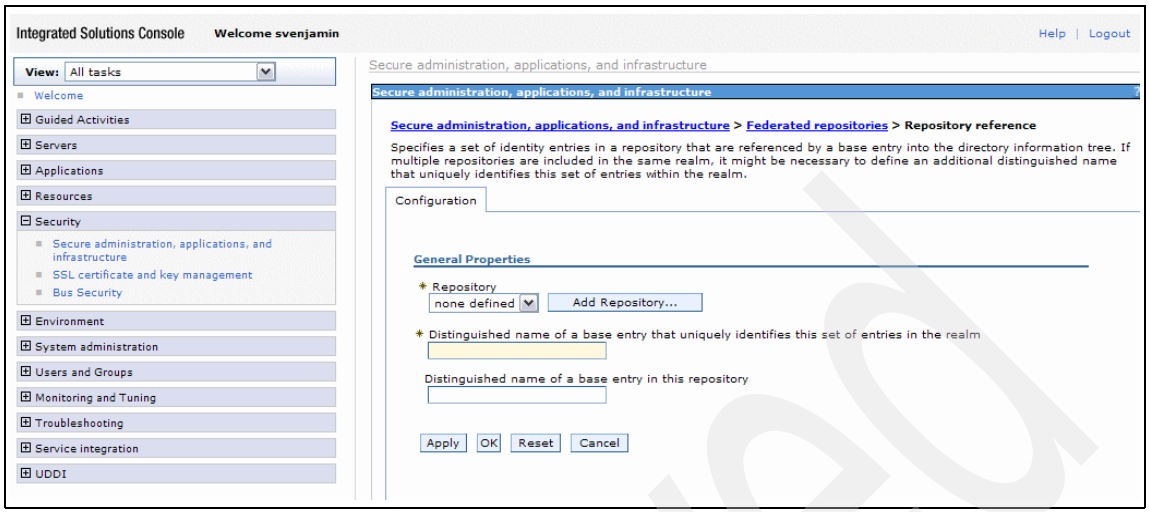


Figure 9-18 Add Repository

On the Federated repositories configuration page, fill in the information specific to the corporate LDAP server that will serve as your main user repository, as shown in Figure 9-19.

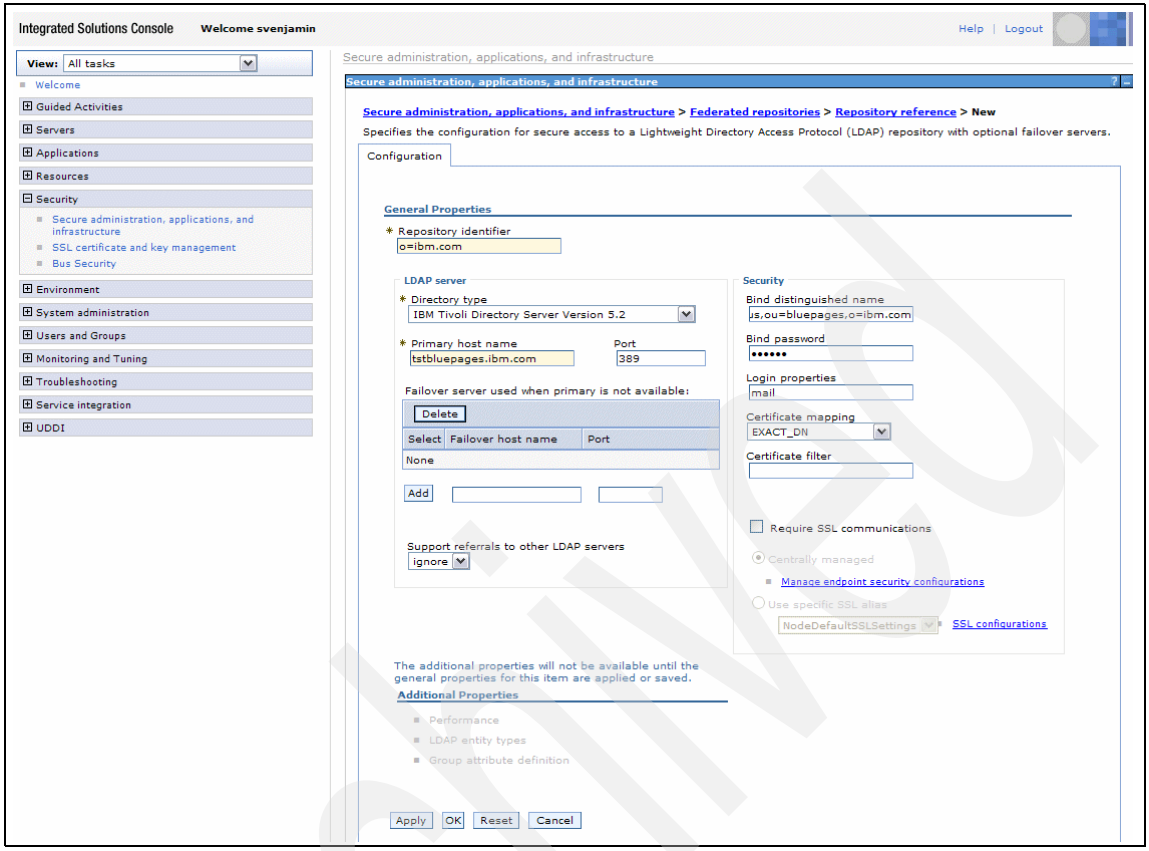


Figure 9-19 Add Repository

Click **Apply** and **Save**. You will be returned to the Repository reference configuration panel as shown in Figure 9-20. In this panel, fill in the suffix of the LDAP server in the “Distinguished name of a base entity in the repository” field. This entry will be the relative root from which all other searches are done for this repository. So, for instance, if you configure o=ibm.com as your root, in your LDAP entity types PersonAccount configuration section if you wanted to only look under the dn of ou=bluepages,o=ibm.com you would only need to specify ou=bluepages for your search base.

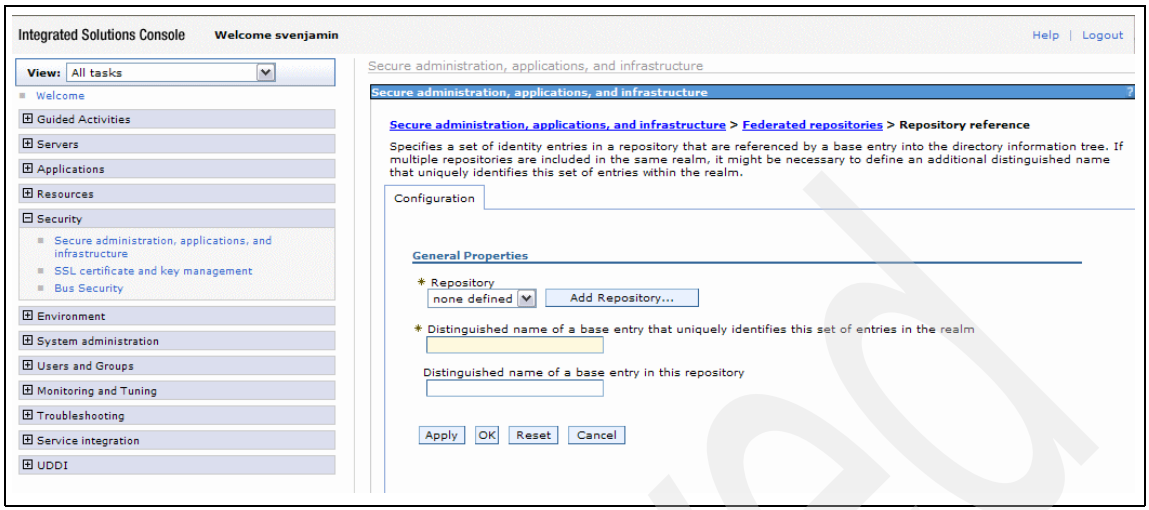


Figure 9-20 Repository reference

Click **Apply**, then **Save**. You will be redirected to the Federated repositories configuration page for the repository you are working on (Figure 9-21). Note that now the Additional Properties links at the bottom of the page are active. Click the **Repository Identifier** field for the base entry just created (o=ibm.com), then click the **LDAP entity types** link to configure the Group and PersonAccount LDAP entities.

Select the **PersonAccount** link as shown in Figure 9-21.

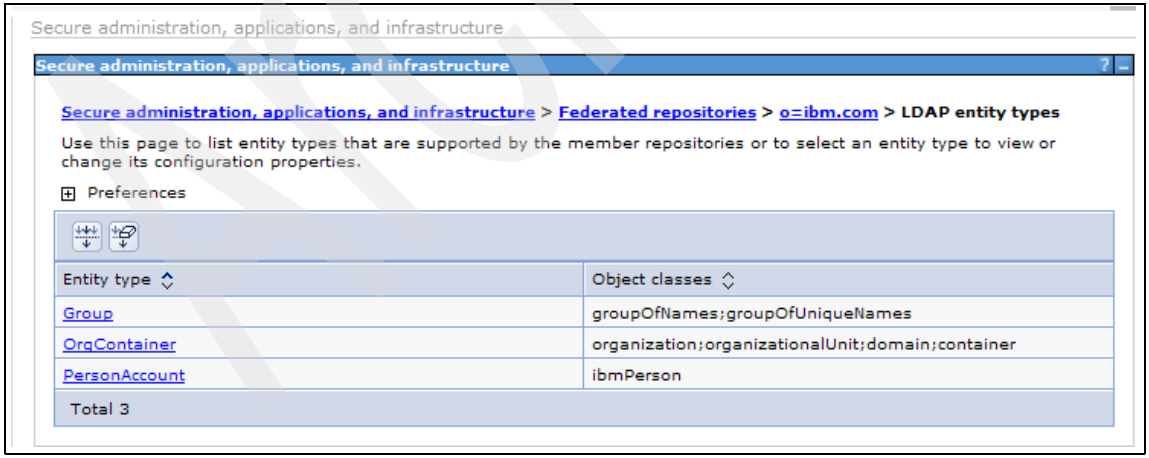


Figure 9-21 LDAP entity types definition

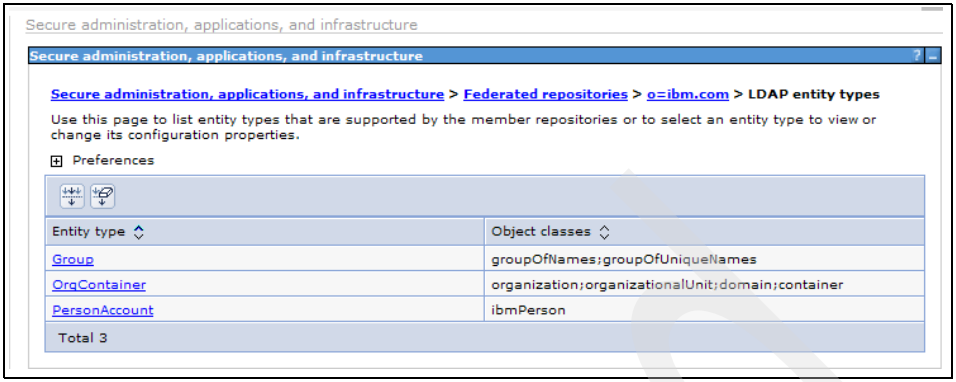


Figure 9-22 Adding repository

For this LDAP directory, you only care about the objectclass of ibmPerson, so change the objectclass to ibmPerson and use the filter (objectclass=ibmPerson) in the Search Filter field to limit your result set, as shown in Figure 9-23. Note that with a federated repository, you cannot use substitution variables like cn=%v, as you can in the stand-alone LDAP repositories. This is because you must specify what attribute users will be identified by when they log in, as shown in Figure 9-19.

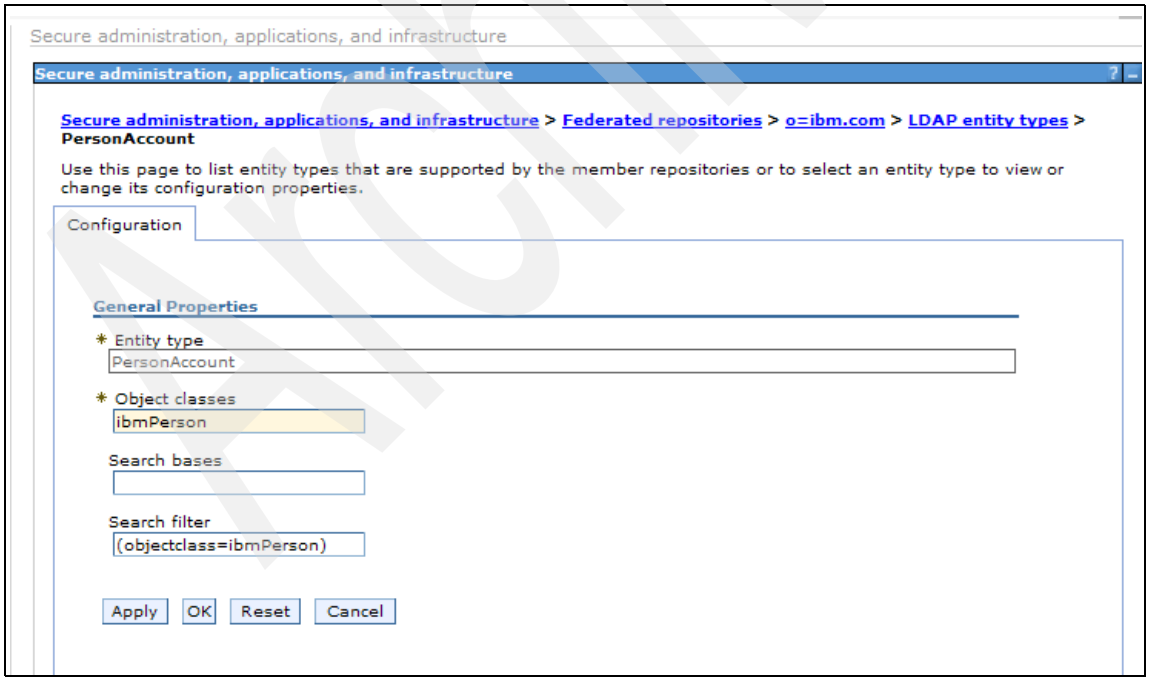


Figure 9-23 PersonAccount definition

Click **Apply**, then **Save** to save this configuration, then go to the Group entity types page (Figure 9-24). Here you specify both groupOfNames and groupOfUniqueNames as the objectclasses you care about when determining group membership. Use the following search filter:

```
(|(objectclass=groupOfNames)(objectclass=groupOfUniqueNames))
```

Do not use any substitution variables in this section because they will cause group gathering to fail.

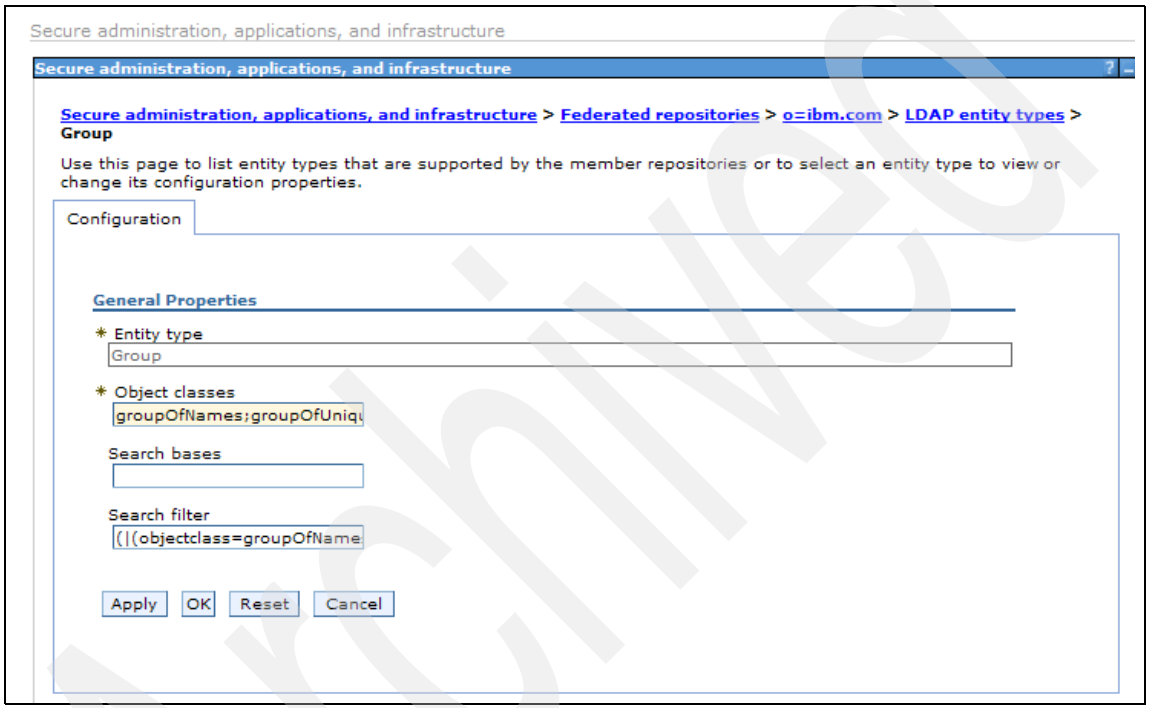


Figure 9-24 Group entity definition

The result of the configuration is shown in Figure 9-25.

Figure 9-25 Configuration results

After Applying and Saving all of these pages, you are returned to the Federated Repositories configuration page. Click **Apply** and **Save** one more time to be brought back to the main Security configuration page.

Restart the WebSphere server to activate these changes. To verify the changes have been made, after the server starts up, log back into the administration console and go to **Users and Groups** → **Manage Users** or **Manage Groups**. You should be able to search for users and groups in the repository that you have just configured.

Once the primary registry has been configured, follow a similar procedure to configure the LDAP registry that holds both external users and the OpenPGP keys. The users in this directory are added as inetOrgPerson objectclasses instead of ibmPerson objectclasses. Also, it is your choice as to what attribute should be used for determining authentication. For this proof of concept, we used the e-mail address for both the Bluepages configuration and the external user's configuration since this should give a unique login ID for all users across both repositories.

9.5.5 Configuring user roles

There is really only one user role that needs to be configured for this application: the PKSAdmin role. This role grants access to add external users to the LDAP directory through this application. There is no restriction on who can be in the PKSAdmin group; however, it would make sense to have this be people from within the organization and not any external users.

As we prepared this book, the team debated whether we should include roles for External users as well as internal users. However, there is no functional difference between what either group has access to based on the deployment of this application, so we decided to leave it as only one role of PKSAdmin. There is one group already defined to the PKSAdmin role called PKSAdmin. There is also one mapped user called AllAuthenticated in the role AllAuthenticated.

To configure the security roles, navigate to **Applications** → **Enterprise Applications** → **IBM_OpenPGP_Public_Key_Server** → **Security role to user/group mapping**.

To add additional groups or individual users to the role of PKSAdmin, select **PKSAdmin** and then click either **Look up users** or **Look up groups** (Figure 9-26).

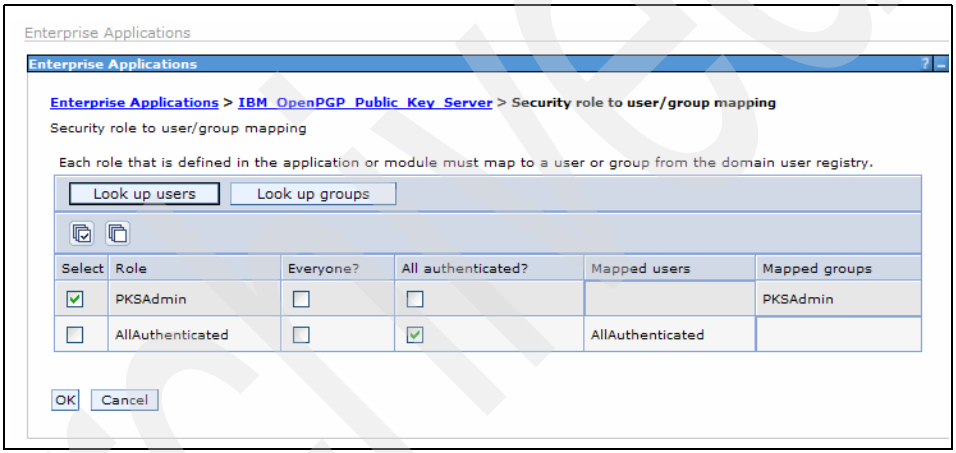


Figure 9-26 Adding group/user to a role

From here, you can search for individual users or groups and add them to this role, as shown in Figure 9-27.

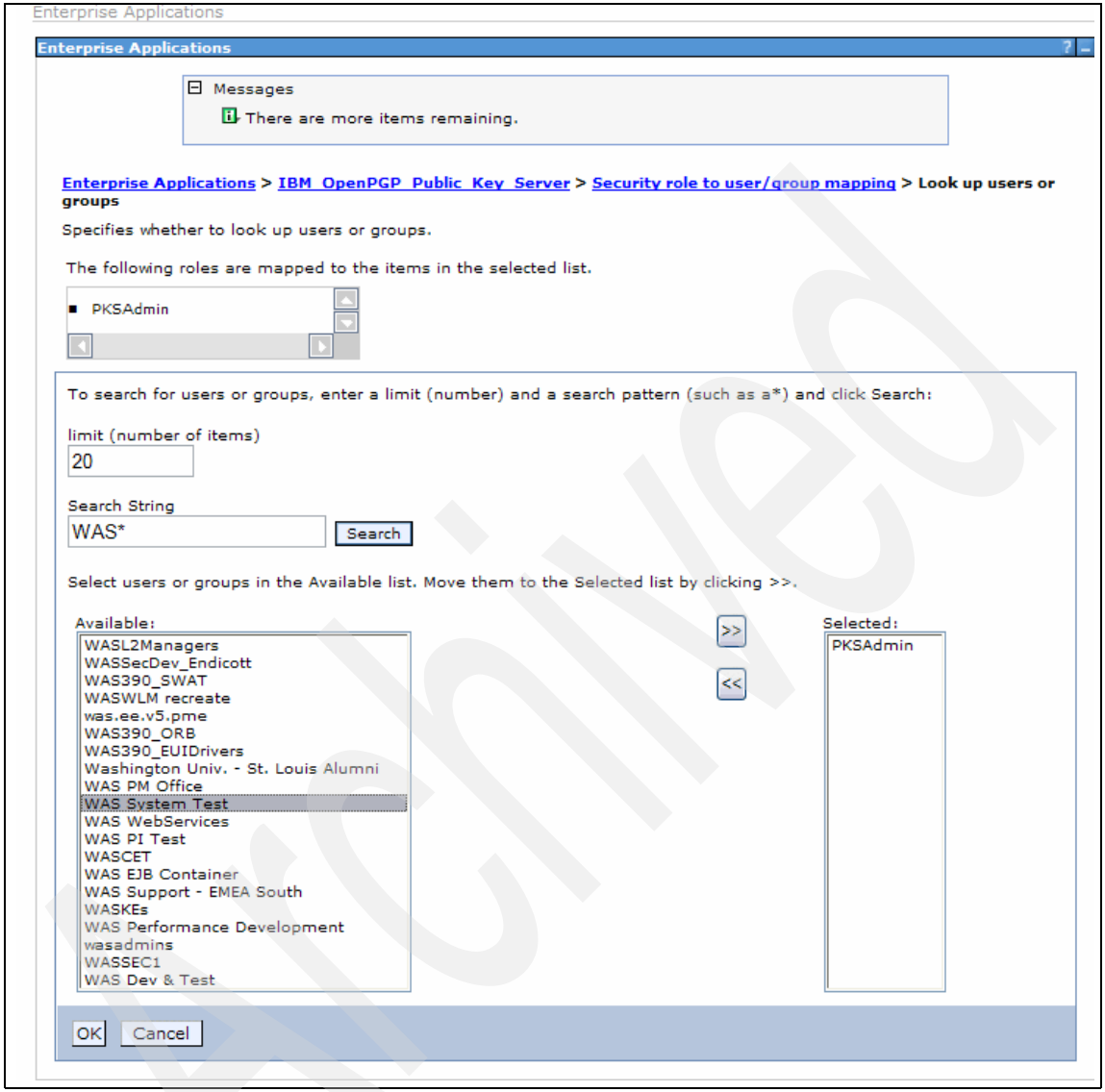


Figure 9-27 Adding users/groups to a role

After adding all of the users and groups to the role, click **OK** and **Save**. It is recommended that you restart WebSphere at this point to pick up all of the security configuration changes.

9.5.6 Create pgppkiserver.properties

This file contains information about the LDAP server, and stores external user's authentication information as well as all of the OpenPGP certificates used by this application. This file must be placed in the directory pointed to by the OPENPGP_HOME variable. It must be readable by the WebSphere servant address space and must contain the following properties. The values within brackets, and the brackets themselves, must be replaced with values from the LDAP administrator.

```
ldap.host=<hostname or ip of LDAP server that will store OpenPGP keys>

ldap.port=<port number for LDAP server. Usually 389>

ldap.suffix=<suffix used by LDAP server for OpenPGP keys>

ldap.userid=<LDAP administrator user id>

ldap.userid.password=<LDAP administrator user id's password>
```

At this time, the `ldap.userid.password` must be in the clear.

9.6 Using the IIPKS

At this point, the Key Server should be ready for use. The context root of this application is /pks, so point your browser to http://hostname:port/pks. This should redirect you to the login page if you have not already authenticated. Figure 9-28 shows the login page.

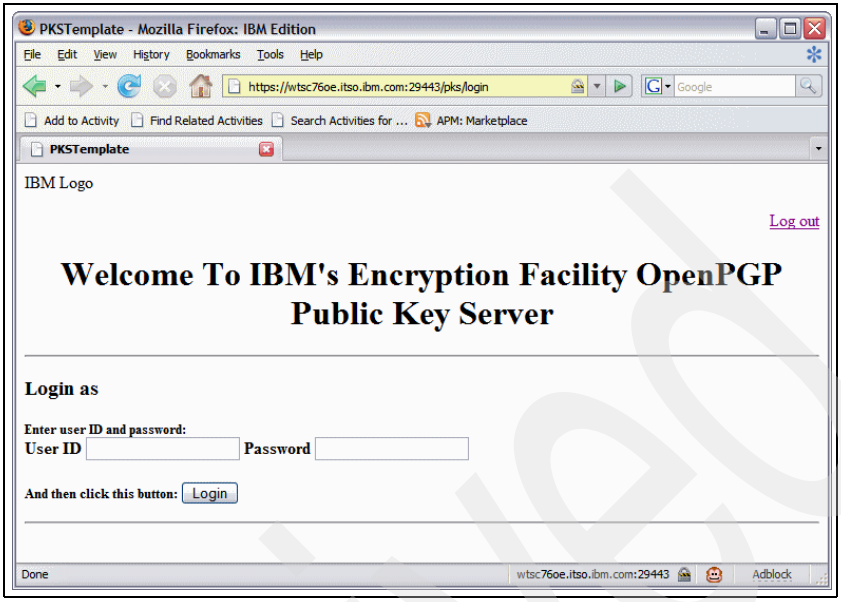


Figure 9-28 Login screen

Once you have logged in, you will see a home page like the one in Figure 9-29. In this example, we are listed as a regular user. We determine that we are not an administrator since there are only options to Submit and Search for OpenPGP keys. Administrator privileges grant capabilities to Remove a key, Add User, and Enable or Disable a key, as shown in Figure 9-30.

Figure 9-29 Default user menu

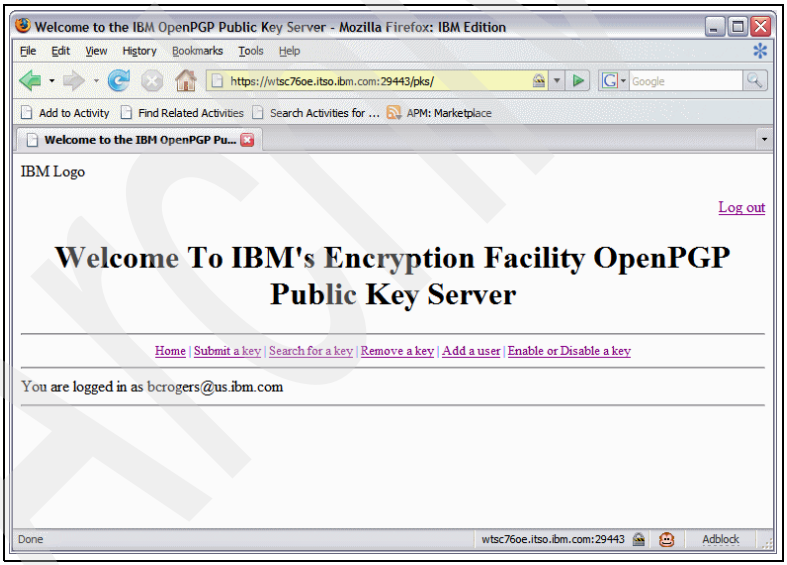


Figure 9-30 Administrator user menu

The menu selections and options on each page are somewhat self-explanatory, so we only provide a brief discussion of the functions in the following sections.

9.6.1 Search for an OpenPGP key

To search for an OpenPGP key, you can search by key ID, which must be prefixed with `0x`, or you can search the user IDs associated with the key with a plain string like `ken`. There is only one search box on this form (Figure 9-31); it will handle either text or hex key searches.

Figure 9-31 Search panel

Figure 9-32 is an example of a “vindex” search result. The index search result lists several pieces of information about the certificate, including the size of the key, the type of key, the key identifier, the date it was created, and the key owner’s identity. The verbose index (or vindex) search displays signature data for the key as well. In this figure, we can see that Todd Zimmerman’s key has been signed with key ID D927A19BCB9FC634. However, because this key ID is not found in our keyring, it is reported as “Unknown User.”

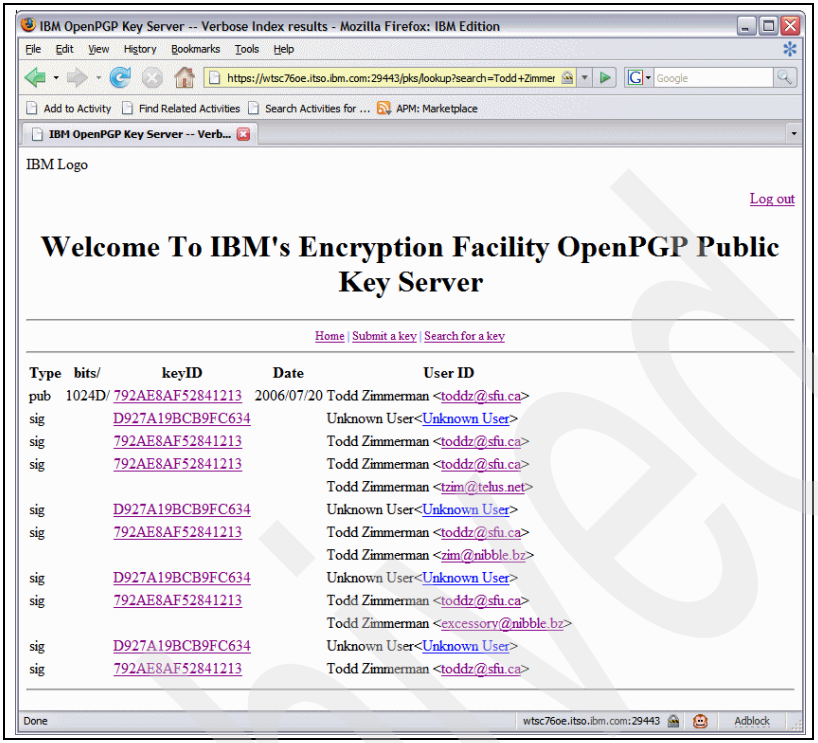


Figure 9-32 Index search results

To retrieve the actual key, we can click on the link under keyID. This will perform a “get” operation and will return a result like the one seen in Figure 9-33.

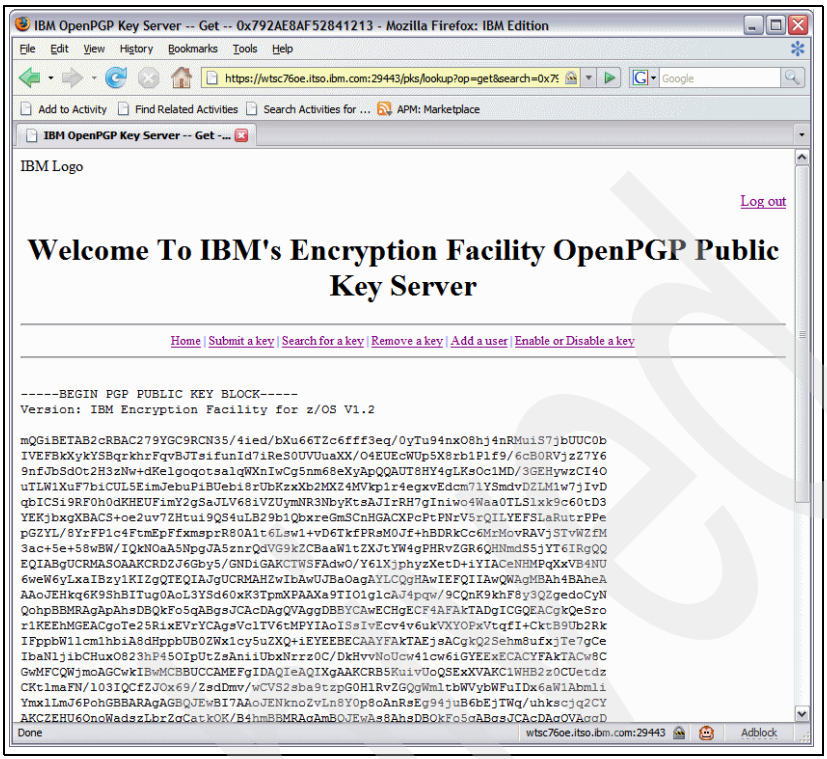


Figure 9-33 Retrieving the key

9.6.2 Submit an OpenPGP key to the key server

To submit a key to the key server, you must have an ASCII-armored version of the key. The easiest way to check this is to open the file containing the key in a simple text editor like Notepad or Wordpad. You should see a clear text header and footer stating **"**`-----BEGIN PGP PUBLIC KEY BLOCK-----`**"** and **"**`-----END PGP PUBLIC KEY BLOCK-----`**."** Copy the entire contents of the file into your computer's copy buffer and paste the contents into the text box. Be sure that you get everything between and including the `-----BEGIN PGP PUBLIC KEY BLOCK-----` and `-----END PGP PUBLIC KEY BLOCK-----` in the text box. Also, be sure when you paste this information that it appears as it does in the text editor. The number of characters per line and formatting is *very* important. A sample key can be found in Figure 9-34.

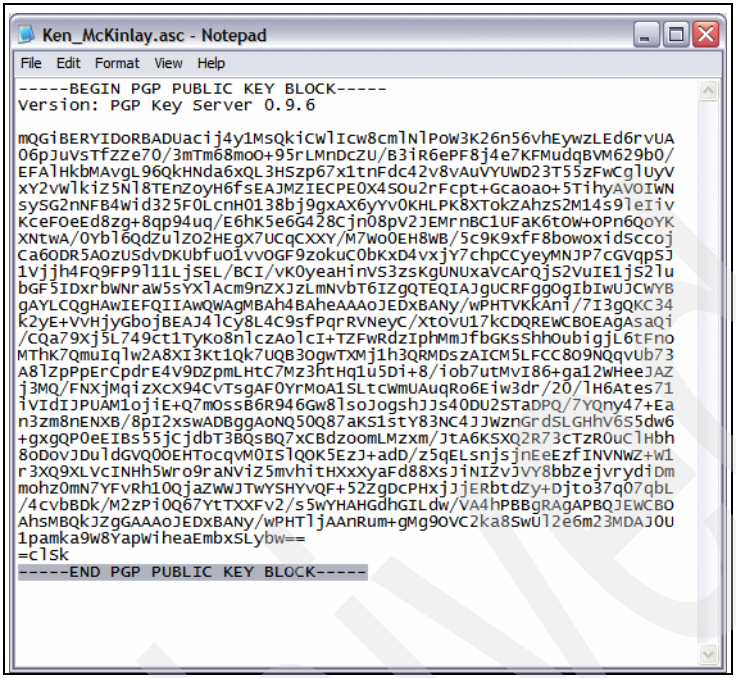


Figure 9-34 Sample key to submit to key server

Select the **Submit a key** link and paste in the contents of your copy buffer and click **Submit this key to the keyserver!** as shown in Figure 9-35.

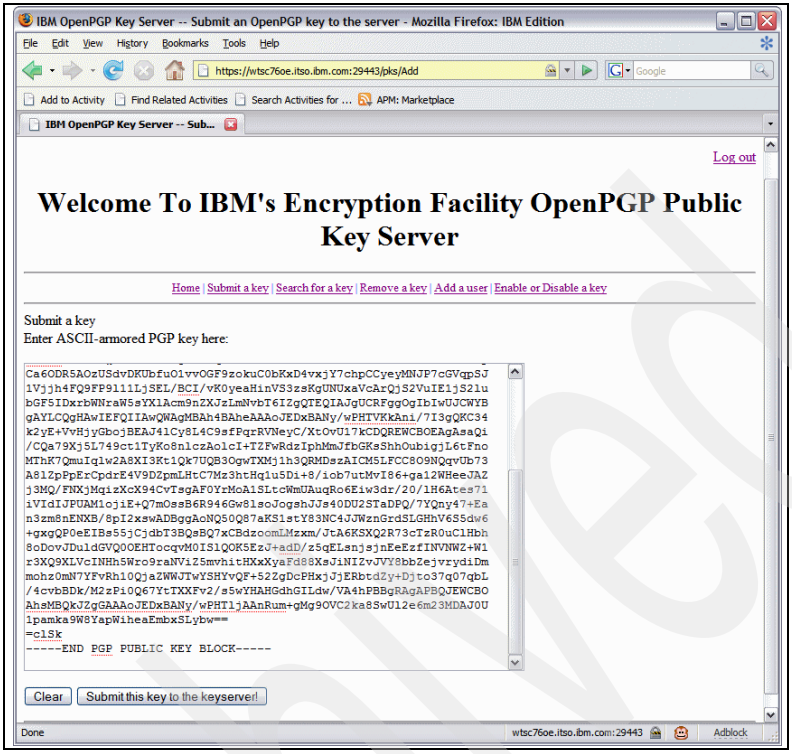


Figure 9-35 Submitting new Key

After submitting the key to the key server, you will receive a message indicating whether the key has been successfully added to the server.

9.6.3 Add external user

An administrative page is available for adding users that are not already in the primary user registry. This page attempts to add a user to the LDAP directory that is defined in pgppkiserver.properties. At this time, there is no ability to modify the required input for creating a user. The Add user function is a very simplistic method for adding a user to an LDAP directory. It requires a first name (cn), last name (sn), email address (mail), and a password. Optional fields on this form include Organization name (ou) and telephone number (telephonenumber). If desired, users can be added to the LDAP directory through another means like manual entry, or some automated process. This page is really intended only as a quick utility interface into the LDAP directory. Figure 9-36 shows an example of the Add user panel.

Figure 9-36 Add user

9.6.4 Using the Enable/Disable function

The Enable/Disable function allows administrators to remove a key from the view of non-administrative users. Disabling a key does not change the characteristics of the key, but instead flips an indicator within the LDAP directory that lets the application know this key should not be used. This is in contrast to the notion of revoking a key. When a key is revoked, the certificate material surrounding the key is changed and then signed. Typically in the OpenPGP paradigm, a public key can only be revoked by its corresponding private key. Since this is potentially problematic, such as is the case when a user needs their key revoked but can't

remember the password to their private key, this application uses a less stringent approach to preventing users from using a key.

The Enable/Disable function can only be performed on one key at a time, and when searching for a key to disable, the administrator must know the Key ID of the key they wish to revoke. This is an attempt to prevent the unintended disabling of keys that could occur if multiple keys are returned from a search.

To enable or disable a key, you must be logged on as an administrator. Click the **Enable or Disable key** link and enter the Key ID of the key you wish to operate on. The Key ID must be preceded with 0x, which indicates it is a hex value for a specific key, as can be seen in Figure 9-37.

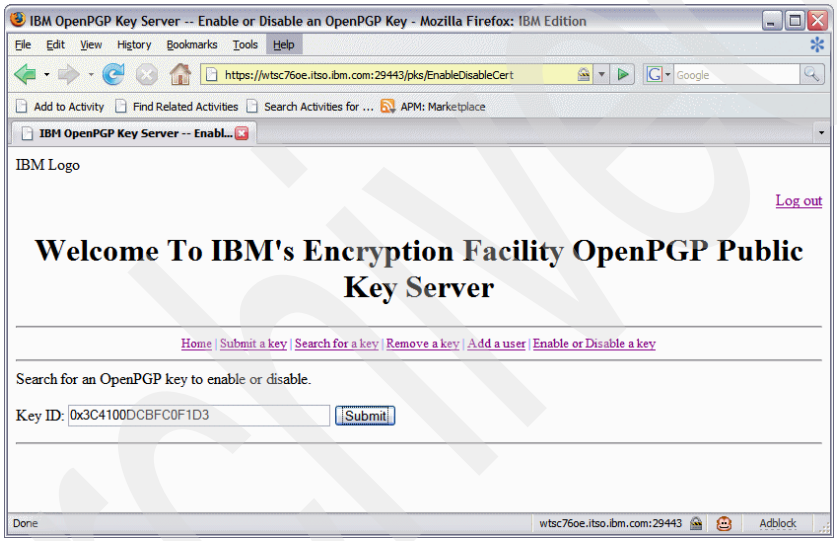


Figure 9-37 Searching for Key ID

When the search is complete, a set of radio buttons will be available, along with a Submit button. The radio button is always selected so that no action is taken by default. This means that if a key is enabled, when the search result comes back, the Enable radio button is selected. If you wish to disable the key, you must explicitly select the Disable radio button and click Submit as can be see in Figure 9-38.

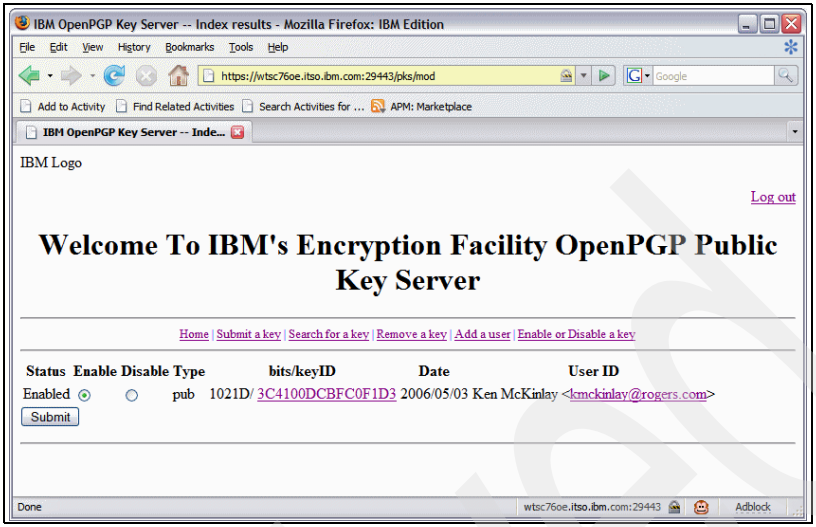


Figure 9-38 Enable/Disable function

9.7 Putting it all together

This section walks the reader through the entire process as experienced by both the organization or person submitting an OpenPGP key and the person retrieving the OpenPGP key for use with the Encryption Facility OpenPGP feature.

The first thing we do is add a new “external” user to our repository. We add Ned Flanders for this exercise, as shown in Figure 9-39.

Figure 9-39 Create external user Ned Flanders

Figure 9-40 shows that we have successfully added the user Ned Flanders to our LDAP repository.

Figure 9-40 Successfully added Ned Flanders

After Ned has been created, we log out of the application.

Next, Ned logs in as ned@flanders.com to this application. This is done by pointing the browser to the login page http://localhost:8080/pks. For this exercise, Ned was given a password of `secret`.

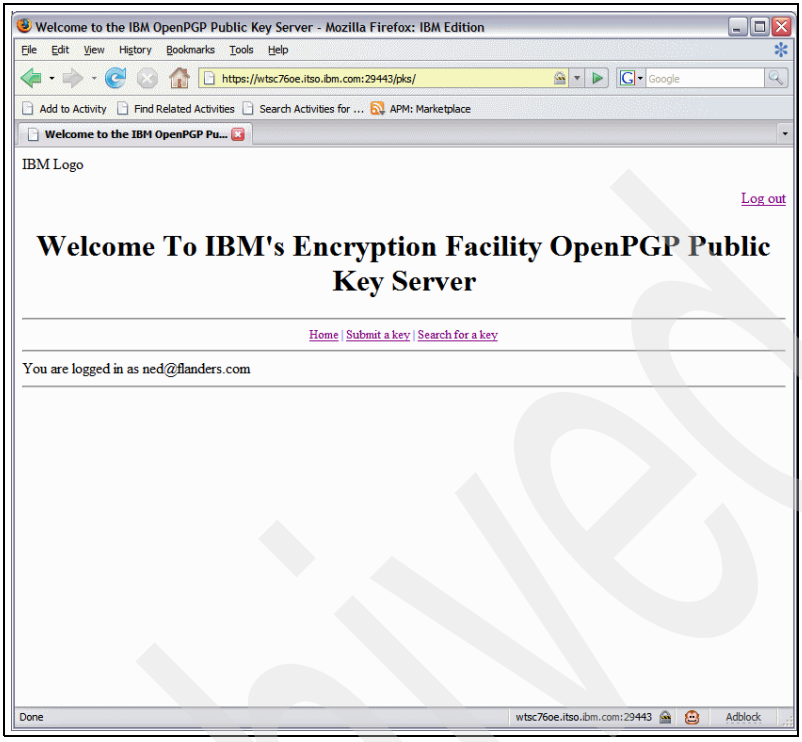


Figure 9-41 Successful log on

As Figure 9-41 shows, Ned has successfully logged in as ned@flanders.com. Ned is now ready to submit his OpenPGP key to the server. First, Ned right-clicks on his ASCII armor OpenPGP key file (probably on his workstation) and opens it with a simple text editor like Notepad or Wordpad. Then, after selecting all of the text, he copies it to the clipboard by using Ctrl + c. Next, Ned clicks the **Submit a key** link in the application, which brings up the page in which an ASCII armor key can be entered. He right-clicks in the text box in the Web browser and selects **Paste**. This should paste the ASCII armor certificate into the text box as demonstrated in Figure 9-42. Finally, he clicks **Submit this key to the key server!** A result page indicating that the key has been successfully added to the key server is displayed, as shown in Figure 9-43.

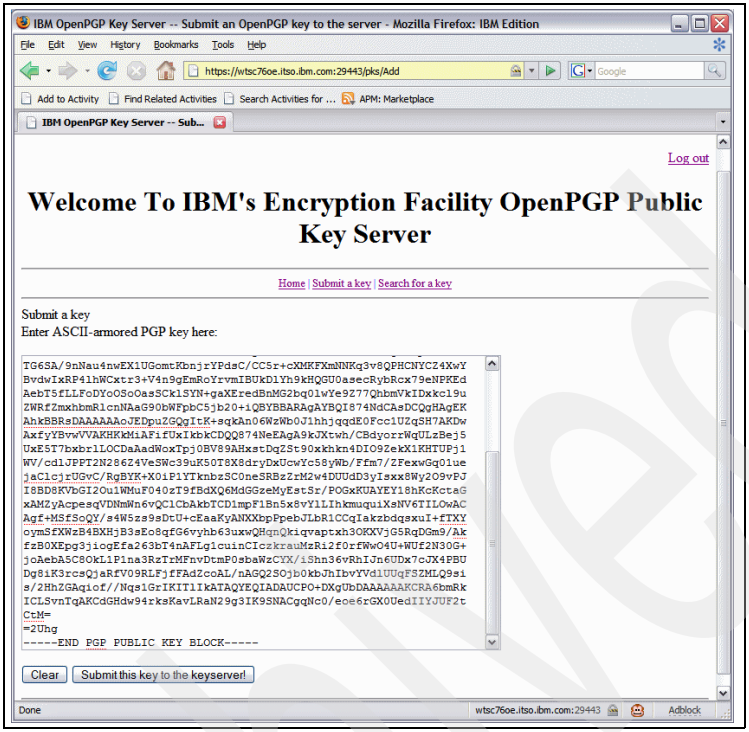


Figure 9-42 Pasting the ASCII armored certificate

Figure 9-43 Successfully added Ned Flander's key to the key server

Ned's part in this is now complete. The organization that needs to send Ned encrypted data now has his OpenPGP key and can retrieve it for processing. At this point, Homer, a user from within the organization, has logged in and is searching for Ned's OpenPGP key. Homer has navigated to the Search for a key page and has entered ned as his search argument. He clicks **Do the search** and an index result page is displayed with one result matching the search criteria of ned, as is shown in Figure 9-45.

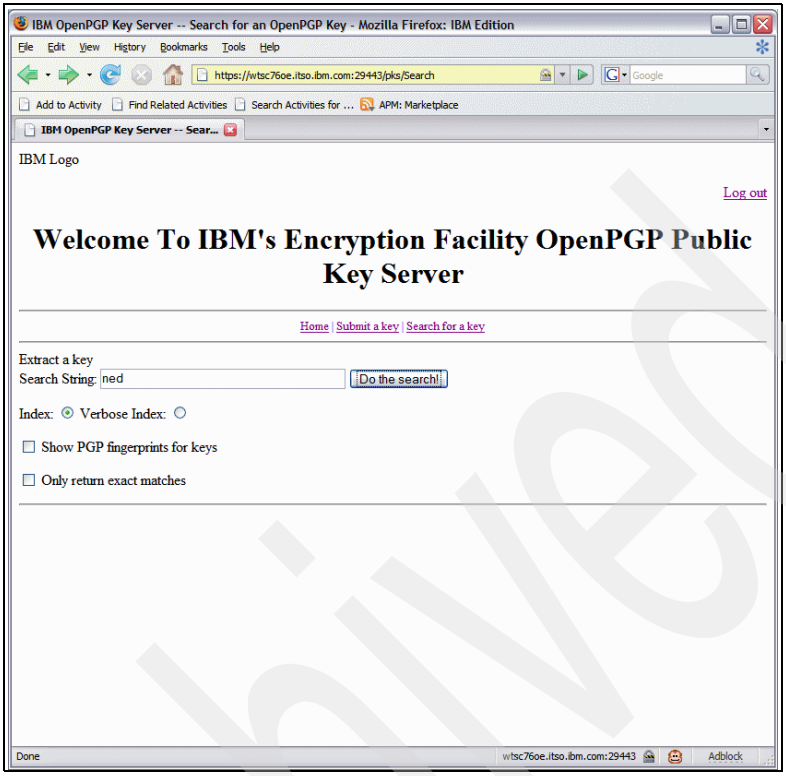


Figure 9-44 Searching for Ned's OpenPGP key

Figure 9-45 Ned's OpenPGP key

Homer recognizes Ned's email address as dr_ned_flanders@hotmail.com and thus has a reasonable assurance that this key is the one he wants to use to send encrypted data back to Ned. Homer clicks the keyID link, which brings up the ASCII armor version of Ned's key, as shown in Figure 9-46.

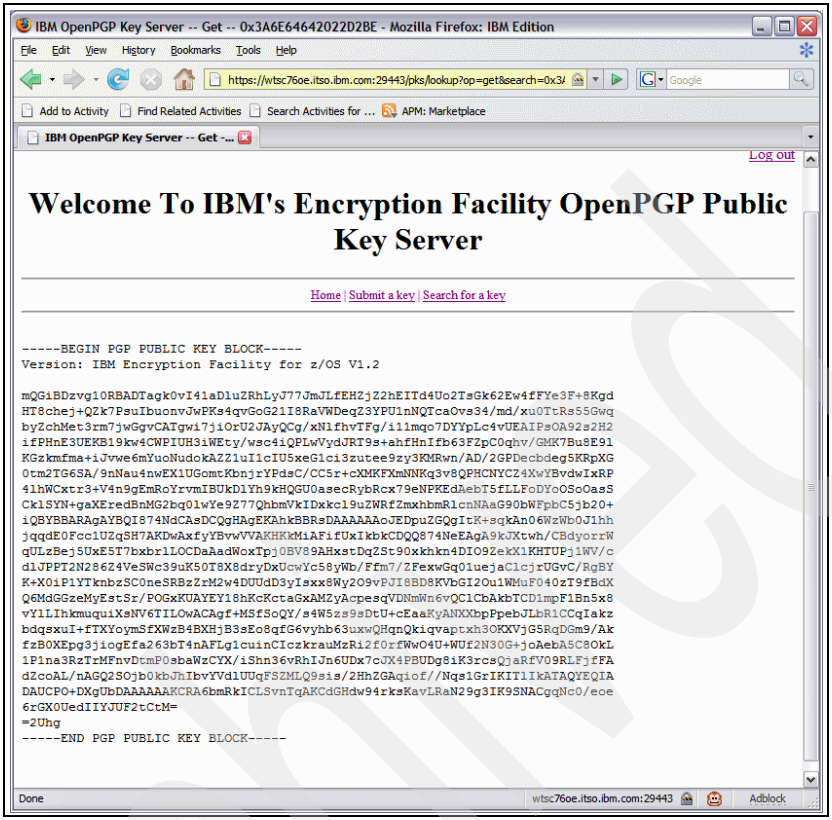


Figure 9-46 Ned Flander's OpenPGP key

Homer saves Ned's OpenPGP key to a file locally on his workstation before FTPing it over to his z/OS system, as shown in Figure 9-47.

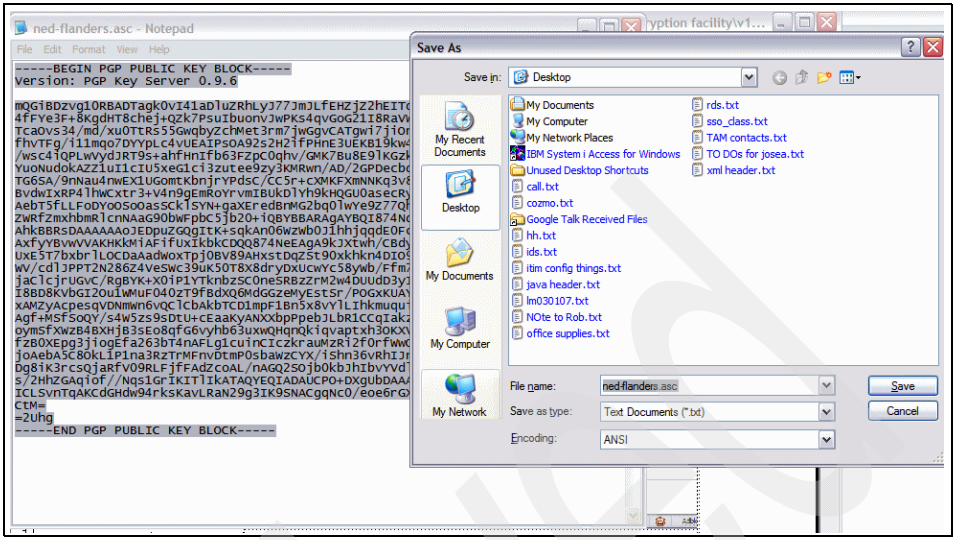


Figure 9-47 Save Ned's OpenPGP key locally

Once Homer has saved Ned's OpenPGP key to his workstation, he can FTP this file to his z/OS system. He FTPs the ASCII armor file in "ascii" mode, otherwise it could not be imported into the Encryption Facility OpenPGP key ring. A quick way to check whether the OpenPGP key will probably be able to be imported is to open the file with OEDIT or **vi** and double check that it looks like it did before it was sent over, that is, the clear text header and footer are still here.

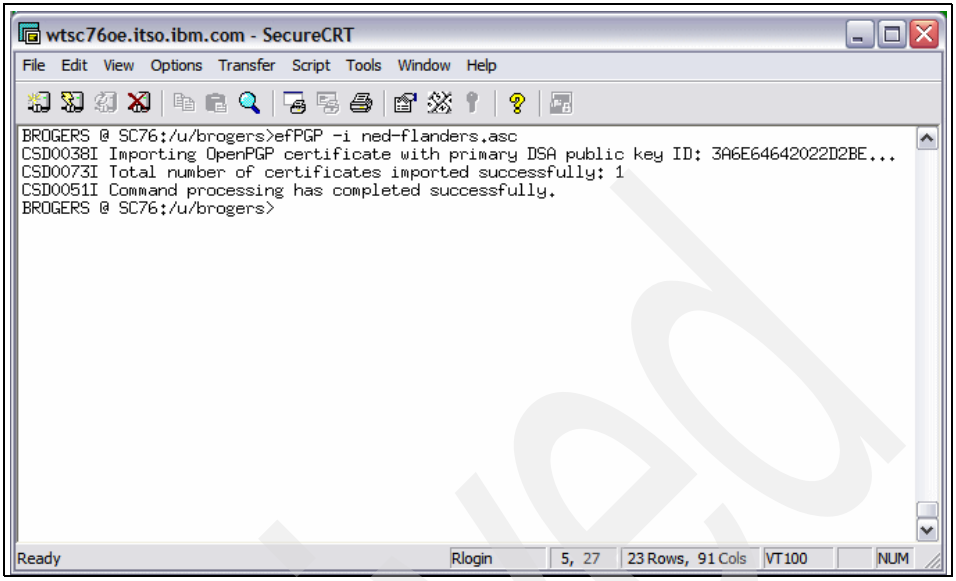


Figure 9-48 Importing the OpenPGP certificate

Now that the file ned-flanders.asc has been FTPed over to the z/OS system, Homer can import this key into his OpenPGP keyring on his z/OS system. This is done with the **-i** (import) option on the OpenPGP utility. Figure 9-48 illustrates this operation with the use of a shell script that makes typing slightly less problematic. The full text of this command is:

```
java -jar /usr/lpp/encryptionfacility/CSDEncryptionFacility.jar
-homedir /u/brogers -i ned-flanders.asc
```

At this point, Ned has sent his OpenPGP key to Homer's organization and Homer has retrieved this key and imported it into his local keyring so that he can create encrypted files that will be sent to Ned.

10

Performance

This chapter discusses performance in the context of the two metrics: the use of the zAAP processor and the use of parallel processing. Statistics and comparisons illustrate the impact of the different performance-enhancing options and demonstrate how the OpenPGP support on z/OS can provide full OpenPGP compliance while efficiently using system resources and completing tasks in a timely manner. Ultimately, the efficient use of system resources will result in an overall lower financial cost for performing essential data integrity services within the enterprise.

Attention: Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending on considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

This information is presented with general recommendations to help you better understand IBM products. In addition, note that:

- All tests were run on IBM zSeries System z9.
- All data was gathered using a 1.5 GB input file.
- All I/O is done to and from data sets on DASD.
- The encryption algorithm used was AES with 128-bit keys.
- Due to compression, the output file might not be the same size as the input file.
- The CPU utilization percentage given in many tables is the average utilization measured over all enabled processors of the same type. That is, an average might be given for all the zAAP processors and a different average might be given for all the general CPUs. As a result, when multiple processors are online, the calculated average might appear to be affected. However, the aggregate of CPU service unit consumption is roughly equivalent.
- The elapsed time is given in seconds.
- For statistics for zAAP processors, two internal throughput numbers are given: throughput for the general CPUs only and throughput for the general CPU and zAAP combined.
- When doing compression, the ZIP compression algorithm was used with level 1. The value of 1 represents best speed.

  Note: Tests have shown a compression level of 9 – best compression, results in a very significant performance degradation.
- Compression performance and compression percentages are greatly affected by the type of data being compressed.

10.1 Overview

Performance is critical to the deployment of any solution in a business's enterprise. The following two performance metrics are used to discuss the performance implications of deploying Encryption Facility V1.2's OpenPGP support:

- Execution time measured by a standard watch or clock and the amount of work accomplished within that time
- Cost measured by the CPU consumption on general-purpose processors

The OpenPGP support is a stand-alone Java application. Applications written in Java on z/OS require the IBM Java Runtime Environment for z/OS (JRE). This environment consists of an address space where the JRE executes. The JRE interprets compiled Java byte code that serves as the OpenPGP support's program code. Given the additional layer of execution and Java's interpretive nature, the amount of CPU time consumed and the amount of execution time to complete a task are affected immediately. Moreover, the cryptographic nature of the Encryption Facility products requires the completion of computation-intensive operations that could further increase the amount of CPU time consumed and execution time. Finally, OpenPGP's main purpose is to provide data integrity services on data sets and UNIX System Services (USS) files. This involves I/O-intensive operations that might result in inefficient utilization of the CPU. To address these issues, performance-enhancing options are available in three areas:

- IBM Java 5 SDK Runtime Environment for z/OS
- z/OS specialized hardware: zAAP and CPACF specialized processors
- OpenPGP support code

10.1.1 IBM Java SDK 5 Runtime Environment for z/OS

The IBM JRE for z/OS provides the just-in-time (JIT) compiler. The JIT compiler is responsible for optimizing Java byte code and compiling the interpreted code into persistent load modules. Compiling code into the persistent load modules alleviates the CPU service unit cost of reinterpreting code when it is executed again during the life of the application. Moreover, because it is aware of the processor type, the JIT compiler can use the latest hardware technology to optimize the compilation.

Note: It is strongly recommended that the JIT compiler always remain enabled. Thus, we will not discuss the performance implications of disabling the JIT compiler. If disabled, the performance of any Java application may be degraded significantly.

10.1.2 z/OS specialized hardware: zAAP and CPACF

The CP Assist for Cryptographic Function (CPACF) is the z/OS feature that can be used to accelerate symmetric encryption and decryption and hash calculations. The CPACF performs cryptographic functions at an accelerated rate, helping to reduce the total elapsed time and CPU time.

The System z Application Assist Processor (zAAP) is specialized processing unit that only runs Java workloads. When workloads are dispatched on a zAAP processor, no general CPU service units are consumed. These features can ultimately result in a lower overall financial cost of performing data integrity services on data sets or files.

10.1.3 Encryption Facility OpenPGP support

The OpenPGP support provides configuration options that enable parallelized internal processing. This directly addresses the inefficient use of the CPU while I/O operations are being performed. These options pipeline the different tasks that are required to provide OpenPGP compliance; encryption, I/O, and compression (if enabled) may execute in parallel and on different CPUs.

10.2 General CPU Service Units reduction using z/OS specialized hardware

Two specialized processors may be leveraged to make a positive impact on performance: zAAP specialized processor and the CPACF embedded processor. Whereas zAAP usage is external to the OpenPGP configuration, the OpenPGP support allows two options for performing the cryptographic functions required by RFC 2440. The user may set the JCE_PROVIDER_LIST configuration option to `com.ibm.crypto.hdwrCCA.provider.IBMJCECCA` or specify the `–jce-providers` command line option with value `com.ibm.crypto.hdwrCCA.provider.IBMJCECCA`. This enables the hardware cryptographic JCE provider. This provider leverages ICSF and the CPACF to perform cryptographic functions. If these options are not specified and the security.provider.1 keyword in the

$(java_home)/lib/security/java.security file has not been updated to list the com.ibm.crypto.hdwrCCA.provider.IBMJCECCA provider, Java code within the JCE component of the JRE will perform all the cryptographic function.

10.2.1 Hardware cryptographic acceleration

Table 10-1 lists the performance statistics measured *without* the hardware provider enabled, only *one* general CPU online, and compression level 1 enabled.

Table 10-1 Performance statistics without hardware provider and one CPU

Input data size (bytes)	Output data size (bytes)	Elapsed time (sec)	MB/sec	CPU utilization %	MB/CPU sec
1,506,431,796	346,727,868	172.86	8.31	74.38%	11.17

Table 10-2 lists the statistics measured with the hardware provider *enabled* and only *one* general CPU online.

Table 10-2 Performance statistics with hardware provider and one CPU

Input data size (bytes)	Output data size (bytes)	Elapsed time (sec)	MB/sec	CPU utilization %	MB/CPU sec
1,506,431,796	346,727,868	144.71	9.93	70.84%	14.01

The hardware provider reduced the elapsed time by 16% and improved the MB/CPU seconds by 25%.

10.2.2 zAAP usage

Table 10-3 lists the statistics for the same test but measured *without* the hardware provider being enabled and with *one* general CPU online and *one* zAAP process online.

Table 10-3 Performance statistics with specialized hardware

Elapsed time (sec)	MB/sec	zAAP utilization %	General CPU utilization %	MB/general CPU sec	MB/general + zAAP sec
167.32	8.59	69.36%	11.62%	73.89	10.60

Comparing the results in tables 1 and 3, general CPU utilization is reduced by more than 84%. Table 3 separates internal throughput (MB/CPU sec) into two categories: “MB/general CPU seconds” and “MB/zAAP+general CPU seconds.”

This distinction is given to allow those who use throughput as an informal indicator of cost to quickly see the financial impact of a zAAP processor. Finally, the introduction of the zAAP processor showed a slight reduction in elapsed time of about 3%. Using a zAAP, the MB/zAAP+general CPU seconds was about 5% lower than the total MB/CPU seconds without a zAAP. Without the hardware provider, 86% of the total CPU activity is eligible to run on a zAAP.

These comparisons are also valid for the case when the specialized cryptographic processor is being leveraged.

Table 10-4 lists the statistics measured with the hardware provider being *enabled* and *one* general CPU online and *one* zAAP process online.

Table 10-4 Performance statistics with specialized hardware and hardware provider

Elapsed time (sec)	MB/sec	zAAP utilization %	General CPU utilization %	MB/general CPU sec	MB/general + zAAP sec
136.53	10.52	66.29%	12.83%	82.02	13.30

Comparing the results in table 2 and 4, general CPU utilization is reduced by more than 80% while the calculated general MB/CPU sec is increased by nearly 10%. Using a zAAP, the MB/zAAP+general CPU seconds was about 5% lower than the total MB/CPU seconds without a zAAP. Using the hardware provider, 84% of the total CPU activity is eligible to run on a zAAP.

10.2.3 Execution time reduction using parallel processing

The OpenPGP support's processing characteristics allow for processing inefficiencies. Specifically, while the OpenPGP support code awaits the completion of an I/O operation the CPU is waiting. This holds true for both general-purpose CPUs and specialized zAAP processors. An idle CPU can be seen as lost opportunity to reduce the total execution time.

The OpenPGP support provides three configuration file options that enable a multi-threaded approach to processing, as described in Table 10-5.

Table 10-5 Configuration file options

Configuration option	Processing task
USE_ASYNC_IO	File or Data Set Input/Output
USE_ASYNC_CIPHER	Encryption/Decryption
USE_ASYNC_COMPRESS	Compression/Decompression

When any or all of these configuration options are enabled, the associated processing tasks are performed in a separate thread of execution. In effect this relieves the CPU from waiting on the task to complete before continuing main line processing. Further, multiple CPUs can be used to handle processing concurrently. These benefits make way for a reduction in execution time.

Table 10-6 (shown here for simplicity) lists some performance statistics of encrypting and compressing *without* any of the options enabled and only *one* general CPU online.

Table 10-6 Performance statistics without any options enabled

Input data size (bytes)	Output data size (bytes)	Elapsed time (sec)	MB/sec	CPU utilization %	MB/CPU sec
1,506,431,796	346,727,868	144.71	9.93	70.84	14.01

In contrast, Table 10-7 lists the statistics when USE_ASYNC_IO, USE_ASYNC_COMPRESS, and USE_ASYNC_CIPHER are *enabled* and only *one* general CPU is online.

Table 10-7 Performance statistics with options enabled

Input data size (bytes)	Output data size (bytes)	Elapsed time (sec)	MB/sec	CPU utilization %	MB/CPU sec
1,506,431,796	346,727,868	178.90	8.03	74.22	10.82

With only one general CPU available, full parallel processing actually shows an increase of elapsed time of about 24% and a 23% drop in MB/CPU seconds.

Table 10-8 shown next, lists the statistics when USE_ASYNC_IO, USE_ASYNC_COMPRESS, and USE_ASYNC_CIPHER are enabled and *four* general CPUs are online.

Table 10-8 Performance statistics with options enabled

Input data size (bytes)	Output data size (bytes)	Elapsed time (sec)	MB/sec	CPU utilization %	MB/CPU sec
1,506,431,796	346,727,868	94.02	15.28	33.91	11.27

A 47% improvement is seen when comparing the elapsed times of *enabling all* of the parallel processing features and with *four* general CPUs online, to *no parallel processing* and only *one* general CPU online. External throughput (MB/sec) is significantly improved. Internal throughput (MB/CPU sec) shows a slight improvement.

Comparing to Table 10-6, on a system with four general CPUs the ASYNC_ options provide a 35% reduction in elapsed time. However, they increase the CPU cost by 24%.

10.3 Putting it all together

The previous sections have separately shown the effectiveness of introducing zAAP and hardware cryptography. Also, when multiple CPUs were available, the parallel processing options showed improvements in the elapsed time needed to complete encryption with compression. This section demonstrates the overall impacts of combining parallel processing with multiple CPUs and specialized hardware.

Table 10-9 lists the statistics for the same tests when USE_ASYNC_IO, USE_ASYNC_COMPRESS, and USE_ASYNC_CIPHER are enabled and two general CPUs are online and two zAAP CPUs are online.

Table 10-9 Performance statistics with specialized hardware and hardware provider

Elapsed Time (sec)	MB/sec	zAAP utilization %	General CPU utilization %	MB/general CPU sec	MB/general zAAP sec
128.11	11.21	56.10%	3.59%	156.19	9.39

Table 10-10 (shown here again for simplicity) lists the statistics when one general CPU is online, no parallel processing options are enabled, and the hardware provider is not specified.

Table 10-10 Performance statistics without hardware provider

Input data size (bytes)	Output data size (bytes)	Elapsed time (sec)	MB/sec	CPU utilization %	MB/CPU sec
1,506,431,796	346,727,868	172.86	8.31	74.38	11.17

Table 10-9 shows significant improvement over Table 10-10:

- Elapsed time was reduced by more than 25%.
- General CPU utilization was reduced by more than 95%.
- MB/CPU seconds increased by approximately 35%.
- While the total CPU cost (general processor and zAAP) went up by 19%, a full 94% of the CPU activity was able to run on a zAAP. For installation with multiple CPUs and zAAPs configured, this probably is the lowest cost option.

10.4 Conclusion

Performance characteristics can differentiate a product's usefulness to an enterprise. Two metrics that are key to the analysis of performance are elapsed clock/watch time needed to complete a task and the CPU time used to complete said task. We demonstrated some of the techniques that can used to achieve enhanced performance when using OpenPGP services. Clearly, all installations with CPACF should exploit this technology. Installations with multiple general CPUs and zAAP processors online can benefit from enabling of all of the parallel processing options for the OpenPGP support.

A

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 “secret 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. In fact, key1 and key2 are mathematically linked and can only work with each other.

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 or 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 raises the issue of securely transferring the secret key to 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.

First, 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.

Second, 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 sender knows the receiver'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 (Figure A-1).

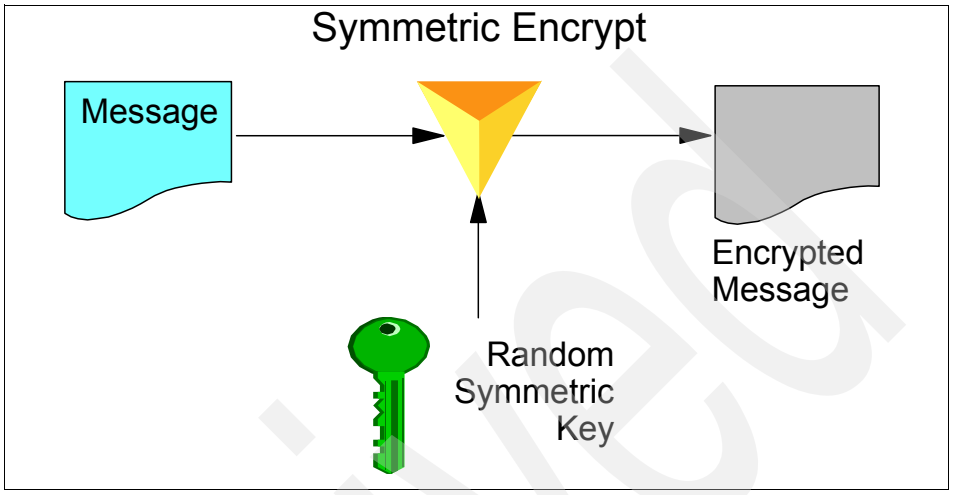


Figure A-1 Secure message process: Symmetric encryption of data

2. The sender now encrypts the random symmetric key using the receiver's public key (Figure A-2).

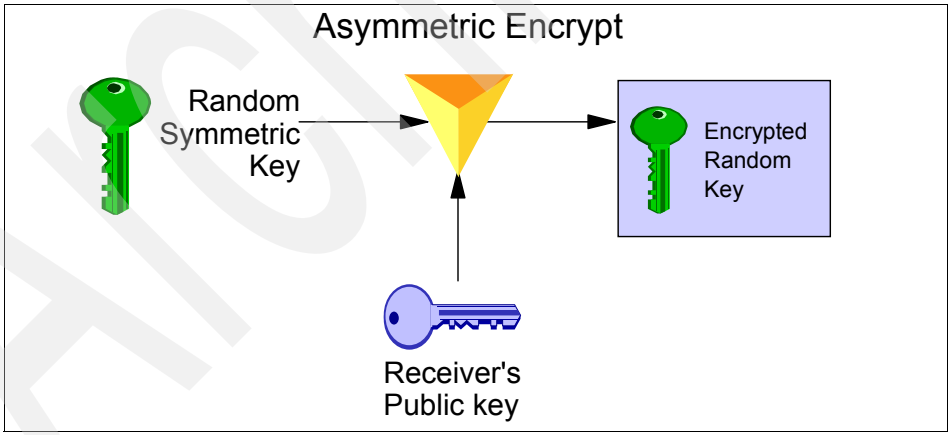


Figure A-2 Secure message process: Asymmetric encryption of symmetric key

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 (Figure A-3).

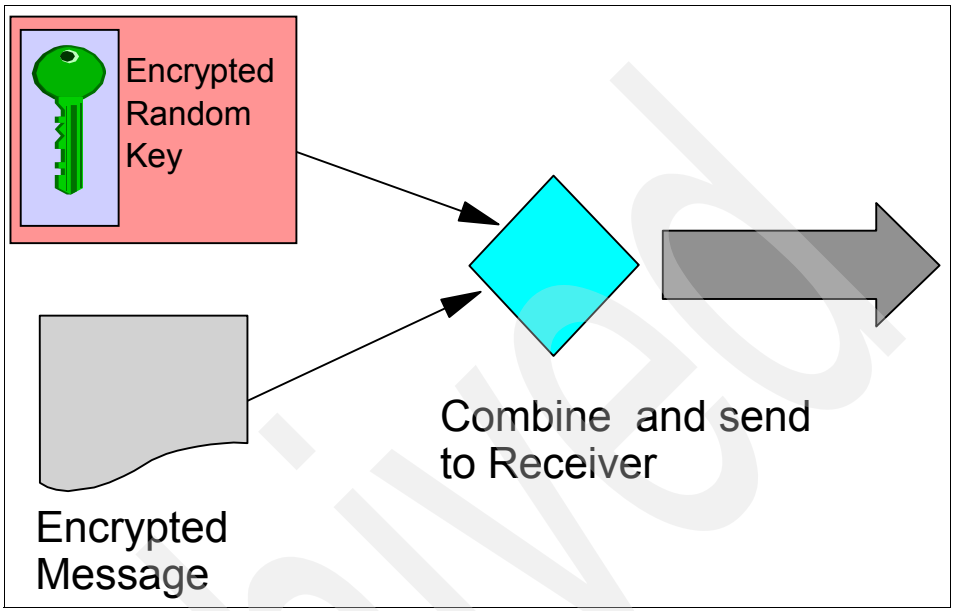


Figure A-3 Secure message process: Composition of the message

When the receiver receives the package, the receiver proceeds as follows:

1. The encrypted data key is decrypted with the receiver's private key (Figure A-4).

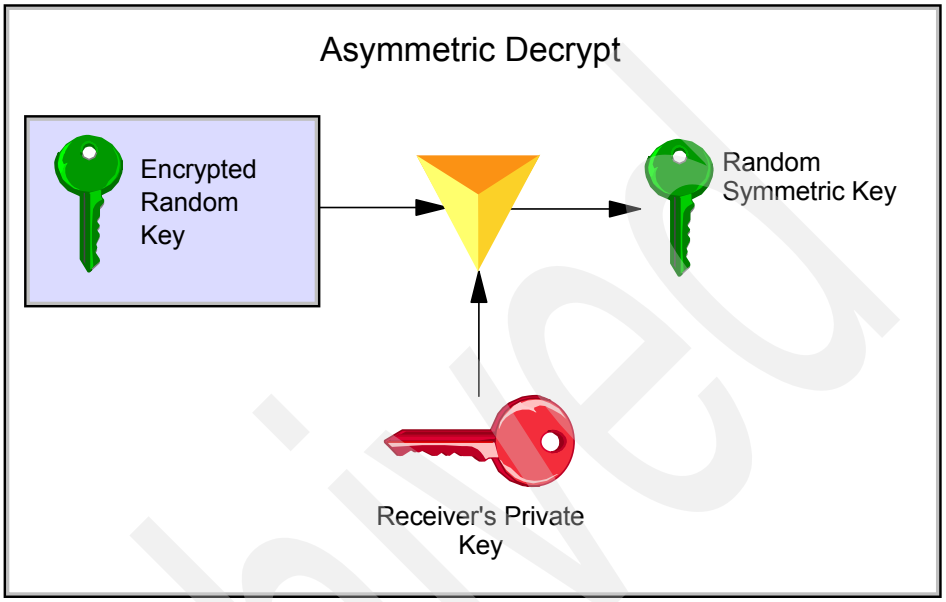


Figure A-4 Secure message process: Retrieval of the data key

2. The resulting symmetric key is then used to decrypt the long message (Figure A-5).

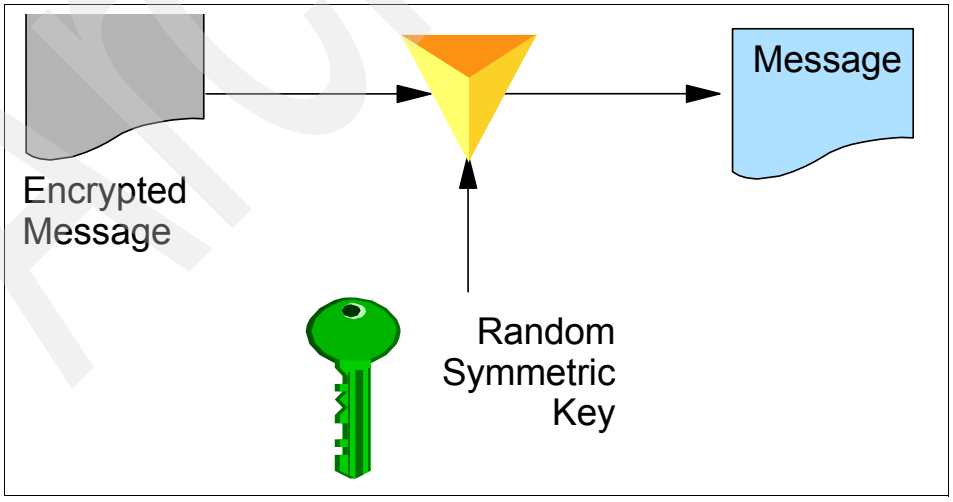


Figure A-5 Secure message process: Recovery of clear data at the receiver's site

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 are digitally signed by a trusted certificate authority, and therefore, the owner’s name is cryptographically bound to the key value.

A widely used format for these certificates is known as the X.509 V3 certificate, which is promoted by the IETF PKIX group. In this section we provide explanations based on the X.509 standard implementation, while the discussion in the rest of the book is based on the OpenPGP certificate format and how it compares to X.509. Note, however, that the Encryption OpenPGP support internally uses X.509 V3 certificates and keys.

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 it is 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 to the wider discussion of a public key infrastructure. This book does not cover this subject: for more information, refer to *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 that are 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 A-1.

Table A-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.

What it means to use cryptography

This section, as a complement to the previous information, provides some general considerations about the use of cryptography.

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.
- 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 flow 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

In the previous sections, we introduced 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 computer 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 for 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 is achieved when processing a long block of data, for example, a one megabyte-long record, for each encryption call, and the poorest throughput usually occurs with very short blocks of 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 with regard to 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 become more vulnerable to compromise as their lifetime increases. In current common practice, 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 encountered with software implementations. Hardware cryptographic coprocessors:

- 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.
- 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.

> **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 that heavily use 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 up the possible issue 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 a long period of time.

B

Configuration file options

This appendix describes the command options available, at the time of writing, as specified in the configuration file, with reference to the equivalent option in the command line. We sorted the options in this list by categories, as follows:

Note: The following configuration file options do not have equivalent command line options, and must therefore be specified in a configuration file if their default value does not fit the user's needs.

- KEY_RING_FILENAME
- USE_ASYNC_IO
- USE_ASYNC_COMPRESS
- USE_ASYNC_CIPHER
- DEFAULT_OUTPUT_DIRECTORY
- HIDDEN_PASSWORD

Commands output location

OUTPUT_FILE *user-specified-name*

Specifies the name of the destination file or z/OS data set. For data sets, you must preallocate the data set and specify a prefix of //. For example: //USER1.OUTPUT or for PDSE data sets, with the enclosed member name in single quotations, for example: '//USER1.OUTPUT(MEMBER1)'

Default: None.

Equivalent command option: -o — Specify an output location.

Arguments: For user-specified-name, the name of the file or data set.

OpenPGP keyring specification

KEY_RING_FILENAME *file-name*

Specifies the name of the OpenPGP keyring file where you can store OpenPGP certificates.

Default: /var/encryptionfacility/ibmpkring.ikr .

Equivalent command option: None.

Arguments: For file-name, the name of the OpenPGP keyring file.

RACF keyring specifications

RACF_KEYRING_USERID *RACF-id*

Specifies the RACF user ID to use when loading a RACF keyring. It is suggested that you use this option. If it is not specified, the user ID under which Encryption Facility runs might not be used even if it is specified on the JCL.

Default: The RACF user ID that Encryption Facility runs under.

Equivalent command option: -racf-keyring-userid — Specify a RACF user ID.

Arguments: For RACF-id, the RACF user id.

Java keystores specifications

JAVA_KEY_STORE_TYPE *type*

Sets the Java keystore type.

Default: Type as specified in the keystore.type property in the Java security properties file, JAVA_HOME/lib/security/java.security; if the property does not exist, JKS is the default.

Equivalent command option: -keystore-type — Specify the keystore type.

Arguments: For type, one of the following:

- JKS
- JCEKS
- JCECCAKS
- JCECCARACFKS - Read only
- JCERACFKS - Read only

JAVA_KEY_STORE_NAME *user-specified-name*

Sets the file name of the Java keystore. The value can either be one of the following:

- UNIX System Services filename
- RACF keyring name.For a RACF keyring name, the keystore type must be JCERACFKS. For a RACF keyring with keys in PKDS, the keystore type must be JCECCARACFKS and the hardware provider is required. You cannot specify z/OS-type data set names.

Default: None.

Equivalent command option: -keystore — Specify the name of the Java keystore.

Arguments: For user-specified-name, the name of the Java keystore.

KEYSTORE_PASSWORD *user-specified-password*

Specifies the password used to access the keystore.

Default: None.

Equivalent command option: -keystore-password — Specify the keystore password.

Arguments: For user-specified-password, the password for the keystore.

Key generation specifications

KEY_PASSWORD *user-specified-password*

Specifies the password for generating a new key in the keystore or for self-signing certificates when exporting a keystore key as an OpenPGP certificate.

Default: None.

Equivalent command option: -key-password — Specify the password for a new key.

Arguments: For user-specified-password, the password for generating a new key in the keystore.

KEY_ALIAS *user-specified-alias*

Specifies the alias for generating a new key in the keystore.

Default: None.

Equivalent command option: -key-alias — Specify the alias of a new key.

Arguments: For user-specified-alias, the alias for generating a new key in the keystore.

KEY_SIZE *size*

Specifies the key size for generating a new key in the keystore.

Default: 1024.

Equivalent command option: -key-size — Specify the key size to generate.

Arguments: For size, a key size to generate in the keystore. Key sizes depend on the hardware and software you are using.

Data signing key specifications

SIGNERS_KEY_PASSWORD *user-specified-password*

Specifies the password to access the signer's key in the keystore on this system. Encryption Facility uses this key when it works with message signatures.

Default: None.

Equivalent command option: -signers-key-password — Specify a password for the system key.

Arguments: For user-specified-password, the password to access the signer's key.

SIGNERS_KEY_ALIAS *user-specified-alias*

Specifies the alias of the signer's key in the keystore for the key on this system. Encryption Facility uses this key when it works with message signatures.

Default: None.

Equivalent command option: -signers-key-alias — Specify an alias for the system key.

Arguments: For user-specified-alias, the alias of the signer's key.

Certificate signing key specifications

SYSTEM_CA_KEY_ALIAS *user-specified-alias*

Specifies the alias of a certificate authority (CA) key in the keystore.

Default: None. Self-signs generated certificates.

Equivalent command option: -system-CA-key-alias — Specify an alias for a new key pair certificate.

Arguments: For user-specified-alias, the alias of the CA key.

SYSTEM_CA_KEY_PASSWORD *user-specified-password*

Specifies the password to access a certificate authority (CA) key in the keystore.

Default: None. Self-signs generated certificates.

Equivalent command option: -system-CA-key-password — Specify a password for the certificate authority key.

Arguments: For user-specified-password, the password to access a CA key.

JCE provider specifications

JCE_PROVIDER_LIST *string*

Prefixes the list of JCE providers in the java.security file that resides in $JAVA_HOME/lib/security/java.security A JCE provider in the list implements all cryptographic functions. See the java.security file in the ${java-home}/lib/security/java.security directory for more information on the provider list.

Default: List as specified in the security.provider.<n> properties in the Java security properties file JAVA_HOME/lib/security/java.security, JCE_PROVIDER_LIST

com.ibm.crypto.hdwrCCA.provider.IBMJCECCA. For hardware cryptographic acceleration, set the value to JCE_PROVIDER_LIST com.ibm.crypto.hdwrCCA.provider.IBMJCECCA. However, depending on the algorithms that you use and your ICSF and hardware and software System z configuration, you might obtain some errors.

Equivalent command option: -jce-providers — Specify JCE class names.

Arguments: For hardware cryptographic acceleration that ICSF provides, use the following default value for JCE_PROVIDER_LIST: com.ibm.crypto.hdwrCCA.provider.IBMJCECCA Otherwise, for string, your own value for your hardware provider.

RNG_JCE_PROVIDER *value*

Sets the JCE provider according to the system random number generator.

- For ICSF hardware cryptographic acceleration with an enabled cryptographic module, set the value for RNG_JCE_PROVIDER as follows: com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
- If an ICSF cryptographic module is not enabled, set the value for RNG_JCE_PROVIDER as follows: com.ibm.crypto.provider.IBMJCE If you do not specify a value, the random number generator uses the FIRST provider defined in the JCE provider list.

Default: The first JCE provider in the list.

Equivalent command option: None.

Arguments: For value, the name of a fully-qualified JCE provider class name.

Arguments: For user-specified-comment, a comment string.

Cipher specifications

Some of these options override the preferences, if any, specified in the recipient's certificate.

CIPHER_NAME *algorithm*

When producing an encryption facility message, uses the specified algorithm for encryption.

Default: If the recipient preference is not available in the OpenPGP certificate, TRIPLE_DES.

Equivalent command option: -cipher-name — Specify the algorithm for encryption.

Arguments: For algorithm, specify a valid encryption algorithm. You can run the -list-algo command to see valid algorithm values.

DIGEST_NAME *algorithm*

When producing an encryption facility message, uses the specified algorithm for hashing.

Default: If the recipient preference is not available in the OpenPGP certificate, SHA_1.

Equivalent command option: -digest-name — Specify the algorithm for the message digest.

Arguments: For algorithm, specify a valid digest name for hashing. You can run the -list-algo command to see valid algorithm values.

S2K_CIPHER_NAME *algorithm*

When producing an encryption facility message, uses the specified algorithm for the passphrase-based encryption (PBE). PBE makes use of the hashing value and the encryption of the session key.

Default: If the recipient preference is not available in the OpenPGP certificate, TRIPLE_DES.

Equivalent command option: -s2k-cipher-name — Specify the algorithm to use for passphrase-based encryption (PBE).

Arguments: For algorithm, specify an algorithm for PBE. You can run the -list-algo command to see valid algorithm values.

S2K_DIGEST_NAME *algorithm*

When producing an encryption facility message with passphrase-based encryption (PBE), uses the specified digest algorithm for password based encryption of the session key

Default: If the recipient preference is not available in the OpenPGP certificate, SHA_1.

Equivalent command option: “-s2k-digest-name — Specify the digest algorithm for passphrase-based encryption (PBE)”.

Arguments: For algorithm, specify a digest algorithm. You can run the -list-algo command to see valid algorithm values.

S2K_MODE *value*

When producing an encryption facility message with passphrase-based encryption (PBE), uses a specified value for hashing and encryption of the session key. It is suggested that you

specify 1 or 3. You can only use one password to encrypt the message.

Default: S2K_MODE 3.

Equivalent command option: -s2k-mode — Specify the mode for passphrase-based encryption (PBE).

Arguments: For value, one of the following: 0 Simple, 1 Salted, 3 Salted and iterated

S2K_PASSPHRASE *passphrase*

Description When producing an encryption facility message, use the passphrase for passphrase-based encryption (PBE) or decryption.

Default: None. Prompts for passphrase. You need to run S2K_PASSPHRASE from the UNIX Systems Services environment.

Equivalent command option: -s2k-passphrase — Specify the passphrase to use for passphrase-based encryption (PBE) and decryption.

Arguments: For passphrase, specify a passphrase.

TRUST_VALUE *number*

Species the level of trust for an OpenPGP certificate.

Default: 10.

Equivalent command option: -trust-value — Specify a trust value.

Arguments: For number, you can specify a value from 0 to 255.

TRUSTED_COMMENT *text*

Specifies a comment for the level of trust of an OpenPGP certificate.

Default: Trusted.

Equivalent command option: -trusted-comment — Specify a trust comment.

Arguments: For text, you can specify any comment string.

HARDWARE_KEY_TYPE *type*

Specifies the type of hardware key to generate. The following types are available: PKDS keys are managed by ICSF.

Default: None. If nothing is specified, in a UNIX Systems Services environment, the user is prompted to enter a value.

Equivalent command option: None.

Arguments: For type, one of the following:

- PKDS

- CLEAR

USE_MDC Format USE_MDC

Description: While encrypting data for OpenPGP uses modification detection code (MDC), which species a symmetric integrity protected data packet.

Default: if not specified, do not set.

Equivalent command option: -use-mdc — Specify the use of modification detection code.

Arguments None.

Data recipient specification

RECIPIENT_USER_ID *user-specified-IDs*

Specifies one or more user IDs for the recipients of an encrypted message. Encryption Facility attempts to find the public key for the recipient in the keyring and uses asymmetric encryption of the session key.

Default: None.

Equivalent command option: -rP

Arguments: For user-specified-IDs, one or more user IDs separated by commas.

RECIPIENT_KEY_ID *user-specified-key IDs*

Specifies one or more 8-byte hexadecimal values for the key ID of each recipient of an encrypted message. Encryption Facility attempts to find the public key for the recipient in the keyring and uses asymmetric encryption of the session key.

Default: None.

Equivalent command option: -rK — Encrypt for a specified key ID.

Arguments: For user-specified key-IDs, one or more key IDs separated by commas.

RECIPIENT_ALIAS *user-specified-aliases*

Specifies one or more aliases in the keystore for each recipient of an encrypted message. Encryption Facility attempts to find the public key for the recipient in the keyring and uses asymmetric encryption of the session key.

Default: None.

Equivalent command option: -rA — Encrypt using the public key from the Java keystore.

Arguments: For user-specified-aliases, one or more aliases separated by commas.

Miscellaneous processing options

Some of these options do override the preferences, if any, specified in the recipient's certificate.

USE_ASCII_ARMOR Specifies that when you export an OpenPGP certificate you are to use ASCII Armor.

Default: If not specified, do not use ASCII armor.

Equivalent command option: -a — Use ASCII Armor for the message output.

Arguments: None.

ARMOR_COMMENT *user-specified-comment*

Adds a comment to an OpenPGP certificate that is encoded by ASCII Armor.

Default: None.

Equivalent command option: -comment — Add a comment header to ASCII Armorized messages.

COMPRESSION *value*

Species compression of an encrypted message.

Default: 0.

Equivalent command option: -z — Compress data.

Arguments: For value, a compression value. You can specify one of the following values: 0 Do not use compression. 9 Use the best compression possible. Setting this value can result in a considerable impact to performance. 1 Use the best performance for compression.

COMPRESS_NAME *algorithm*

When producing an encryption facility message, uses the specified algorithm for compression.

Default: If the recipient preference is not available in the OpenPGP certificate, ZIP.

Equivalent command option: -compress-name — Specify the algorithm to use for compression.

Arguments: For algorithm, specify an algorithm for compression. You can run the -list-algo command to see valid algorithm values.

CONFIDENTIAL When processing an OpenPGP message or an Encryption Facility message, does not store the data in the message to a data set or file; instead, sends the data to STDOUT.

Default: If not specified, do not process as confidential.

Equivalent command option: -no-save — Display data to STDOUT only.

Arguments: None.

USE_EMBEDDED_FILENAME *user specified file-name*

When consuming an OpenPGP message or an Encryption Facility message, stores the data in the message to the file or data set that is specified in the message.

If you specify DEFAULT_OUTPUT_DIRECTORY, and the embedded filename does not refer to a data set, Encryption Facility writes the data to this directory.

Default: if not specified, do not use an embedded file name as output.

Equivalent command option: -use-embedded-file — Write data to a file specified in the data packet.

Arguments: For file-name, a file or data set name.

DEFAULT_OUTPUT_DIRECTORY When using embedded filenames and the embedded name does not refer to a data set, stores the data in this directory using the embedded filename.

Default: Current working directory.

Equivalent command option: None.

Arguments None.

ANSWER_YES Assumes yes for yes/no questions.

Default: If not specified, prompt for any yes/no question. You need to run ANSWER_YES from the UNIX Systems Services environment.

Equivalent command option: -yes — Specify yes to prompts.

Arguments None.

ANSWER_NO Assumes no for yes/no questions.

Default: If not specified, prompt for any yes/no question. You need to run ANSWER_NO from the UNIX Systems Services environment.

Equivalent command option: -no — Specify no to prompts.

Arguments None

HIDDEN_PASSWORD Does not display the password in response to the prompt. If you are using a TELNET 3270 session, Encryption Facility displays the password.

Default: if not specified, do not hide responses to password prompts. You need to run HIDDEN_PASSWORD from the UNIX Systems Services environment.

Equivalent command option: None.

Arguments None.

Text characters translation

LITERAL_TEXT_CHARSET *set*

Specifies a character set. Encryption Facility performs character conversions as follows:

- When producing an Encryption Facility message (commands -e, -s, and -c), Encryption Facility converts the data from the system's character set to this value. In addition to the character conversions, Encryption Facility converts end-of-line characters to carriage return and line feed.
- When processing an RFC 2440 message or an Encryption Facility message (commands -d and-v), Encryption Facility converts the data from this value to the system's character set. In addition to the character conversions, Encryption Facility converts end-of-line characters to line feed.
- When creating a detached signature (command -b), Encryption Facility converts the data from the local code page to UTF-8 and uses the UTF-8 characters to calculate or verify the detached signature. (Note that in this instance, the specified value is ignored as the local code page is assumed for the text.) In addition to the character conversions, Encryption Facility converts end-of-line characters to carriage return and line feed.
- When verifying with detached signatures (command -v), Encryption Facility converts the data from this character set to UTF-8 and uses the UTF-8 characters to calculate or verify the detached signature. In addition to the character conversions, Encryption Facility converts end-of-line characters to carriage return and line feed.A value of _LOCAL is equivalent to the system's current character set.

Default: If not specified, the data is processed as binary. If specified without a value, UTF-8 is the default.

Equivalent command option: -t — Treat input as text.

Arguments For charset_name, the name of the character set to use for character conversion. If you specify the string _LOCAL, the command uses the default system code page. If not specified, the data is processed as binary. If specified without a value, UTF-8 is the default. The -list-algo command lists all the available character sets.

Note: When using a z/OS data set as text data input, every new record is treated like a new line in the text data. Conversely, when decrypting text data to a z/OS data set, every new line in the text data forces a new record in the data set.

Debug facility

LOG_FILE *file-name*

Enables logging to a file. You must ensure that ACTIVE_LOGGERS and DEBUG_LEVEL are set for log data to be written. Log output is in XML so the file has an .xml extension.

Default: None. Log not active.

Equivalent command option: -log-file — Write trace information to a file.

Arguments: For file-name, the name of the log file.

CREATE_TRACE Enables the logging of trace and debug information to STDERR. You must ensure that ACTIVE_LOGGERS and DEBUG_LEVEL are set for data to be written.

Default: If not specified, does not display the trace information to STDERR.

Equivalent command option: -debug-on — Activate debugging information.

Arguments: None.

ACTIVE_LOGGERS *value*

When you specify LOG_FILE, CREATE_TRACE, or both, specifies the components that produce debugging and log information. Default: ACTIVE_LOGGERS -1 is initially set in the configuration file and indicates logging for all components. If you do not specify a value, 0 is the default.

Equivalent command option: -debug-level number— Specify a bit mask value for logging.

For value, one of the following component trace options:
- 0 No logging active
- 1 Async facility

- 2 Cipher facility
- 4 Compress facility
- 8 Digital signature facility
- 16 I/O facility
- 32 Message component
- 64 Packet component
- 128 ASCII Armor facility
- 256 Primitives component
- 512 Passphrase-based encryption component
- 1024 General facility
- 2048 Initialization
- 4096 Command processor
- -1 All components

DEBUG_LEVEL *value*

Specifies how debug information is to be collected. For all levels, specify 0.

Default: DEBUG_LEVEL 700 is initially set in the configuration file. If you do not specify a value, 0 is the default.

Equivalent command option: -debug-level level — Specify a level for trace information to be sent to the log file.

Arguments: For value, one of the following options:

- 1000 SEVERE (error information only)
- 900 WARNING and SEVERE (error and warning information)
- 800 WARNING, SEVERE, INFO (error, warning, and informational messages)
- 700 WARNING, SEVERE, INFO, CONFIG (error, warning, informational and configuration messages)
- 500 WARNING, SEVERE, INFO, CONFIG, Fine TRACE/DEBUG (error, warning, informational and configuration messages and fine level of debug tracing)
- 400 WARNING, SEVERE, INFO, CONFIG, Finer TRACE/DEBUG (error, warning, informational and configuration messages and finer level of debug tracing)
- 300 WARNING, SEVERE, INFO, CONFIG, Finest TRACE/DEBUG (error, warning, informational and configuration messages and finest level of debug tracing)
- 0 All

Asynchronous processing facilities

USE_ASYNC_IO Activates asynchronous I/O processing.

Default: If not specified, does not activate.

Equivalent command option: None.

USE_ASYNC_COMPRESS Activates asynchronous compression.

Default: If not specified, does not activate.

Equivalent command option: None.

USE_ASYNC_CIPHER Activates asynchronous encryption.

Default: If not specified, does not activate.

Equivalent command option: None.

C

OpenPGP key exchange and migration

Business exchange processes must define the mechanism for establishing trust among partners. Using cryptography as foundation for trust, the OpenPGP support on z/OS and its use of the Java Cryptography Extension (JCE) within IBM's Java 5 SDK for z/OS, allow for three options:

- Passphrase exchange
- OpenPGP certificate exchange
- X.509 certificate exchange

These three choices each involve the exchange of key material; a passphrase is equivalent to secret key material and must be treated as such. In addition to key exchange among trusted partners, migration of public/private key material from one format to another may be useful. Transfer of public/private key pairs can allow OpenPGP exchanges to leverage existing keys or key migrated from other OpenPGP tools to the implementation provided with IBM's Encryption Facility V1.2.

This appendix will detail the different steps for exchanging OpenPGP certificates, X.509 certificates, and public/private key pairs, using the OpenPGP support and IBM Java 5 SDK tools: **keytool** and **hwkeytool**. Since its main focus is to describe the use of the tools on z/OS passphrase exchange will not be discussed.

Exchanging OpenPGP certificates

OpenPGP certificates encapsulate public key material. In addition to public key material, an OpenPGP certificate can contain additional data including:

- 0 or more User ID Packets – a User ID packet consists of a name, email address, and comment string. The name is the only required information. Its purpose is to identify the entity who owns the certificate. Multiples are allowed since its very common for a person to have multiple email addresses.
- A primary public key – a primary public key must be of a type that can verify signatures, that is, RSA or DSA.
- 0 or more sub public keys – a sub public key can be of any type supported by Encryption Facility: RSA, DSA, ElGamal.
- 0 or more Signature sub-packets – Signature sub-packets encapsulate the results of a signature calculation on different elements of a certificate. They can exist in many different sections of the certificate and may be calculated by anyone's private key or even calculated using the private key pair of the public keys encapsulated within the certificate. Moreover the sub-packet contains information that facilitates the search for the public key needed to verify the packet. This in turn, allows for verification of the calculated signature they represent. Depending on its contents, a Signature sub-packet may be:
  - Asserting the validity of the certificate and any user ID packets included in the certificate. This is paramount for establishing trust.
  - Asserting the expiration date of the certificate
  - Asserting the revocation of the certificate

Exporting OpenPGP certificates

Encryption facility's OpenPGP support provides three commands that export public key material in OpenPGP certificates: -eA, -eP, -eK.

The –eA command interactively creates a new OpenPGP certificate, using the public key material of the key in the Java keystore whose alias is specified as a command line parameter. (See the **JAVA_KEY_STORE_NAME** and **JAVA_KEY_STORE_TYPE** keywords in the configuration file in Appendix B, "Configuration file options" on page 237 for information on how to specify the location of the Java keystore file and its type). Figure C-1 on page 255 is an example of an –eA invocation. This invocation was executed from a USS shell.

```
/home/suimgwi/>java -jar CSDEncryptionFacility.jar -homedir . -a -o openpgp.asc
-eA saheemRSA
CSD0029I Exporting an OpenPGP certificate for saheemRSA...
CSD0026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email
address(optional).
CSD0020A Real Name:
redbooks 1
CSD0024A Comment:
redbooks1 example
CSD0022A Email address:
redbooks@us.ibm.com
CSD0025A You specified user ID: "redbooks 1 <redbooks1 example>
<redbooks@us.ibm.com>"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSD0027A Add another user ID? (yes/no)
n
CSD0019A For how many days should this OpenPGP certificate be valid (0 for
always valid):[0]
0
CSD0032A Do you want to add the exported OpenPGP Certificate to your OpenPGP
key ring? (yes/no)
Yes
CSD0075A Certificate <4FA96D749D661CE2> already exists in the key ring. Would
you like to replace it? (yes/no)
Yes
CSD1347I 1 OpenPGP certificate(s) were exported successfully to openpgp.asc.
CSD0051I Command processing has completed successfully.
```

Figure C-1 Exporting public key sample

The file **openpgp.asc** was created and now may be exchanged with a business partner. This command also results in a copy of the newly created OpenPGP certificate being stored in the OpenPGP keyring file used by the OpenPGP support. (See **KEY_RING_FILENAME** keyword in the configuration file described in Appendix B, “Configuration file options” on page 237 for information on how to specify the location of the keyring file.)

The –eP command exports certificates that exist in the OpenPGP keyring file. It relies on a case-insensitive, substring match of a User ID packet. Figure C-2 on page 256 is an example for searching the keyring for the OpenPGP certificate created by the result of the –eA command.

```
/home/suimgwi/>java -jar CSDEncryptionFacility.jar -homedir . -a -o openpgp.asc
-eP redbooks
CSD0700A File <openpgp.asc> already exists.  Do you want to overwrite? (yes/no)
Yes
CSD0029I Exporting an OpenPGP certificate for "redbooks"...
CSD1347I 1 OpenPGP certificate(s) were exported successfully to openpgp.asc.
CSD0051I Command processing has completed successfully.
```

Figure C-2 Exporting certificate sample

The –eK command relies on the Key ID of the public key (primary or sub in the case of OpenPGP certificates) associated to a X.509 or OpenPGP certificate to locate the public key material to export. Key IDs are an 8 byte number, specified in hexadecimal (using 16 characters.). The OpenPGP support firsts searches the OpenPGP keyring for a certificate that encapsulates the public key who's calculated key ID matches the specified command line option. If no key is found in the OpenPGP key ring, the keystore is searched. The –pK, -pA, and –pP display commands display key information including the Key ID. Figure C-3 is an example of –pA that shows the Key ID of the key used for the first export example:

```
/home/suimgwi/>java -jar CSDEncryptionFacility.jar -homedir . -a -o openpgp.asc
-pA saheemRSA
CSD1352I Displaying certificate with alias: saheemRSA
dn: CN=Saheem Granados, OU=zos-security, O=IBM, L=Poughkeepsie, ST=NY, C=US
primary: RSA 1,024  key ID: 4FA96D749D661CE2  fingerprint:
15A586227F82F9F705BFAB414FA96D749D661CE2  created: 1/20/06 [expired: 4/20/06]
```

Figure C-3 Display key information sample

Now, Figure C-4 is an example of –eK:

```
/home/suimgwi/>java -jar CSDEncryptionFacility.jar -homedir . -a -o openpgp.asc
-eK 4FA96D749D661CE2
CSD0700A File <openpgp.asc> already exists.  Do you want to overwrite? (yes/no)
Yes
CSD0029I Exporting an OpenPGP certificate for 4FA96D749D661CE2...
CSD1347I 1 OpenPGP certificate(s) were exported successfully to openpgp.asc.
CSD0051I Command processing has completed successfully.
```

Figure C-4 Sample of -eK command to export certificate

The OpenPGP support located the certificate in the OpenPGP keyring.

Importing OpenPGP certificates

Encryption Facility's OpenPGP support provides the –i command to import OpenPGP certificates. The certificates may be in ASCII armor form or binary form.

Note: A certificate file kept in z/OS in the ASCII armor form has actually an EBCDIC encoded content that displays the same as when the certificate is displayed on an ASCII based platform.

During import, the OpenPGP support attempts to verify any signature sub-packets it encounters. If it encounters a sub-packet that requires a public key that is not readily available in its OpenPGP keyring or Java keystore, it warns the user and asks for permission to continue the import. In addition, if the certificate contains signature sub-packets that assert that the certificate has been revoked or expired, it warns the user and asks for permission to continue the import. Finally, if key material enclosed within the certificate exists in another OpenPGP certificate within the invocation's OpenPGP key ring, it asks if it should replace the existing certificate with certificate to import. Figure C-5 is an example of an import command:

```
/home/suimgwi/>java-jarCSDEncryptionFacility.jar-homedir.-ieswpluto2.asc
CSD0038I Importing OpenPGP certificate with primary DSA public key ID:
B22653E6B463DC6C...
CSD0075A Certificate <B22653E6B463DC6C> already exists in the key ring. Would
you like to replace it? (yes/no)
Yes
CSD0073I Total number of certificates imported successfully: 1
CSD0051I Command processing has completed successfully.
```

Figure C-5 Importing certificate sample

Exchanging X.509 certificates

Encryption Facility's OpenPGP support does not provide commands for exchanging X.509 certificates. Fortunately, since it relies on Java and the JCE framework, two tools are readily available to work with X.509 certificates: **keytool** and **hwkeytool**.

Note: Java tools currently do not import certificates into Java keystores that are based on the z/OS Security Server, that is, keystore types JCERACFKS and JCECCARACFKS. These keystores are"read-only" keystores from the Java framework perspective. Their contents can only modified using commands provided in the Security Server. In the case of RACF, refer to RACF publications for RACF commands that import X.509 certificates. For other security server products, refer to the appropriate documentation for availability of commands and/or interfaces for importing X.509 certificates.

Using keytool with X.509 certificates

keytool should be used with the JCEKS and JKS keystore types. Figure C-6 shows keytool being used on a distributed platform to export an X.509 certificate:

```
keytool.exe -export -keystore keytool.jceks -storepass TKETKE -storetype jceks
-alias first -v -rfc > first.export.asc
```

Figure C-6 Keytool syntax

The file **first.export.asc** is transferred to the z/OS host. Note unlike ASCII armor with OpenPGP certificates, the file which contains a DER base64 encoded certificate must be in ASCII on z/OS (that is must be kept in its binary form, meaning that transfers implying text translation must not be used). Figure C-7 shows the import of a self-signed X.509 certificate:

```
/home/suimgwi/>keytool -import -rfc -alias firstimportRSA -keystore ibmpgp.jks
-storetype jks -storepass TKETKE -file first.export>
Owner: CN=Saheem Granados, OU=zOSRSA, O=STG, L=Pok, ST=NY, C=US
Issuer: CN=Saheem Granados, OU=zOSRSA, O=STG, L=Pok, ST=NY, C=US
Serial number: 46a6474b
Valid from: 7/24/07 2:39 PM until: 10/22/07 2:39 PM
Certificate fingerprints:
         MD5:  76:5C:DA:34:B1:A2:F6:53:1B:E3:46:58:89:0B:97:6F
         SHA1: AF:01:58:14:F6:17:A5:49:E4:F6:2D:7E:15:44:26:C9:F8:02:7C:D0
Trust this certificate? [no]: yes
Certificate was added to keystore
```

Figure C-7 Import of self-signed certificate sample

Now, the OpenPGP support can use this certificate to encrypt data. The –rA option specifies the imported X.509 certificate and the –e command forces the OpenPGP support to use public key based encryption.

```
/home/suimgwi/>java -jar CSDEncryptionFacility.jar -homedir . -rA
firstimportRSA -o enc.out.imported -e PERSON1.SMALL.LDIF
CSD0700A File <enc.out.imported> already exists.  Do you want to overwrite?
(yes/no)
Yes
CSD1333A Alias firstimportRSA refers to an X.509 certificate that is not valid.
Do you want to continue? (yes/no)
Yes
CSD0768I Output data can be exchanged with the owner of the key with key ID
65BB2F961C557DEF.
CSD0051I Command processing has completed successfully.
```

Figure C-8 Sample of -rA option

Using hwkeytool with X.509 Certificates

Use of **hwkeytool** for importing X.509 certificates is very similar to using **keytool:**

```
hwkeytool -import -rfc -alias firstimportRSA -keystore cca2.jks -storetype
JCECCAKS -storepass TKETKE -file first.export.asc -provider
com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
Owner: CN=Saheem Granados, OU=zOSRSA, O=STG, L=Pok, ST=NY, C=US
Issuer: CN=Saheem Granados, OU=zOSRSA, O=STG, L=Pok, ST=NY, C=US
Serial number: 46a6474b
Valid from: Tue Jul 24 14:39:07 EDT 2007 until: Mon Oct 22 14:39:07 EDT 2007
Certificate fingerprints:
         MD5:  76:5C:DA:34:B1:A2:F6:53:1B:E3:46:58:89:0B:97:6F
         SHA1: AF:01:58:14:F6:17:A5:49:E4:F6:2D:7E:15:44:26:C9:F8:02:7C:D0
Trust this certificate? [no]:  y
Certificate was added to keystore
```

Figure C-9 hwkeytool sample

The main difference is due to the extra parameter and a different store type value needed by **hwkeytool.** The –storetype is JCECCAKS. This type is required for using ICSF/Hardware keys. The provider parameter is needed to enable the IBM CCA JCE provider; similar to the OpenPGP support's **JCE_PROVIDER_LIST** configuration keyword, the -provider option enables the JCE provider that leverages ICSF and hardware cryptography.

Considerations on certificate exchange

Exchanging certificates is an essential task for establishing trust among trusted business partners. The OpenPGP support and IBM's Java SDK provide the opportunity to finalize the exchange. Many examples were shown in this section, demonstrating the mechanics of exporting and importing certificates. Aside from these tasks, users must understand that using the tools may not be sufficient for establishing trust. In all the examples shown here, no certificate authority was used for the X.509 certificates. Moreover, OpenPGP certificates use a decentralized approach for trust verification of certificates. Without a trusted certificate authority, users must carefully inspect certificates and establish out of band procedures for accepting the validity of a certificate. If the public keys are available, the OpenPGP support verifies all signature information in the certificate. This ensures the certificate has not been manipulated. However, users still may need to use Key IDs, fingerprints, visual, and verbal verification with partners to really cement certificate based trust.

Migrating key pairs

Exchanging public key material is a well documented endeavor. The public nature of the key material and use of digital certificates and digital signatures reduce the security requirements and thus the complexity needed for transfer. (Note as previously stated, the benefits of public key exchange do not necessarily mean that establishing trust is a trivial task). Migration of key material that is re-formatting a key so that it could still be usable in a different standard, is a completely different matter.

Migration of key pairs implies that the owner of a public/private key pair would like to continue to use the same key pair in their OpenPGP environment. Many times these key pairs were generated for use by other cryptographic tools, or even other OpenPGP tools. The OpenPGP support in Encryption Facility does not provide any direct support for transfer of public/private key pairs. (Note however that RFC 2440 describes Secret Packets that can serve as a defined syntax for transferring private keys in the OpenPGP space). Fortunately, the **keytool** and **hwkeytool** tools allow for exchange of private keys using the PKCS#12 format.

The use of PKCS#12 format provides many benefits:

- RACF security server allows for importing of password protected PKCS#12 format files - Refer to RACF publications for importing of password protected PKCS#12 data
- IBM Java SDK on z/OS tools allow for imported private keys to be stored in the ICSF PKDS

- Migration from other OpenPGP tools that provide export of public/private keys in PKCS#12 syntax

Figure C-10 shows exporting a public/private key pair using keytool on a distributed platform:

```
$ keytool.exe -export -keystore keytool.jceks -storepass TKETKE -storetype
jceks -alias firstRSA -rfc -v -pkcs12  > first.export.asc
Enter key password for <firstRSA>TKETKE
```

Figure C-10 Exporting public/private key on distributed platform

After the key has been exported, it must be transferred to z/OS. Once again, it must remain in binary format, without any attempt to perform text translation. Here it can be imported into the PKDS:

```
/home/suimgwi/>hwkeytool -import -rfc -alias firstimportCCARSA -keystore
cca2.jks -storetype JCECCAKS -storepass TKETKE -file first.export.asc -provider
com.ibm.crypto.hdwrCCA.provider.IBMJCECCA -pkcs12 -hardwaretype PKDS
Enter key password for <firstimportCCARSA>TKETKE
Content of pkcs12 file was imported in keystore
```

Figure C-11 Importing the keys into PKDS

An additional option was provided on the **hwkeytool** invocation from the above example. The –hardwaretype option allows for the user to specify where the key should be stored. The options are RETAINED, CLEAR, or PKDS. RETAINED stores the key material in the Crypto Coprocessor and is not recommended. CLEAR stores the key material in the Java keystore protected by the security semantics of Java keystores. Finally, the PKDS option stores the key in the ICSF PKDS.

Once imported, the key can now be exported as an OpenPGP certificate:

```
/home/suimgwi/>java -jar CSDEncryptionFacility.jar -homedir . -a -o pkcs12.as
-eA firstimportCCARSA
CSD0029I Exporting an OpenPGP certificate for firstimportCCARSA...
CSD0026I At least one user ID is required for an OpenPGP certificate.
A user ID consists of three parts: a name, a comment(optional), and an email
address(optional).
CSD0020A Real Name:
redbooks pkcs12
CSD0024A Comment:
pkcs12 ecxample
CSD0022A Email address:
pkcs12@us.ibm.com
CSD0025A You specified user ID: "redbooks pkcs12 <pkcs12 ecxample>
<pkcs12@us.ibm.com>"
Change (N)ame, (C)omment, (E)mail, (X)Cancel or (O)kay to accept?
o
CSD0027A Add another user ID? (yes/no)
n
CSD0019A For how many days should this OpenPGP certificate be valid (0 for
always valid):[0]

CSD0032A Do you want to add the exported OpenPGP Certificate to your OpenPGP
key ring? (yes/no)
Yes
CSD1347I 1 OpenPGP certificate(s) were exported successfully to pkcs12.as.
CSD0051I Command processing has completed successfully.
```

Figure C-12 Exporting the key as OpenPGP certificate

Using GNU's Privacy Guard (gpg) on distributed, the OpenPGP certificate is imported:

```
$ gpg --import pkcs12.as
gpg: WARNING: using insecure memory!
gpg: please see http://www.gnupg.org/faq.html for more information
gpg: key 1C557DEF: public key "redbooks pkcs12 <pkcs12ecxample>
<pkcs12@us.ibm.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Figure C-13 Importing the OpenPGP certificate

Now, gpg can be used to encrypt data as shown in Figure C-14 (note that, as an exemple, we are encrypting here the PKCS#12 file that was initially generated on the distributed platform).:

```
$ gpg –t -o encrypted.gpg -r redbooks -e first.export.asc
gpg: WARNING: using insecure memory!
gpg: please see http://www.gnupg.org/faq.html for more information
gpg: 1C557DEF: There is no assurance this key belongs to the named user

pub  1024R/1C557DEF 2007-07-24 redbooks pkcs12 <pkcs12 ecxample> <pkcs12@us.ibm
com>
 Primary key fingerprint: 6802 99C4 7F22 F176 0CD0  9612 65BB 2F96 1C55 7DEF

It is NOT certain that the key belongs to the person named
in the user ID.  If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y
```

Figure C-14 [1]Encrypting the data

Now the data has been encrypted using gpg, the OpenPGP support in Encryption Facility must decrypt the data using the imported private key:

[1] Note the gpg output warns for acceptance of trust.

```
/home/suimgwi/>java -jar CSDEncryptionFacility.jar -homedir .  -o decrypto.out
-d encrypted.gpg
CSDO051I Command processing has completed successfully.
/home/suimgwi/>cat decrypto.out
-----BEGIN CERTIFICATE-----
MIIGpwIBAzCCBmEGCSqGSIb3DQEHAaCCBlIEggZOMIIGSjCCAycGCSqGSIb3DQEHAaCCAxgEggMU
MIIDEDCCAwwGCyqGSIb3DQEMCgECoIICsTCCAqOwJwYKKoZIhvcNAQwBAzAZBBT43vBoroGJi8V7
3f7vNj8L9uhmiAIBCgSCAoAYF4gkEqvWVdDZbAgirzyFXiJRF5+Sws4hZ4OslIeMwFULRJLsI2zY
/TWA8GtzF5WhKD38+ccXD3xCLen54MgGm+IJrYVhM2ovmGIm6SIVVue4mkgOBuzp2+BumIOeo/qS
J6YmL+8scT/kjn3gV1S+QqXfUeGOWzFHvHvqa7uKTdMeJ2SCcwr6SWFOaE3smZCVUkaTd1DWmr96
8WO4L5JTz6P84bfaz+QugHGSgtwmdOSmqqO9oXt5f89jng7+gK2OWrOVKxbxo37vNA2z1bNLJdDk
Ujsflk5MBjkqTVvd79IR4QWs1iJeFJTYkWiuuWl9MNkOOxh6f3+k2ak+V13upctevnCUh9SjCylr
ny+gUc7mJNKqvJ17xLEf3yqxOB5phxrrQMVxYHCmzTytAjOOfd8omT6VPoRwEpQCbU96zvoNPvmE
1ZmIfuMsGOIFKPc+trZRcLzOeWUfV2Hkw3tGoDqofdWQjLmxyL78ARNXBU5adN/hn11XegmZ8OcA
ZWtjj8sugw/p127hvZsrX9wHQDYFHdle1F7vf5/ODGsiCOkkpXeuMADaXo/2u7zSYwAnBZ597Tso
7cdeCXH6bndO6CqAsO5HuWMc59iy6D5CYFVNS377HrVpyneCgkZCtd2FKU6XcxC1Oamw5uMYl8zR
HdAXvpFitCawJIjsORd1Jr32BO4EqreMFwj/kU5biUNz2hfbPmZe5GY4myBaMB6YkhFnIITq1bmJ
LOXA8TY7XynGKVUbckHErb3eZj6ckH++5JROBYmZfYBAbrK/ELkcEa3mIwG8QVwaZY2RB6XwWpuW
VRg23/RvOCSOrFJXd+O4fCyaYv4hBhNmISstfRV1MUgwIQYJKoZIhvcNAQkUMRQeEgBmAGkAcgBz
AHQAUgBTAEEAADAjBgkqhkiG9w0BCRUxFgQUH8LM4QXPP9M9U7CTRj5OS4hSljgwggMbBgkqhkiG
9w0BBwagggMMMIIDCAIBADCCAwEGCSqGSIb3DQEHATAoBgoqhkiG9w0BDAEDMBoEFAmlNfckUNZ1
fkjMwL4+cmjwS+HaAgIHOICCAsh1/UAK76sobtEXUseXYAkO8DcaeFicUy/tUD/NWr4mmKl8dgn8
kQ7HCyzMsXEOXVQUKDFzfyprRXxmAdIZiIkXkHAJkFs8s98vOZAIhfRn79Z5hBwIuN7kwDUNW4Sn
QxKWHtNDSUQjL5uu5liosNz45lVBft75RHahOzzDoZVE+8wyJ+26tlWAHPOdvqY1dUZpn5KSHrBw
I8S41bztDnBex5jQqY3Jd6h+GkqnZ31KZDbAWWSgvA5W8KgqY3FG7rfe+2PMh2LLIsN6DymO6Ke8
w1rpgzAWVO8Kur121GrYbgGOqqy+pWUZjDC5TQjFaRJVoif1a48N15xBr1kDqRr9uQQcWI6cyDQ8
EcBipkbXPyAVTLQDGNthRnBYRO6lOW3X7GaODyIVJUZszgR/t6LOApjxMjp9RH7O+F/mSRcvNLVA
GEdxL/9rhDlOLZLQhacExp9B9LLrcJU/y+RAVloUdHGGWqRfO9nOafnA5iad7rRqiwu6npplccpg
3p8xjigxSR9/AhpHXpPlA9HHcYh4hv3qmlLaMlTHR8yIUNDQFFXgG43NFx8Eiex72wMf5OeEonsa
upPle+Lr+txg2AXLy6dtrlQPg38RQEzLfYrQQ5yhaEhBivPgkiOMa/aM/H5JzP1997hGl137ASsr
5/bKFtvWJz8gBi2FH2+O6V9QB4MIVJjMFvQyuwOvVf2FOADAuUia2mpvF3EjaoogJKKcFBnlgxmP
GCI51LqX/LqqfR3QzUVVQBxDSuyNKUUA6lFpNiRlF1ZS1/z/JlSaiWdjb6OUzJC7WudMhp1Eiej8
NI47oebNgLgVBZ5z2Io7gdHJX7hRlnIZcr2DXPDAawNvnB44JvqQv3GbxaN3rDBNjSZrhCidW56+
g1HPMF/TJejowOW1QPdGNAd7gJf/YGDHWpFCP/k1yZBBHue7ZoXLWsDmvq6ByvyCMDOwITAJBgUr
DgMCGgUABBRbQ2u2yqfGrOqOXrwCZOYe72ZdbwQUTP622khWDXz527doycMwFHYiTm8CAgfQ
-----END CERTIFICATE-----
```

Figure C-15 Decrypting the data using the imported private key

The output includes the contents of the decrypted data for completeness.

Considerations on key pair migration

Key migration may be useful in certain environments. Although, IBM's current set of tools do not provide a direct OpenPGP exchange of private key material, the **hwkeytool** and **keytool** tools provide a means for migrating key pairs using X.509 and PKCS#12 formats. **hwkeytool** provides a means for storing the migrated key material in the ICSF PKDS. The RACF security server also provides a means for importing password protected PKCS#12 files. Both serve as very secure repositories for storing cryptographic keys and certificates. One essential point that must be stressed is that transport of private key material must be done with extreme caution.

D

Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG247434

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbook form number, SG24-7434.

Using the Web material

The additional Web material that accompanies this book includes the following files:

File name *Description*

SG247434.zip	Zipped Code Samples including the ear files and ldap schemas

Ldap schemas provided as courtes of PGP Corporation. Reprinted by Permission.

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. Follow the installation instruction described in “Installing and configuring the IIPKS” on page 170.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks publications” on page 270. Note that some of the documents referenced here may be available in softcopy only.

- *Encryption Facility for z/OS v1.10*, SG24-7318
- *Implementing PKI Services on z/OS*, SG24-6968

Other publications

These publications are also relevant as further information sources:

- *IBM Encryption Facility for z/OS: Planning and Customizing,* SA23-2229
- *Program Directory for IBM Encryption Facility for z/OS V1.2.0*, GI10-0771
- *IBM Encryption Facility for z/OS: Using Encryption Facility for OpenPGP*, SA23-2230
- *z/OS Cryptographic Services 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

Online resources

These Web sites are also relevant as further information sources:

- RFC 2440

  `http://www.ietf.org/rfc/rfc2440.txt`
- IBM Java cryptographic hardware service providers

http://www.ibm.com/servers/eserver/zseries/software/java/j5jcecca.html

- Java keytool utility

  http://www.ibm.com/developerworks/java/jdk/security/142/secguides/keytoolDocs/KeyToolUserGuide-142.html

- Java hwkeytool utility

  http://www.ibm.com/servers/eserver/zseries/software/java/hwkeytool.html

- Encryption Facility for z/OS Java Client:

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

- IBMJCECCA provider:

  http://www.ibm.com/servers/eserver/zseries/software/java/j5jcecca.html

- *Unrestricted policy file* ca be downloaded from:

  http://www.ibm.com/servers/eserver/zseries/software/java/j5jcecca.html

- ICSF PKDS Key Management enhancement:

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

How to get IBM Redbooks publications

You can search for, view, or download Redbooks, IBM Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Abbreviations and acronyms

AES	Advanced Encryption Standard
API	Application Programming Interface
ASCII Armor	encoding of binary data in a sequence of ASCII printable characters
Blowfish	Symmetric block cipher used as a drop-in replacement for DES or IDEA. It takes a variable-length key, from 32 bits to 448 bits. Designed in 1993 by Bruce Schneier as a fast, free alternative to existing encryption algorithms. Unpatented and license-free.
CA	Certificate Authority
CCA	Common Cryptographic Architecture
CCF	Cryptographic Coprocessor Facility
CEX2C	Crypto Express2 Coprocessor
CPACF	CP Assist for Cryptographic Function
CTO	Chief Technology Officer
DCB	Data Control Block
DDNAME	Data Definition Name
DER	Distinguished Encoding Rule
DES	Data Encryption Standard
DFSMS	Data Facility Storage Management
DFSMSdss	DFSMS Data Set Services - see also DFSMS
DFSMSShsm	DFSMS Hierarchical Storage Management - see also DFSMS
DSA	Digital Signature Algorithm
ElGamal	ElGamal algorithm - an asymmetric key encryption algorithm for public key cryptography based on Diffie-Hellman key agreement.Described by Taher Elgamal in 1984. Used in the free GNU Privacy Guard software, recent versions of PGP, and other cryptosystems.
HFS	Hierarchical File System
HKP	HTTP Keyserver Protocol
HMAC	Hash-based Machine Authentication Code
IBM	International Business Machines Corporation
ICSF	Integrated Cryptographic Services Facility
IETF	Internet Engineering Task Force www.ietf.org
ISOC	Internet SOCiety www.isoc.org
ITSO	International Technical Support Organization
JCE	Java Cryptographic Extension
JCL	Job Control Language
JDK	Java Development Kit
JES	Job Entry Subsystem
JNI	Java Native Interface
JRIO	Java Record I/O
JSSE	Java Secure Socket Extension

JVM	Java Virtual Machine
KSDS	Key Sequenced Data Set
OAEP	Optimal Asymmetric Encryption Padding
OpenPGP	Open Pretty Good Privacy
PBE	Passphrase based encryption
PCICC	PCI Cryptographic Coprocessor
PCIXCC	PCIX Cryptographic Coprocessor
PDS	Partitioned Data Set
PDSE	Partitioned Data Set Extended
PKDS	Public Key Crypstographic Data Set
PKI	Public key infrastructure
RACF	Resource Access Control Facility
RFC	
RSA	Algorithm publicly described in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman at MIT; the letters RSA are the initials of their surnames.
SDK	Software Development Kit
SSL	Secure Sockets Layer
TDES	Triple DES - see DES
TLS	Transport Layer Security
USS	UNIX System Service
VPN	Virtual Private Network
VSAM	Virtual Sequential Access Method
WTO	Write to operator
zAAP	System z Application Assist Processor
zFS	zSeries File System

Index

F

H

I

J

K

L

M

O

P

R

S

T

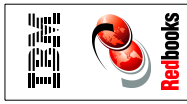
U

V

W

X

Z



Encryption Facility for z/OS V1.2 OpenPGP Support

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

Encryption Facility for z/OS V1.2 OpenPGP Support

Introduction to OpenPGP and review of cryptography concepts

Expert guidance to achieve high security and high performance

Detailed implementation procedures and practical usage scenarios

This book is about the OpenPGP support available in the Encryption Facility for z/OS V1.2 (Program Product 5655-P97). It begins with a discussion of the principles of operation of the OpenPGP protocol and a review of some basic cryptographic topics. It presents detailed steps for installing and configuring Encryption Facility for z/OS and implementing OpenPGP support.

Practical examples from our lab and from the authors' real world experiences demonstrate how to set up and use the capabilities of OpenPGP support. The options available within the product are discussed, and recommendations for appropriate selections within the context of your intended use are offered.

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

SG24-7434-00 ISBN 0738489182