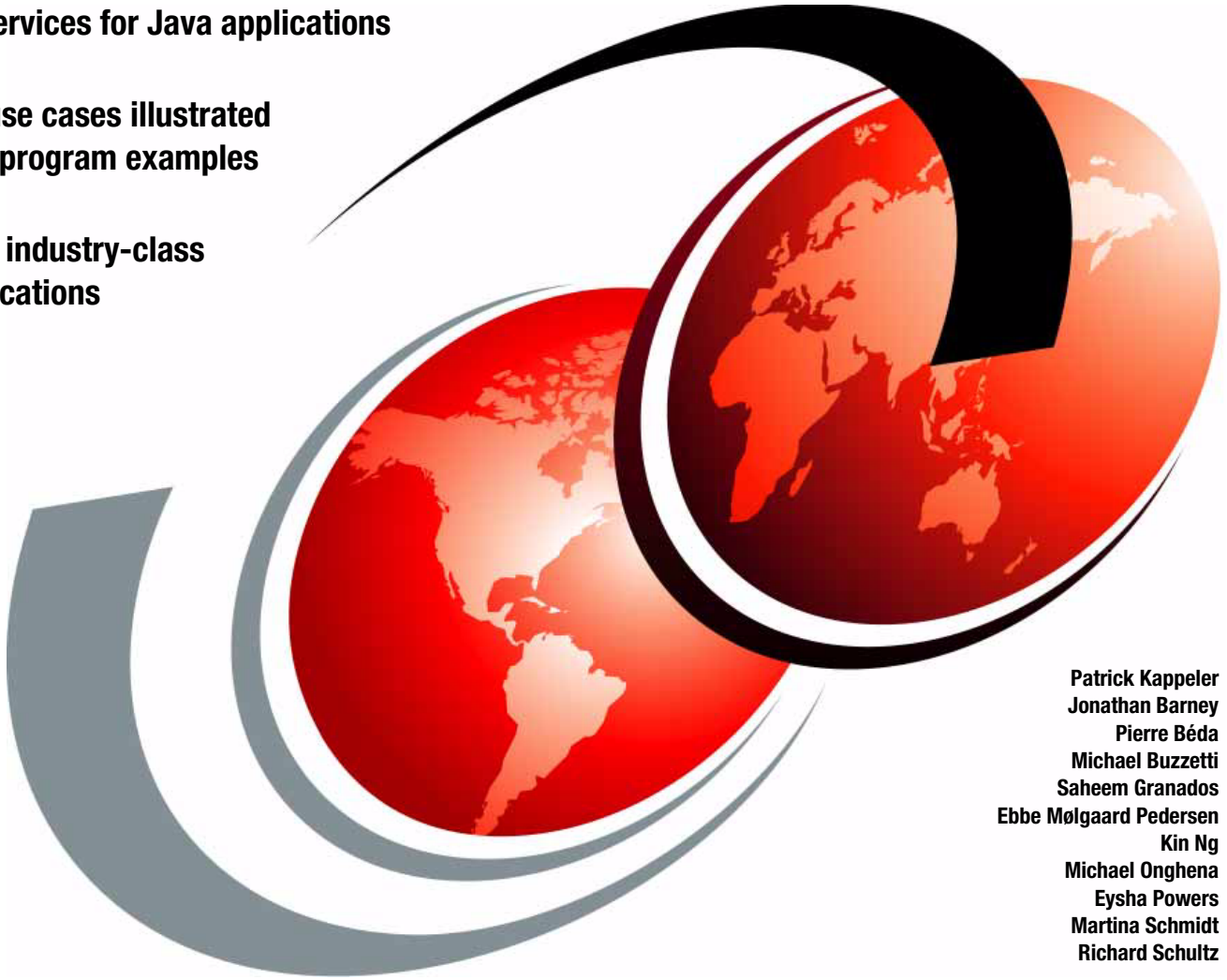
IBM

Java Security on z/OS - The Complete View

- Comprehensively describes z/OS security services for Java applications
- Provides use cases illustrated with Java program examples
- Discusses industry-class Java applications



Patrick Kappeler
Jonathan Barney
Pierre Béda
Michael Buzzetti
Saheem Granados
Ebbe Mølgaard Pedersen
Kin Ng
Michael Onghena
Eysha Powers
Martina Schmidt
Richard Schultz

Redbooks

ibm.com/redbooks

International Technical Support Organization

Java Security on z/OS - The Complete View

December 2008

SG24-7610-00

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

First Edition (December 2008)

This edition applies to Version 1, Release 10 of z/OS (Program Number 5694-A01).

© Copyright International Business Machines Corporation 2008. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2008. All rights reserved.

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.

© Copyright IBM Corp. 2008. All rights reserved.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

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

AIX®
CICS®
DB2®
eServer™
i5/OS®
IBM®
Language Environment®
Lotus Notes®
Lotus®
MVS™
Net.Commerce™
NetView®
Notes®
OS/390®
OS/400®
Processor Resource/Systems Manager™
RACF®
Redbooks®
Redbooks (logo) ®
REXX™
S/370™
System Storage™
System z10™
System z9®
System z®
System/370™
Tivoli®
WebSphere®
xSeries®
z/OS®
z10™
z10 BC™
z10 EC™
z9®
zSeries®

The following terms are trademarks of other companies:

Novell, the Novell logo, and the N logo are registered trademarks of Novell, Inc. in the United States and other countries.

EJB, J2EE, Java, Javadoc, JDK, JNI, JRE, JVM, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

ESP, 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.

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

Preface

This IBM® Redbooks® publication describes and explains which z/OS® security services can be exploited by Java™ stand-alone applications executing on z/OS. It is intended for experienced z/OS users with a moderate knowledge of Java, and experienced Java users with some knowledge of z/OS. For experimentation and customization, it provides use cases that were composed and tested on a z/OS platform at z/OS V1R10 and SDK 6 SR1.

The book describes the Java security model as implemented in the z/OS JVM™, and explains the role of the major infrastructure components such as Security Manager, Access Controller, Class Loader and Byte Code Verifier. It also addresses specific z/OS-provided facilities such as the JZOS toolkit and the Java record I/O (JRIO), and explains how they fit within both security models.

A full chapter is dedicated to Java Authentication and Authorization Services (JAAS) with practical examples of its use in z/OS, including the LoginModules that interact with the SAF interface. A discussion of the relationship of these services to the z/OS built-in security functions such as APF, Program Control, and so on is also provided.

The book then addresses the specific security-relevant services that are provided to Java applications executing on the z/OS platform, and gives practical examples of their setup and use. Java SAF classes, the JSec API, exploitation of RACF® PassTickets, and the use of the z/OS Enterprise Identity Mapping (EIM) infrastructure are covered.

Exploitation of z/OS integrated hardware cryptography by Java applications is detailed, along with numerous practical examples of the use of these services. z/OS cryptographic key management features are also discussed. Finally, the book discusses two industry-class IBM Java products that exploit z/OS hardware cryptography, IBM Encryption Key Manager and IBM Encryption Facility for z/OS OpenPGP Support. It focuses on the exploited functionalities and performance optimization.

It is strongly recommended that readers also refer to the following Redbooks for additional information about executing Java stand-alone applications on z/OS:

- *Java Stand-alone Applications on z/OS Volume 1*, SG24-7177
- *Java Stand-alone Applications on z/OS Volume 2*, SG24-7291

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 led this Redbooks project. He is a System z® Security Specialist working both for the Montpellier European Products and Solutions Support Center (PSSC) and the International Technical Support Organization (ITSO) Center in Poughkeepsie, USA. He holds a French Air Force Ecole Technique degree in Electronics and Computer Science. He joined IBM in 1970 and held many international positions dealing with mainframe hardware and software technical support and education during his 38-year career. Patrick is now an IBM Consulting IT Specialist on mainframe e-business Security, and writes, presents, and consults extensively on this topic around the world.

Jonathan Barney is an Enterprise Security Architect in the United States. He has seven years of experience in z/OS, Java, and UNIX® System Services. He holds a Bachelor of

© Copyright IBM Corp. 2008. All rights reserved.

Science degree in Computer Science from Clarkson University. Jonathan's areas of expertise include Java, RACF Java security, tape encryption, host encryption facility, the WebSphere® family of products, and the PKI and OpenPGP systems.

Pierre Béda is an IT Specialist in the Banking Competence Center (BCC) in Switzerland. He has more than 14 years of experience in mainframe security. He joined IBM in 2007. Pierre's areas of expertise include RACF, security administration, and z/OS and storage administration.

Michael Buzzetti is an IT Specialist at the IBM Design Center in Poughkeepsie, New York. He specializes in virtualization and security. He is a vocal advocate of Open Source and Linux®. Mike holds a Bachelor of Science degree in Computer Science from Clarkson University.

Saheem Granados is an Advisory Software Engineer who has worked for IBM for 10 years. He is a Certified Information Systems Security Professional who 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, Trusted Key Entry, and Encryption Facility V1.2. Saheem has spent a majority of his career focusing on Java-based security on z/OS. He recently rejoined the Tivoli Directory Server for z/OS team.

Ebbe Mølgaard Pedersen is an IT Architect from Denmark. He has 14 years of experience within IBM, developing and architecting applications for z/Series, mainly for the banking industry. Ebbe's areas of expertise include z/OS, DB2®, AIX®, C, C++, Java, application architecting and application developing.

Kin Ng is a Software Developer in Poughkeepsie, NY. He has more than 25 years of experience with large systems software. He has worked on TSO/E development, and was part of the team that contributed to the original releases of APPC/MVS™ and z/WLM. Kin has contributed to many releases of z/WLM and was part of the EWLM development team. He has worked in the z/OS Java security area for the past year.

Michael Onghena is an Advisory Software Engineer in zSeries® Security at IBM. He is a developer for RACF and has been with IBM for 16 years.

Eysha Powers is a Software Engineer in the United States. She has been with IBM for more than four years. Eysha holds a Bachelor of Science degree in Computer Science from the University of Illinois at Urbana-Champaign and a Master of Science degree in Information Technology from Rensselaer Polytechnic Institute. Her areas of expertise include RACF, EIM, and Java Cryptography.

Martina Schmidt is a Technical Presales Specialist in Germany. She has two years of experience in mainframes and holds a Bachelor in Applied Computer Science degree from the University of Cooperative Education in Stuttgart. Martina's areas of expertise include Java, Java batch and WebSphere Application Server on z/OS, with a focus on performance and security.

Richard Schultz has worked for IBM for 29 years. He has held positions dealing with mainframe hardware and software technical support, with a concentration in performance measurement and analysis. For the past six years, he has specialized in performance evaluation of z/OS middleware cryptographic products. Richard provides advanced technical support on this topic internally, and provides data on this topic that is published for customers worldwide.

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

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

The z-Delivery team
Banking Competence Center (BCC), Switzerland

Anne Emerick
IBM STG z/OS Security Server Development, Poughkeepsie

Brian Beegle
IBM STG z/OS Java Development, Endicott

Daniel Dranchak
IBM STG z/OS Java development, Endicott

Jason Katonica
IBM STG z/OS Software Developer - Java, Poughkeepsie

Deborah Mian
IBM STG z/OS Directory and Java Development People Manager, Poughkeepsie

Maura Schoonmaker
IBM STG z/OS Security People Manager, Poughkeepsie

Ross D. Cooper
IBM STG, z/OS Security Software Development, Poughkeepsie

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 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

Part 1

Java and Security

This part introduces the basic concepts of Java Security, along with the Java 2 Security model. Specific considerations for z/OS users of Java to keep in mind when implementing a secure environment for executing their Java stand-alone applications are also highlighted in this part.

© Copyright IBM Corp. 2008. All rights reserved.

1

Overview of Java on z/OS

This chapter introduces the main concepts underlying the operations of the Java software platform, and provides practical information about the specific features provided by z/OS Java.

It also covers the Security framework incorporated in the Java platform, and explains how z/OS Security contributes to enhancing the environment where the Java Virtual machine (JVM) is executing.

© Copyright IBM Corp. 2008. All rights reserved.

1.1 Why to choose Java

There are many reasons why business enterprises and developers choose Java today. These range from strategic business visions to technical advantages over other software platforms. Java, along its succeeding evolutions, has become an industry-class platform and one of the most widely used programming languages.

Developers from around the world know how to program in Java. Many projects, from large-scale enterprise applications to hobby programs, are written entirely in Java. Many universities also offer undergraduate and graduate classes with Java as part of the core curriculum. The viability of future skills induces many corporations to migrate existing code to a Java base.

1.1.1 Introduction to the Java programming language

The programming language Java was introduced by Sun™ Microsystems in 1995 as part of the Java platform. The Java platform contains a number of software products, specifications, and programming interfaces. Together, these components form an environment for cross-platform application development and deployment.

Java in an object oriented language, with syntax similar to the C or C++ language. It does not provide as many low-level functions as C or C++ does, but it does offer a simpler object model.

Java programs are not compiled into machine code, but rather into “byte code”. This byte code is executed by the Java Virtual Machine (JVM). Developers can compile their Java applications once, into byte code, and run the application on any platform that the JVM supports.

Java is a modular programming language. Java source files normally contain one class. A *class* is a construct that contains the set of Java statements and variables that is used to build an object. The object, as a result of the class instantiation, appears to users as one of the functional modules that make up the Java application.

The Java source statements in a class can be modified and maintained without affecting the source statements in other objects. Java objects express their behavior through a set of public methods instead of directly accessible sets of variables or statements. Objects interact by using these public methods to send messages to one another. Therefore, the underlying code in the object can change without affecting the system as long as the public methods remain the same.

Method invocation is not restricted, as could be the case with other languages. Objects can communicate across the boundaries of system processes, threads, JVMs, and even across machines.

Java also provides facilities that enable information in objects to be hidden when needed, so that actual data cannot be freely disclosed.

1.1.2 Java package

The term “Java package” designates a set of Java classes that have been packaged for delivery. The classes in the package are designated as being part of a specific namespace. As an example, com.ibm.encryptionfacility is a Java package header for a specific Java

application. Continuing the example, classes in this package would be com.ibm.encryptionfacility.Encode.java or com.ibm.encryptionfaciliy.Decode.java.

Java packages are delivered and deployed as Java ARchive (JAR) files. JAR files can contain the source code of the Java application, compiled classes, and any number of other resources. JAR files are compressed using the zip file format. They also integrate additional meta-data that relates to the application. For instance, JAR files include, by default, a META-INF directory. This directory contains the *manifest* of the JAR. The manifest contains a set of name-value pairs that describe the package.

If the JAR file contains a number of Java classes, the manifest can contain the Main-Class keyword. This tells the JVM which class's main method should be used if the JAR file were to be executed as if it were a stand-alone program.

JAR files can also be a set of classes that make up a library. The manifest can state which other JAR files that the classes included depend on. For more information about JAR files, refer to the JAR File Overview page located at:

http://java.sun.com/javase/6/docs/technotes/guides/jar/jarGuide.html

1.2 Java Native Interface

The Java platform offers many different APIs to interact with various system resources. However, there are still situations in which services of the hosting platform cannot be invoked using the provided Java API. In other words, this is a case where a given application cannot be written entirely in Java. For such applications, the Java Native Interface (JNI™) framework might provide a solution. The JNI can be used in two ways:

- It can be used to implement native methods in a Java application to invoke non-Java functions.
- It can also be used to modify a native application so that it can be called by Java applications.

JNI was originally designed to be used in combination with C/C++. However Enterprise COBOL for z/OS V3R4 proposes now interlanguage interoperability using JNI.

In most practical cases today, JNI is still used in combination with C/C++. In the z/OS case, C/C++ allows z/OS users to access a large number of z/OS UNIX Assembler Callable Services. In turn, this enables these users to reach resources that may not be accessible directly through Java code. This is a situation where users could decide to use a native language to achieve their goals.

C/C++ can also be used as an intermediary interface, because a module of C/C++ code can provide a link to an Assembler module. That way, JNI can be used to integrate Assembler into a Java application.

Figure 1-1 illustrates an example of a Java program that uses JNI and C to get to a z/OS resource. This infrastructure comprises essentially three functions:

1. A Java program that is the driver which writes to the output device. The source code should contain the interaction between Java and the native method invocation. And preferably, it should also contain logic to deal with error conditions.
2. A Java class with the native method definition.
3. A DLL module, written in C, that contains the entry points for the basic functions. Those functions are then mapped by the z/OS C/C++ Run Time Library (RTL) to Assembler

Callable Services (BPX....), which in turn are executed by the z/OS UNIX Kernel. These callable services allow z/OS UNIX programs to access system resources.

Therefore, the JNI is basically located between the Java Virtual Machine (JVM) and the operating system (z/OS). However, instead of going directly from the JVM to the operating system (as the dotted line in Figure 1-1 indicates), a native C/C++ library is used to invoke access to the required resources.

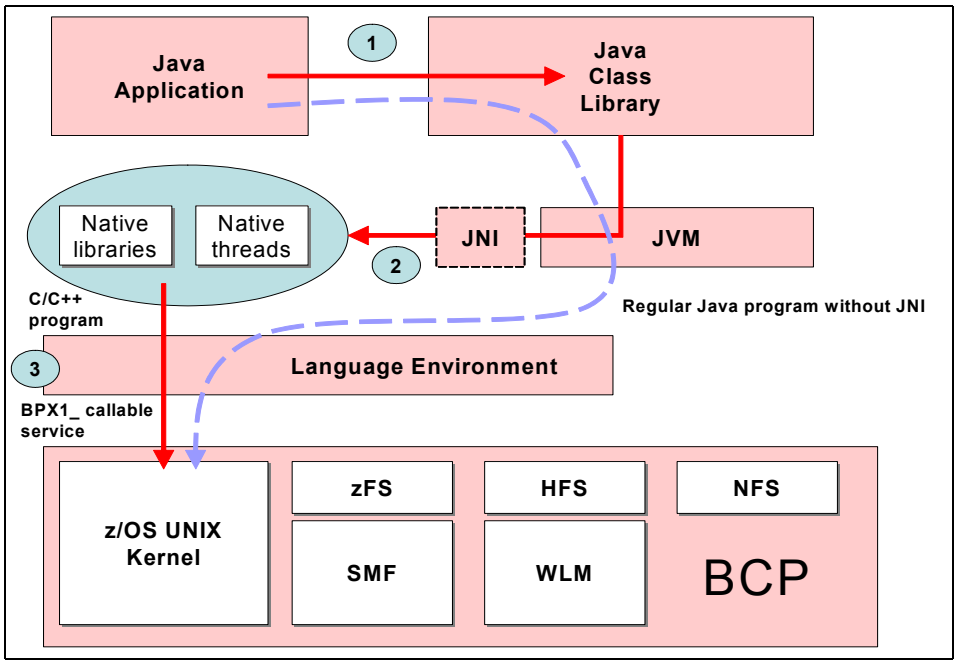


Figure 1-1 Example of using native methods for file I/O

Using JNI offers advantages over using a 100% pure Java program. For example, it allows users to consider the reuse of code from another language that is already available. In addition, the native code generally executes faster than Java code, so some performance gains could also be expected from the use of JNI.

However, there are also disadvantages to using JNI that users need to consider:

- Using JNI is not recommended for a J2EE™ environment such as WebSphere Application Server, because any operational anomalies in the calling program adversely affect the operations of the J2EE server.
- The zAAP (z Application Assist Processor) engine can exclusively be used to run Java code. Therefore, to realize substantial benefit from the zAAP, it is recommended that the use of native code be somehow limited.

Note: Details about Java program developments that use the JNI is a very broad topic, and complete coverage of that topic is beyond the scope of this book.

A basic introduction to Java programming with JNI is available at IBM developerWorks:

`http://www-128.ibm.com/developerworks/edu/j-dw-javajni-i.html`

1.2.1 Basic elements of the Java Native Interface

This section discusses the most important elements to consider in the JNI specifications.

JNI primitive types and reference types

There are two data types used in Java to pass arguments and obtain returned values, which enable the user to interact between the Java language and the native language. The JNI specifications define *primitive types* and *reference types* to represent Java language elements.

The primitive types are very straightforward and can be mapped from C/C++ to Java just by adding a `j` character to the primitive type, as illustrated in Figure 1-2.

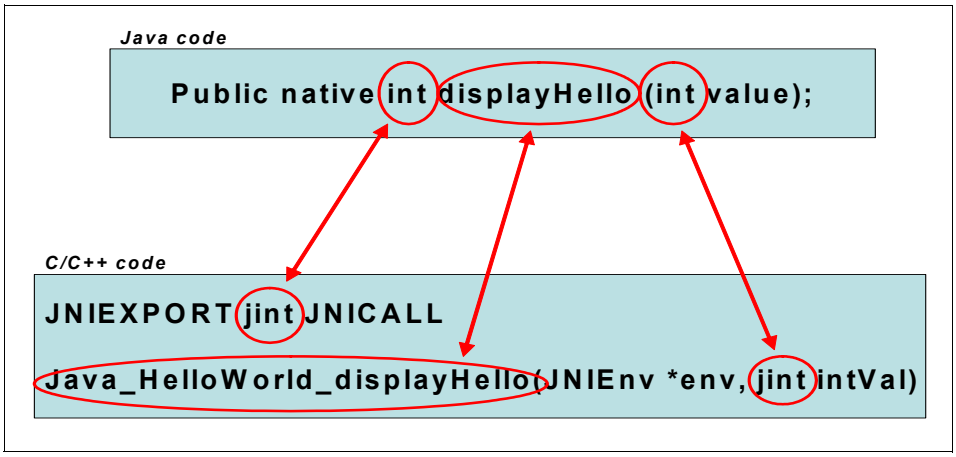


Figure 1-2 Methods mapping

Table 1-1 lists the native data types and describes their size and attributes. Use this table when mapping Java native types to C data types in JNI code.

Table 1-1 Mapping of primitive types

Java type	Primitive type	Description
Boolean	jboolean	unsigned 8 bits
Byte	jbyte	signed 8 bits
Char	jchar	unsigned 16 bits
Short	jshort	signed 16 bits
Int	jint	signed 32 bits
Long	jlong	signed 64 bits
Float	jfloat	32 bits
Double	jdouble	64 bits

JNI reference types correspond to different types of Java objects. Sometimes reference types are also called *object types* (either term is acceptable). They are organized in a hierarchy, as shown in Figure 1-3.

In the C programming language, all non-C objects map to the `jobject` reference type. Accordingly, a Java `String` object will be referenced in C as a `jstring`. A `java.util.Hashtable` object will be referenced as a `jobject`.

Object references play an important role when accessing Java objects from native code with JNI functions, or when passing Java objects as parameters to a native implementation.

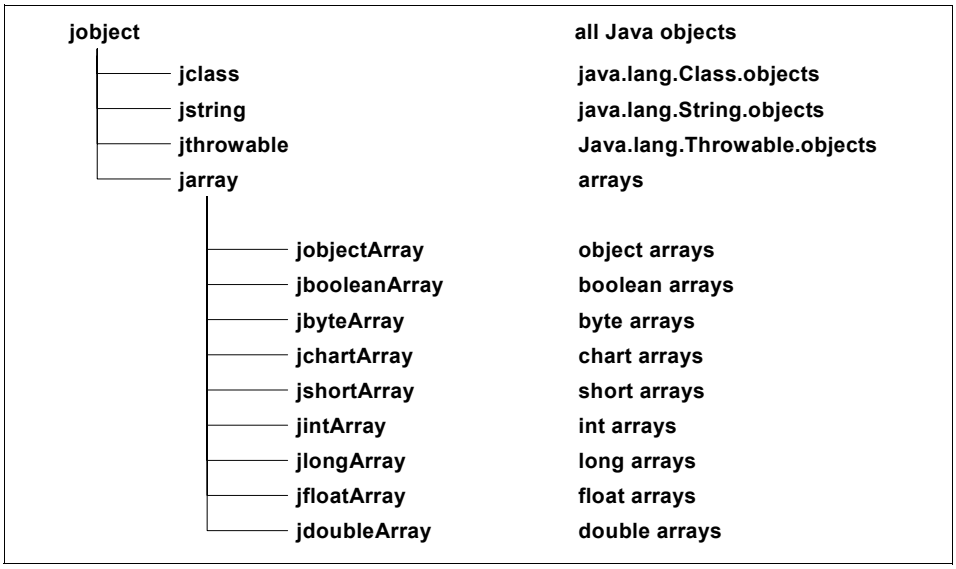


Figure 1-3 JNI reference types

Signature type

Another main data type available in JNI is the signature type. To execute a Java method from native code, the user has to specify the exact signature of the method. For example, consider the following the Java method:

```
long myMethod (float f, String s, Hashtable[] harr);
```

It has the following type signature:

```
(FLjava/lang/String; [Ljava/util/Hashtable;)J
```

This may be considered as a more advanced topic, because calling Java code from within a native program can rapidly become complex to implement. If unsure about the exact signature of a Java method, the user can perform these simple steps:

1. Create a dummy Java class that defines a native method that has the same arguments and return types. For example:

   ```
   class Helpme {
      native long myMethod (float f, String s, java.util.Hashtable[]    harr);
   }
   ```

2. Compile the Java class.
3. Run the **javah** command on the class:

   ```
   javah -jni Helpme
   ```

4. The resulting C header file Helpme.h contains a description of the native method, including its signature:

   ```
   /* DO NOT EDIT THIS FILE - it is machine generated */
   #include <jni.h>
   /* Header for class Helpme */

   #ifndef _Included_Helpme
   #define _Included_Helpme
   #ifdef __cplusplus
   extern "C" {
   #endif
   ```

```
/*
 * Class:     Helpme
 * Method:    myMethod
 * Signature: (FLjava/lang/String;ÝLjava/util/Hashtable;)J
 */
JNIEXPORT jlong JNICALL Java_Helpme_myMethod
  (JNIEnv *, jobject, jfloat, jstring, jobjectArray);
#ifdef __cplusplus
}
#endif
#endif
```

1.2.2 JNI and Security

JNI offers application developers a way to bridge earlier code with Java code. Implementing applications with JNI is not trivial because JNI requires a considerable amount of effort to learn and to implement. When developing code that utilizes the functions that JNI provides, the user should consider the following points:

- Subtle errors in the JNI native code can destabilize the JVM.
- The JNI native code is outside the control of the Java Security mechanisms.
- Any malicious code written in JNI can circumvent Java 2 security, including access control policies.

The use of JNI native code should be strictly controlled. Because it is executed outside the JVM, access restrictions should be enforced using the native system Security model.

Java security as implemented by the JVM operates above the operating system security services, meaning that Java Security is actually implemented in the JVM run-time environment. The privileges to be given at the host operating system level to the Java applications execution streams are well established and fully controlled. In contrast, each new JNI native program brings a new set of functions that need to be controlled, with the exposure of leaving unwanted system privileges to the program.

1.3 Accessing z/OS MVS datasets from Java

One of the most important data stores used by Java implementations are byte-stream files, and the generic file access classes of Java can operate with the z/OS UNIX file as implemented in the z/OS HFS.

In the MVS environment of z/OS, data is stored in the record-oriented datasets. It is important for Java applications executing on z/OS to be able to access such datasets.

There are two different APIs available to access z/OS datasets from Java applications:

- The Java Record I/O API (JRIO)
- The JZOS toolkit API

When using either one of these APIs, the z/OS user ID used to access the dataset is the user ID that invoked the Java application.

1.3.1 Using the Java Record I/O API

The Java Record I/O (JRIO) library lets Java applications access traditional mainframe file systems that the Java I/O library does not support. It also it provides the record-oriented view of the data stored in a file, 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 the product information and the instructions to download and install it:

http://www-03.ibm.com/servers/eserver/zseries/software/java/jrio/overview.html

The JRIO library enables Java applications to access the following types of dataset organizations:

- Partitioned Data Set (PDS) and Partitioned Data Set Extended (PDSE)
- Virtual Sequential Access Method (VSAM) data sets (only the KSDS type of VSAM datasets)
- Other non-VSAM datasets

For more information about JRIO, refer to the Redbooks publication *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177.

1.3.2 Using the JZOS toolkit API

The JZOS toolkit library is a complementary set of functions to the Java Record I/O library (JRIO).

The toolkit provides Java wrappers for the z/OS C/C++ Library functions, so that Java programs can access any z/OS datasets supported by the C/C++ library, using the JZOS Toolkit. This includes the following:

- Partitioned Data Set (PDS)
- Partitioned Data Set Extended (PDSE)
- Sequential data sets
- Virtual Sequential Access File (VSAM) of the type KSDS, RRDS, or ESDS

The JZOS toolkit API and the JRIO library both contain methods for performing I/O in the z/OS environment. They are complementary to each other and provide the user with a choice of both how to access and work with files and datasets, and what kind of access should be performed.

The JRIO library is modeled to be complementary to the standard Java I/O packages, thereby giving a Java developer the ability to work with HFS files and data sets in a Java-like paradigm. Alternatively, the JZOS toolkit API contains JNI wrapped C/C++ methods that allow programmers who are more familiar with C/C++ I/O on z/OS to use that model for their I/O needs.

The list provided in Table 1-2 indicates when to use JZOS and JRIO.

Table 1-2 When to use JZOS

Type of access required	API to use
C/C++ library interface	JZOS
Java data set record stream abstractions	JRIO

Type of access required	API to use
Fine access to system error codes	JZOS
Data set access (Text Stream, Binary Stream and Record Mode)	JZOS
Data set access (Record mode)	JRIO
Portable text file processing (HFS)	JZOS
Portable text file processing (data sets)	JZOS
VSAM data set access (KSDS, ESDS, RRDS)	JZOS, JRIO (KSDS only))
HFS access	JRIO, java.io

For more information about JZOS, refer to the Redbooks publication *Java Stand-alone Applications on z/OS, Volume II*, SG24-7291.

1.3.3 Running a Java program as a batch job

Running Java as a batch job allows the easy integration of Java programs into existing workloads, with proper prioritization by the z/OS Workload Manager (WLM) along with the ability to provide thorough accounting for the Java jobs and to run hundreds of these jobs in parallel.

To run a Java application as a batch job, follow these steps:

1. Write the Java program and compile it.
2. Prepare a JCL script to run the Java program.
3. Submit the job.
4. Examine the job output.

There are two ways to run a Java program as a batch job:

- By using the BPXBATCH or BPXBATSL utility programs
- Or by using the JZOS Batch Launcher

Note that the z/OS user ID which starts a Java batch job is also attached to the JVM execution threads; that is, the JVM is running under the same user ID as the job itself.

1.3.4 Job management using the BPXBATCH and BPXBATSL utility programs

Just as with any other program written for mainframe operating environments, Java batch jobs must be submitted and controlled using JCL statements. IBM provides a utility program called BPXBATCH to run shell scripts and executable files that reside in the z/OS Hierarchical File System, as per directives in a deck of JCL statements.

BPXBATCH has the restriction of handling standard I/O streams only when used within a JCL script. That is, the standard I/O streams must be directed to temporary z/OS UNIX files and then copied to datasets in a subsequent job step.

Another limitation of BPXBATCH is its inability to allow executing programs to access DD cards defined in the enclosing JCL statements.

Therefore, another utility program called BPXBATSL is provided to address the limitations of BPXBATCH. BPXBATSL provides an alternate entry point into BPXBATCH and forces a program to be initiated in the same address space, using a *local spawn* function instead of the traditional UNIX fork/exec. As a consequence of this, all DD cards specified in the JCL deck are made available to the invoked program.

Table 1-3 provides a comparison of these two approaches.

Table 1-3 Comparison of BPXBATCH and BPXBATSL

	BPXBATCH	**BPXBATSL**
Inherit user profile	Yes	No
Return code	Returns the value returned from the running program.	Returns the value returned from the running program multiplied by 256. This is useful in determining errors in BPXBATSL invocation versus program errors in the running program.
DD cards supported	No	Yes
Use z/OS data sets	Yes (not via DD cards)	Yes
JVM runs in the same address space	No	Yes

For more information about BPXBATCH and BPXBATSL, refer to the Redbooks publication *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177.

Job management using the JZOS Batch Launcher

BPXBATCH and BPXBATSL have been the tools of choice for running executable file and shell scripts residing in the Hierarchical File System (HFS) under UNIX System Services from a TSO session and z/OS Job Control Language (JCL).

These tools are useful and flexible in their own right. However, they have some restrictions from the perspective of traditional batch job management in z/OS systems. The JZOS job launcher and runtime APIs address these shortfalls:

- JZOS includes Java classes that make the console communication from 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 system operators via WTOs. It also provides a method to write messages to MVS system logs directly from Java applications. 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, like accepting a **`modify`** command. Consequently, it is possible to develop Java programs that change behavior depending on the commands received from the system operator, without having to restart the job.
- JZOS enables Java programs to be integrated seamlessly with other job steps within a single job. Execution of a job step is typically controlled by return codes from the previous job step. JZOS reliably delivers the exit status of the execution of a Java program to the

following step in a job, which allows for inclusion of Java program steps in a chain of z/OS utilities and programs.

Another considerable benefit of JZOS is its full support for accessing the 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.

- The JZOS batch launcher can direct stdout and stderr input streams to standard z/OS datasets and to the JES SYSOUT dataset. Also, Java programs are able to read from a stdin that is actually a standard z/OS dataset.

  Consequently, system operators are able to monitor the execution of Java programs via System Display and Search Facility (SDSF), just as they would monitor other job steps in z/OS.

Table 1-4 summarizes the differences between BPXBATCH, BPXBATSL, and JZOS. Although JZOS offers many additional features that are desirable to integrate Java programs in batch jobs, it is not intended to replace BPXBATCH and BPXBATSL. For instance, a job that requires you to run shell scripts or programs residing in a z/OS UNIX file will continue to use BPXBATCH and BPXBATSL.

Table 1-4 Comparison between BPXBATCH, BPXBATSL and JZOS

	BPXBATCH	**BPXBATSL**	**JZOS**
Run in the same address space	No	Yes	Yes
DD statements supported	No	Yes	Yes
stdin, stdout and stderr to MVS data set	No	No	Yes
Console communication	No	Yes, via JNI calls	Yes
Return code (System.exit)	Yes, but always 1 otherwise	RC multiplied by 256	Yes
Running programs and shell scripts in USS	Yes	Yes	No

For more information about JZOS, refer to the Redbooks publication *Java Stand-alone Applications on z/OS, Volume II*, SG24-7291.

1.4 Introduction to Java security

Java was created to be exploited in highly distributed environments. The portability of the language meant that security had to be incorporated in the very first steps of Java application development. This way, application users could be assured that the code they obtained was secure and had not been tampered with during transmission. This section explains how to achieve that security.

1.4.1 The Java Virtual Machine Security framework components

Figure 1-4 gives a high level view of the main components of the security framework implemented in Java. On the bottom is the host operating system and the resources it directly controls. This includes the files, processes, network interfaces, and other resources.

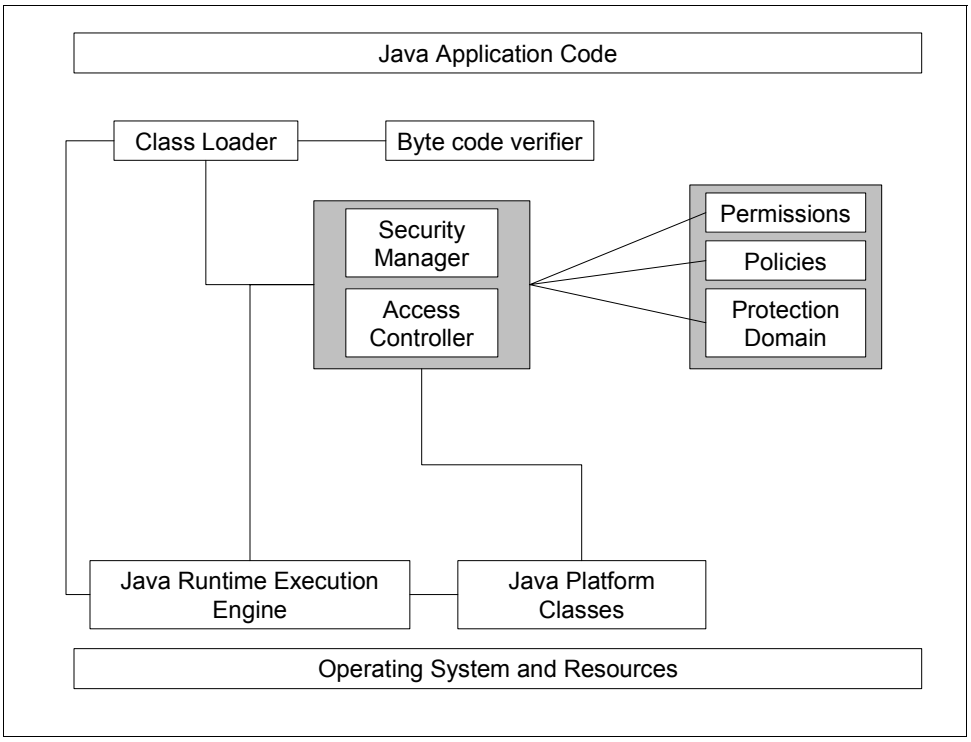


Figure 1-4 Java Security framework components

The JVM that runs above the operating systems has its access to these resources controlled by the operating system access control mechanisms themselves (access control lists, files, or directories, permissions).

The Java Runtime Execution Engine and the Java Platform Classes are two of the core components that make up the Java Runtime Environment (JRE™) that runs above the operating system. The execution engine and platform classes make up the JVM. These components create the necessary abstraction layer to isolate the platform-dependent code from the higher level Java code.

The JVM uses the Class Loader to obtain and load the proper classes to be executed. The class loader executes at a higher lever than the platform class. It does not have any perception of the files and file system infrastructure when loading the byte code into the execution engine. The class loader also allows for libraries to be dynamically loaded at run time.

1.4.2 The byte code verifier

The flexibility of being able to load classes and other code fragments, at any time, from almost any location, led the JVM developers to create the byte code verifier to mitigate the security exposures. The byte code verifier is used by the class loader to ensure that the byte code to be loaded does not have any intrinsic basic security problems. Hostile compilers can create Java byte code that ignores security rules, and also forge pointers and perform many

other malicious acts. These are detected by the byte code verifier, which makes sure that object field and method accesses are always performed according to the rules.

If an object field is set as *private*, then the byte code verifier enforces that all access follows the rules set forth by the language specifically for private members of an object. The byte code verifier also enforces *type safety*; that is, every object is always accessed by what it is. For instance, an object that is instantiated with a type of OutputStream is always accessed as an OutputStream and not as any other type.

The verifier ensures that the code adheres to a number of base constructs:

- There are no stack overflows.
- There are no stack underflows.
- All accesses to both registers and stores are valid.
- All data conversions are legal.
- Byte code instructions parameters are legal.

The verifier is independent of the compiler. It can certify code from the embedded compiler or any other compiler. When using the Just In Time (JIT) compiler, the byte code verifier is used exactly the same way.

Verification and certification at such a low level can assure developers and end users of the validity and safety of the code to be executed. Also note that JAR signing, described in 1.4.4, “JAR file security” on page 16, brings an additional level of verification to the integrity of the byte code.

1.4.3 SecurityManager and AccessController

The JVM also provides security at a higher level than byte code verification via the SecurityManager and AccessController components. Both objects attempt to restrict access as specified by a security policy. SecurityManager can be considered a central control point for all access decisions. AccessController is a particular implementation of an access control algorithm.

Even with SecurityManager in use, many of the methods actually call by default the AccessController implementation introduced with Java2 Security. AccessController decisions are based on policy files, as explained in “Policy file” on page 37.

The role of SecurityManager has diminished in importance with the availability of Java 2 Security. However, it is still used a central “switch” to turn on or off Java 2 Security as indicated in “Running with the SecurityManager” on page 39. SecurityManager also ensures backward compatibility with Java 1 Security.

The SecurityManager.checkXX methods create a PermissionClass and then call the AccessController.checkPermission method. For instance, Example 1-1 shows a simple way to check whether a program has permission to read from a file.

Example 1-1 Checking if a program has permission to read from a file

```
SecurityManager sm = System.getSecurityManager();
if (sm != null ){
   sm.checkRead(file);
}
```

The proper permission would have been created by the default SecurityManager, as shown in Example 1-2.

Example 1-2 Permission created by default SecurityManager

```
AccessController.checkPermssion(new FilePermission(file, “read”));
```

Developers are encouraged to use the more recent AccessController, rather than SecurityManager. Customizing (that is, subclassing) of SecurityManager should only be undertaken with extreme caution. For the purposes of this book, it is assumed that no customization is done and that the default SecurityManager is used.

As shown in Example 1-2 on page 16, the AccessController utilizes classes of type Permission. Permission is an abstract class that represents a system resource.

Java provides a number of Permission classes, such as RuntimePermission, FilePermission, SocketPermission and AllPermission. These classes allow developers to implement the proper access checks prior to manipulating a system resource. They also prevent unauthorized code from running, because a SecurityException is raised in case of denied permission.

Most implementation of the abstract Permission class include a list of “actions” that make up the scope of the controlled access. As an example, FilePermission includes read, write, delete, and execute actions. SocketPermission includes connect, accept, listen, and resolve. Developers are free to create their own Permission classes.

Java 2 Security also introduces the concept of ProtectionDomain. A ProtectionDomain can be thought of as the encapsulation of a set of classes, or a CodeSource, of which instances are granted specific Permissions when the classes are being executed on behalf of a set of Principals. The Permissions granted in the ProtectionDomain override, at class execution time, the security policy that is being enforced by the JVM.

ProtectionDomains allow the current Policy to dynamically map permissions to a CodeSource or a principal identity when it comes to check permission. For example, the default Policy system that is based on the AccessController function reads its policy definitions from a set of static files. These files are read and parsed when the JVM first starts.

If the Policy implementation allows loading the policy from a dynamic data source (such as a database or an LDAP directory), the ProtectionDomain can then couple, at run time, the running CodeSource or principal identity to the permissions specified in the dynamically loaded policy.

Note: Many integrated development environments offer the ability to generate code automatically. Users must exercise caution when exploiting this facility, because when using automatic generation there is always a potential exposure for the final code not to abide by all the specific rules relevant to the user’s specific execution environment.

In other words, Java Security can protect from a large number of attacks, but it cannot protect from mistakes introduced by developers or their tools.

1.4.4 JAR file security

Java archives, or JAR files, can also be secured through a system of signing and signature verification. You can digitally sign a JAR file, thus providing assurance that the code residing inside the JAR file comes from a trusted source. Signing and verification works in conjunction with byte code verification, but provides another level of protection to prevent unauthorized code execution.

The Java platform uses public key cryptography (that is, an asymmetric algorithm with a private and public key pair) to digitally sign JAR files. This is done in a way similar to how the SSL/TLS protocol exploits public key cryptography.

The private key signs the JAR file, and the public key is used to verify the JAR file. The public key and its digital certificate are included in the JAR file. The certificate is digitally signed by a trusted Certificate Authority (CA). The certificate indicates who owns the public key placed in the JAR. The overall process of signing includes two steps:

1. A developer signs the JAR file, using a private key.
2. The public key is placed in the JAR with the accompanying digital certificate.

The Certificate Authority (CA) needs to be in a trusted Java keystore in the receiving system. This keystore should be as protected as the JVM libraries are, from the operating system perspective. The jarsigner tool and keystore tool can make use of the Java Cryptography Extension (JCE). This is discussed in detail in Part 3, “z/OS Java cryptography” on page 119.

After the JAR is signed, a human readable signature file is generated. The signature file is located in the META-INF directory of the signed JAR file, which is the same location as the JAR file’s manifest. The signature file contains a set of hashes, or “checksum” values, of the contents of the JAR file. The hashes are used to verify that the Java classes have not changed since the JAR file was signed.

Example 1-3 displays a sample signature file.

Example 1-3 Sample signature file

```
Signature-Version 1.0
SHA1-Digest-Manifest: h1y5k9TlZrtI+xvgaqMyRsE4V4==
Created-By: 1.6.0
Name: com/ibm/redbooks/sg247610/jaa/RunSamples.class
SHA1-Digest: ba4b72e4c67be054fa91b8eadb477=

Name: com/ibm/redbooks/sg247610/jaa/SampleAuthentication.class
SHA1-Digest: 8psH48cnz3nb57vb1mzOrn76fctde=
```

The SHA1-Digest-Manifest line is the hash of the entire manifest and each file included in the JAR file has its own SHA-1 hash. When the JAR file is being verified by the receiving JVM, before the class loader and byte code verify are started, the signature file is hashed and the results is compared with the SHA1-Digest-Manifest. This prevents malicious attackers from providing fake hashes. Then, each file included in the JAR file is hashed again and the results compared. If any the hashes do not match, a security exception is throw and the class loader does not load the classes.

1.5 Java and z/OS security

As Figure 1-4 illustrates, the Java Virtual Machine (JVM) runs under the control of the host system security. It is granted, by a system administrator, whatever permissions it needs to access system resources.

On z/OS, the JVM runs under the identity of the user that started that application. When running a stand-alone application, the privileges granted to this user might be sufficient. However, depending on the type of application and execution environment, additional privileges might be needed (like access to specific system libraries, files, and services).

The underlying operating system security must be set up to allow proper privileges for the JVM. System file level access control is a crucial element for you to consider.

The JVM libraries should be accessible in read-only mode for all users except for the system administrator who is in charge of installing and maintaining the JVM. This is intended to guarantee that JVM component integrity will not be exposed to compromise by other z/OS components or even by Java applications that would call external z/OS components.

There are two main features of z/OS security that the JVM and application environments must abide by:

- Authorized Program Facility (APF)
- Program Control

The requirements to use APF-authorized libraries or program-controlled libraries must be carefully determined. This can often be difficult to achieve because of a lack of information regarding the deployment of the application on z/OS. In any case, it requires a system security administrator to work jointly with the application developer to establish the proper setup of this part of the execution environment.

1.5.1 Authorized programs

z/OS implements sensitive functions that are to be used by programs with a specific attribute only, called *authorized* programs. By z/OS design, a program is authorized when it executes:

- In supervisor state
- With a Program Status Word (PSW) storage key in the range of 0 to 7
- When designated, or the library it resides in, as authorized by the z/OS Authorized Program Facility (APF)

The PSW storage key that a program starts to run with is determined by the contents of the z/OS Program Property Table (PPT table) in SYS1.PARMLIB(SCHEDxx).

A program starts its execution in supervisor state as per the design of the operating system; or it can switch to supervisor state because it is authorized to a system service that allows this switching into supervisor mode; or it is switched into supervisor state by another authorized program.

Authorized Program Facility

The Authorized Program Facility (APF) is used to allow the system to identify programs that have been granted access to sensitive system functions which need to be protected. In essence, programs that have access to these functions can act as an extension of the operating system.

Using these functions a program can, for example, put itself in supervisor state, modify system control blocks, execute privileged instructions, and turn off logging. APF allows an installation to identify system or user programs that can use sensitive system functions, and grant them this privilege accordingly.

A z/OS program becomes APF-authorized when it meets the following two conditions:

- The program resides in an APF-designated library.

  These libraries, when z/OS datasets, are specified in the PROGxx member of SYSx.PARMLIB. The list can be dynamically modified by the SETPROG and SET PROG operator commands.

For programs residing in the z/OS UNIX file system, the APF-authorized extended attribute bit, also known as the “a” bit, should be set for the file that contains the program.

- The program has been link edited with Authorization Code=1 (AC=1) as a load module that resides in the APF-designated library if it is the first program in a chain of modules. Or it is not link-edited with AC=1, but still resides in an APF-authorized library and has been invoked in a chain of programs that was initiated by an AC=1 program.

Proper protection has to be used to control access to the APF libraries and to the operator commands that modify the list of APF libraries or the “a” extended attribute. For more detailed information about the z/OS RACF External Security Manager and protection of the “a” extended attribute, refer to *z/OS UNIX System Services Planning*, GA22-7800.

Note that If the user tries to run a program from an authorized library that is not link-edited with AC=1, it will not run APF-authorized. However, that same program could be fetched by another that is running APF-authorized and executed in the authorization state in which it is called, or even have its state changed.

Important: The system automatically adds SYS1.LINKLIB and SYS1.SVCLIB to the APF list at IPL. In addition, any module in the link pack area (pageable LPA, modified LPA, fixed LPA, or dynamic LPA) will be treated by the system as though it came from an APF-authorized library.

Users must ensure that they have properly protected SYS1.LPALIB and any other library that contributes modules to the link pack area to avoid system security and integrity exposures, just as they would protect any APF-authorized library.

Authorization to designate a program as APF

The ability of users to designate programs residing in z/OS datasets as APF depends on their access permission to the SYS1.PARMLIB dataset, the Link-Edit Binder program, and to the datasets already designated as APF libraries.

For programs residing in HFS files, users need to be permitted to the BPX.FILEATTR.APF RACF resource in the FACILITY class to be allowed to set the “a” extended attribute for an HFS file. Note that superuser status does *not* grant this capability, and that changing the file contents automatically resets the “a” extended attribute.

To verify whether a program has APF authorization, the **extattr** command should be used in the z/OS UNIX shell. Example 1-4 displays file that does not have authorization.

Example 1-4 File without APF authorization

```
MBUZZET @ WTSC60:/u/mbuzzet>extattr test.txt
test.txt
APF authorized = NO
Program controlled = NO
Shared address space = YES
Shared library = NO
```

Example 1-5 displays a file that does have APF authorization.

Example 1-5 File with APF authorization

```
MBUZZET @ WTSC60:/u/mbuzzet>extattr /usr/lpp/java/J6.0/bin/classic/libjvm.so
/usr/lpp/java/J6.0/bin/classic/libjvm.so
APF authorized = YES
```

```
Program controlled = YES
Shared address space = YES
Shared library = NO
```

1.5.2 Program Control

Program Control was initially designed to control access to programs by giving specific users RACF permissions to resources defined in the PROGRAM class of profiles. These resources are actually load modules that can be loaded and executed in the users' address spaces. In addition to defining resources and activating the PROGRAM class, the Program Control function is turned on in RACF with the command SETROPTS WHEN(PROGRAM).

Along with the pure access control part, the design of Program Control also entails other, related functions such as EXECUTE and Program Access to Data Sets (PADS) protections. The Program Control EXECUTE and PADS protections involve checking whether programs already residing in the address space might be of unknown origin, that is, not coming from program-controlled libraries. This is the "dirty address space" concept in z/OS, and programs responsible for this situation are similar to what are known as "trojans" today in other platforms. If the address space happens to be dirty, then the system refuses to execute sensitive functions that are called by any program that already resides in the address space.

This is this second part of the Program Control implementation that we commonly exploit in the context of z/OS UNIX System Services, although we can still use Program Control for its primary purpose of controlling access to programs for those programs still residing in z/OS datasets (access to programs residing in HFS files is controlled with the "execute" bit in the permission bits associated to the program file).

Actually, the z/OS UNIX exploitation of the Program Control concept is to get a program "tagged" as being controlled so that it can be recognized as such by the system. There are two ways of achieving this:

- When the program resides in a z/OS data set, then a RACF profile in the PROGRAM class is used. Example 1-6 shows the command to mark all programs (shown by the asterisk (*) in the command) in the data set SYS1.SCEERUN2 as being controlled:

Example 1-6 Marking all programs in SYS1.SCEERUN2 as being controlled

```
RALTER PROGRAM * ADDMEM('SYS1.SCEERUN2'//NOPADCHK) UACC(READ)
```

- When the program resides in an HFS file, then the "p" extended attributes must be set for the file. Alike the "a" (APF) extended attributes, only users permitted to the BPX.FILEATTR.PROGCTL resource in the RACF FACILITY class can set the "p" extended attribute for a z/OS UNIX file. Likewise, any change to the file automatically resets the bit. Example 1-4 on page 19 and Example 1-5 on page 19 illustrate the use of the **extattr** command to display the extended attributes of two files. The program control extended attribute is shown along with the APF extended attribute.

The indication of whether a program is controlled or not influences the behavior of z/OS when the BPX.DAEMON profile is defined in the FACILITY class of RACF profiles. When loading programs into the user address space, either from a z/OS data set or HFS file, z/OS checks whether the loaded program is marked as being program-controlled. If the program is not controlled, it therefore renders the address space "dirty" and a message is issued at the system console, as shown in Example 1-7.

Example 1-7 Message issued for program that is not controlled

```
ICH420I  PROGRAM CELHV003 FROM LIBRARY SYS1.CMDLIB CAUSED THE ENVIRONMENT TO
BECOME UNCONTROLLED.
```

The execution of the program modules in the now-dirty address space continues until a program residing in the address space calls one of the following functions:

- setuid()
- seteuid()
- setruid()
- pthread_security_np
- auth_check_resource_np
- _login()
- _spawn()
- _passwd()

Because the system requires that the address space be “clean” to execute any of those functions, the function is not executed in this case and an error code is returned. The message shown in Example 1-7 on page 21 can then be quite useful in pinpointing the reason why the address space became dirty.

1.5.3 APF, Program Control, and the z/OS JVM

The installation process of the JVM qualifies its program components as being program controlled and APF. The relevant extended attribute bits are set during expansion of the files with the **pax** command.

System administrators and Java developers need to understand the differences between APF and Program Control in order to maintain strict control over the security of the system. Any requirement to set program modules as program controlled or APF, on top of what has been automatically set during the JVM installation, must be thoroughly examined. These are requirements usually issued when installing third party applications, or dictated by IBM when in specific execution environments. Examples of IBM requirements to declare additional program modules as APF are:

- The exploitation of the z Application Assist Processor (zAAP), as discussed in 1.6.1, “IBM System z Application Assist Processor” on page 22. It requires a Java library to be APF authorized, so that the Java workload can be properly dispatched onto the specialty engine. The library ships with the JVM and is located at <JAVA_HOME?/lib/s390/libj9ifa24.so.
- WebSphere Application Server’s Control Region (CR) should execute APF authorized by design, but the Servant Region (SR) does not run authorized.
- To utilize Java with CICS®, the Language Environment® (LE) library SCEERUN2 needs to be APF authorized. This is in addition to the library being in defined in both the STEPLIB and DFHRPL. The SDFJAUTH library is also required to be APF authorized.

1.6 Exploiting System z hardware

This section briefly covers the System z hardware features relevant to the execution of Java programs in z/OS; that is, the specialty engine for Java (z Application Assist processor, or zAAP) and the cryptographic hardware devices.

1.6.1 IBM System z Application Assist Processor

The IBM System z Application Assist Processor (zAAP) specialty engine provides an optional and attractively priced execution environment for new Web-based applications and SOA-based technologies, such as Java and XML (for example, z/OS XML System Services).

Note: The zAAP is technically similar to the System z Central processor (CP) general purpose processing unit. However, the System z firmware and z/OS software collaborate to have only Java workload be dispatched on this engine.

When configured with general purpose processors within logical partitions running z/OS or z/OS.e, the zAAPs may help increase general purpose processor productivity and may contribute to lowering the overall cost of computing for z/OS Java technology-based applications. zAAPs are designed to operate asynchronously with the general processors to execute Java programs under the control of the IBM Java Virtual Machine (JVM). This can help reduce the demands and capacity requirements on general purpose processors, which may then be available for reallocation to other workloads in the system.

The IBM JVM processing cycles can be executed on the configured zAAPs with no anticipated modifications to the Java applications. The amount of general purpose processor savings will vary based on the amount of Java application code executed by zAAPs. IBM does not impose software charges on zAAP capacity. Additional IBM software charges apply only when additional general purpose CP capacity is used.

Prerequisites for Java zAAP exploitation are:

- System z10™, System z9®, z990 and z890 systems and future follow-on models only
- z/OS V1R7
- IBM Solution Developers Kit (SDK) for z/OS, Java 2 Technology Edition, V1.4 with PTF (or later) for APAR PQ86689

Java-eligible work may be executed on zAAPs or on central processors subject to one of the following priority execution options specified in the PARMLIB member IEAOPTxx; see Table 1-5.

Table 1-5 zAAP OPT settings

IFAHONORPRIORITY	
NO	Specifies that standard processors will not examine zAAP processor eligible work regardless of the demand for zAAP processors as long as there is standard processor eligible work available.
YES	Specifies that standard processors run both zAAP processor eligible and non-zAAP processor eligible work in priority order when the zAAP processors indicate the need for help from standard processors.

Note: Another zAAP option in the parmlib member IEAOPTxx named IFACROSSOVER is ignored since z/OS 1.8 or since z/OS 1.6 with zIIP support. It should be removed from IEAOPTxx.

For additional information about the zAAP specialty engine, refer to:

http://www-03.ibm.com/systems/z/zaap/

1.6.2 Cryptographic hardware devices

Hardware cryptography on System z allows the offloading of cryptographic computation workload to specialized devices, by thus providing increased performance and an even more secure environment when compared to software cryptography.

System z9 and System z10 host two types of cryptographic devices: CP Assist for Cryptographic Functions (CPACF) and Crypto Express 2 feature (CEX2).

- The CPACF is a standard orderable feature providing a set of basic cryptographic functions in each processing unit (PU) of the system.
- The Crypto Express 2 feature is an optional additional set of cryptographic functions that comes as a priced feature to be plugged into the system's I/O cage.

CP Assist for Cyptographic Functions

CP Assist for Cryptographic Functions (CPACF) comes base with the system. You enable it by installing the LICC Feature Code 3863 in the system. Because there is one CPACF facility per CPU, there is no concept of logical partition sharing for the CPACF. Whatever application is dispatched on a PU has exclusive access to the associated CPACF. The CPACF can be called directly by the application by using the Message Security Assist (MSA) machine instructions or a higher level API (such as ICSF in z/OS). See Part 3, "z/OS Java cryptography" on page 119for a discussion of the use of the hardware cryptographic devices by z/OS Java applications.

The set of cryptographic functions provided by the CPACF includes symmetric encryption (with clear key only) one-way hash algorithms and pseudo random number generation:

- DES
- T-DES
- AES128
- AES192 *
- AES256 *
- MAC
- SHA-1
- SHA-256
- SHA-384 *
- SHA-512 *
- PRNG

* z10 only

Note: There is a CPACF in the specialty engines as well. Workloads eligible for the IFL, zAAP, or zIIP will consume CPACF cycles in the specialty engine if they invoke the MSA functions.

Crypto Express2 (CEX2)

There can be up to 8 CEX2 features plugged into a single system. Each feature contains CEX2 coprocessors that can be shared each between up to 16 logical partitions. The regular feature provides two processors. For System z9 BC and z10 BC™ only, there is a feature available containing just one processor.

Access to the CEX2 services are always mediated through a high level API, for instance through ICSF for the z/OS applications.

Note: CCF, PCICC, and PCICA are cryptographic devices that were available on zSeries systems previous to the z990, z890, z9 and z10 models. The CEX2 feature picked up the functions available in these devices.

Crypto Express2 Coprocessor (CEX2C)

The CEX2C is a so-called *secure* coprocessor. This means the coprocessor deals with secure keys. Those secure keys are always encrypted when they are residing outside the CEX2C physical boundaries so that no one can see the actual value of an application key in system storage or a disk device. The CEX2C coprocessor has been certified at the FIPS 140-2 standard for cryptographic modules with the highest possible level (level 4).

The coprocessor provides the following functions, among others:

- Secure key functions:
  - Symmetric DES, T-DES encryption/decryption
  - Message authentication, hashing
  - PIN processing
  - RSA asymmetric encryption/decryption and digital signature generation/verification
  - Key generation and management, true random number generation
- Clear key RSA functions for SSL/TLS acceleration

The very sophisticated internal architecture, where the cryptographic workload is dispatched and managed into the cryptographic engines by an internal operating system and firmware, also allows as an option to compile into the coprocessor firmware the customer's own cryptographic algorithms.

A CEX2C in System z9 or z10 can be manually switched into accelerator mode (CEX2A). Likewise, it can be switched back to coprocessor mode. Any combination of CEX2Cs and CEX2As is possible in a CEX2 feature.

Crypto Express2 Accelerator (CEX2A)

In accelerator mode, a CEX2A runs a much reduced set of functions operating with clear keys only. Actually these are the functions needed to support the SSL/TLS handshake which are performed, when in this mode, at an extremely high speed.

2

Java 2 authentication and authorization services

This chapter introduces the authorization and authentication services provided by Java 2 security.

The following functions are addressed in this chapter:

- Java Authentication and Authorization Service (JAAS)
  - LoginModules
  - CallbackHandlers
  - LoginContext
- Java policy files

© Copyright IBM Corp. 2008. All rights reserved.

2.1 Introduction to Java Authentication and Authorization Service

Java Authentication and Authorization Service (JAAS) is part of the Java Security framework. JAAS augments Java Virtual Machine code-based security by adding a user-centric framework for implementing application security. JAAS version 1.0 was originally shipped separately from the Java Runtime Environment (JRE). It began shipping as part of the JRE as of Version 1.4.

Java applications can run on a wide variety of platforms. Java was created to be widely distributed. Security had to be integrated into the framework from the very beginning to allow application developers the ability to sent their code to a number of possibly insecure platforms over insecure protocols.

For example, a Java application, called an *applet*, can be executed in a Web browser. Applets can be included on any Web page loaded into the browser and executed. In the Java 2 platform, the security framework was able to enforce Java resource access control on the basis of the Web location from which the application code was delivered. In addition, Java packages can be signed by the application programmer. Digital signing of a Java package guarantees users that the application code they are running came from the intended issuer and was not altered, either maliciously or by accident, during download.

However, these security mechanisms are code-centric and do not provide finely-grained access control on the basis of who is invoking the code. Java Authentication and Authorization Service enhanced the framework by adding support for user-centric security, providing user authentication support and extending the authorization mechanism to enforce access control based on the authenticated identity.

Providing information about creating a new JAAS LoginModule is beyond the scope of this book. For more information about that topic, refer to *JAAS LoginModule Developers Guide*:

`http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASLMDevGuide.html`

When working with the Java platform's security framework, users should be aware of the concepts, classes, and interfaces that are used throughout the platform. Figure 2-1 illustrates the framework components and their interactions.

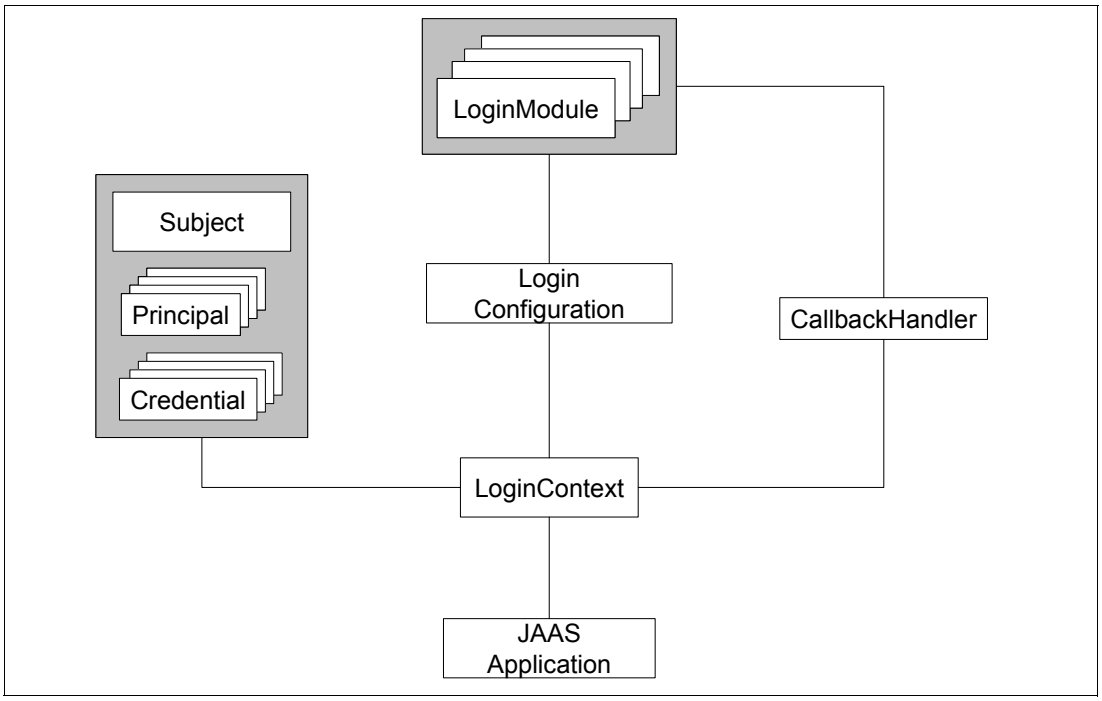


Figure 2-1 JAAS component overview

The javax.security.auth.Subject class represents an identity or a collection of information that comprises at least one identity (note that a single Subject may relate to multiple identities). The binding of these two entities is done through the javax.security.auth.Subject class, using the Principal objects as parameters. A *Principal* represents an entity which could be a person, company, or machine. In addition to Principals, Subjects can store security-related objects that are referred to as *credentials*.

There are two sets of credentials, those that are private to the Subject and those that can be made public. For instance, a Subject might own a secret cryptographic key in its private set of credentials and a public key or Kerberos token in its public credentials.

All of the information stored in a Subject can be used to make access decisions based on who or what the requesting identity is. Subjects, Principals, and credentials are all abstract notions, which makes the security framework very flexible. A Subject can contain information about a user (such as an LDAP distinguished name) as well as the credentials that have been obtained during the log on procedure. The security framework allows you to make a decision based on all of these, on a combination of these, or on a single piece of this information.

Auditing

Note, however, that auditing is not currently addressed in the core feature set of the Java authentication and authorization framework. It is the responsibility of application developers to develop their own auditing scheme.

2.1.1 Differences between IBM JAAS on z/OS and Sun JAAS

For the purposes of this book, we concentrate on the IBM version of JAAS. Sun provides a set of classes with its version of the Java platform. These classes do not directly apply to the z/OS platform.

IBM has made the following changes:

- All classes in the package com.sun.* have been reimplemented and moved to the respective com.ibm.* packages.
- IBM ships a specific JAAS LoginModule, OS390LoginModule, which supports basic login using the z/OS System Authorization Facility (SAF) that is using RACF or an equivalent product.
- IBM ships the SAFPermission classes for checking access to SAF resources. For more information about this topic, refer to the SAFPermission class in Chapter 4, “System Authorization Facility interfaces in z/OS Java” on page 55.
- The IBM specific ThreadSubject.doAs has been included as the default implementation of ThreadSubject.

For more information about the IBM version of JAAS, refer to the following site:

http://www-03.ibm.com/servers/eserver/zseries/software/java/jaas.html

Note: To use the JAAS code that ships with the IBM Java platform on z/OS, you must enable it by adding auth.policy.provider=com.ibm.security.auth.PolicyFile to the java.security file. This file is normally located in $JAVA_HOME/lib/security/java.security.

2.2 Authentication

JAAS authentication is a plugin-based framework. This allows a clear separation between authentication technologies and applications. Thus, application programmers do not have to be concerned with what type of authentication mechanism is being used. Instead, they can develop applications that make sure the Java security framework is properly called.

Note that the pluggable JAAS LoginModules can be chained together. This is similar to the Pluggable Authentication Module (PAM) approach developed for other platforms. It provides applications with a set of possible authentication checks.

To use JAAS, application developers first instantiate a javax.security.auth.login.LoginContext object.The LoginContext works with the LoginModules and a configuration file to authenticate a user.

The configuration file specifies which LoginModules to use for this application, as well as the order in which these LoginModules should be called. The LoginContext can also use javax.security.auth.callback.CallbackHandler to communicate to the entity that is requesting authentication. The CallbackHandler can ask the entity for the information required to complete the process. There are many possible uses of the CallbackHandler, such as asking the user for a user name and password, a personal identification number (PIN), or other kinds of authentication data. In addition, the application can provide the status of the authentication process through the CallbackHandler.

CallbackHandlers can be chained together in a way similar to LoginModules. If the authentication process requires a user name, password, and authentication token (such as a Kerberos ticket, for example), the LoginModules can support the proper CallbackHandler for each type of information.

2.2.1 JAAS LoginModule Configuration

The z/OS System Authorization Facility (SAF) user registry (that is, RACF if the IBM user registry is used) can be invoked by any Java application that uses JAAS.

A OS390LoginModule is provided with the IBM JVM on z/OS, and the relevant configuration file is shown in Example 2-1.

Example 2-1 Configuration file for OS390LoginModule.

```
TestApplication {
com.ibm.security.auth.module.OS390LoginModule required debug=”true”;
};
```

TestApplication is the name of the Java application that will make use of this LoginModule configuration. The Java class used for authentication is com.ibm.security.auth.module.OS390LoginModule.

The required attribute (flag) tells the JAAS runtime that for the login process to succeed, this LoginModule must authenticate successfully.

The JAAS framework implements four different types of flags for LoginModules, as described in Table 2-1.

Table 2-1 LoginModule flag

Flag	Description
required	The login module is required to succeed. - If login fails or succeeds, authentication still continues to proceed down the LoginModule list.
requisite	The login module is required to succeed. - If login succeeds, authentication still continues to proceed down the LoginModule list. - If it fails, control is immediately returned to the application. Authentication does not continue down the LoginModule list.
sufficient	The login module is not required to succeed. - If it does succeed, control is immediately returned to the application and authentication does not proceed down the LoginModule list. - If it fails, authentication proceeds down the LoginModule list.
optional	The login module is not required to succeed. - Whether or not this module succeeds, authentication proceeds down the LoginModule list.

Order matters when traversing the LoginModule list. The overall authentication succeeds if all of the “required” and “requisite” LoginModules succeed. If the configuration defines a “sufficient” LoginModule, and it succeeds, then only the “required” and “requisite” LoginModules encountered before the sufficient module need to succeed because authentication ends as soon as the “sufficient” module succeeds). If the configuration lacks a “required” or “requisite” LoginModule, then there must be at least one “optional” or “sufficient” module defined.

LoginConfigurations can also define a set of options to be passed into the LoginModule. The LoginModule defines what options are available. These options are then placed in the configuration file as a key-value pair. The key and value should be separated by an equal (=)

sign, and the value should be surrounded in double (“) quotation marks. The configuration can also have JVM level variables expanded. For instance, if the configuration contains somekey=“${java.home}“, the configuration will expand that to whatever the java.home JVM variable is set to when starting Java. Example 2-1 illustrates the option debug=true being passed into the com.ibm.security.auth.module.OS390LoginModule.

2.2.2 JAAS sample application

Example 2-2 shows the sample application that will be using the configuration shown in Example 2-1.

Example 2-2 JAAS sample application.

```
package com.ibm.redbooks.sg24160.jaas;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.security.Principal;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.TextOutputCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import com.ibm.security.util.Password;

public class SampleAuthentication {
    public static void main(String[] args) {
        authenticate();
    }

    public static Subject authenticate() {
        Subject subject = null;
        try {
            LoginContext lc = new LoginContext("RunSamples", new
MyCallbackHandler());
            lc.login();

            subject = lc.getSubject();

            Set principals = subject.getPrincipals(Principal.class);
            System.out.println("Login a success. Subject contains : " +
principals.toString());

        } catch (LoginException ex) {
            System.err.println("LoginException: " + ex.getMessage());
            ex.printStackTrace();
        }
```

```
        return subject;

    }
}

/**
 * The application must implement the CallbackHandler.
 *
 * <p>
 * This application is text-based. Therefore it displays information to the user
 * using the OutputStreams System.out and System.err, and gathers input from the
 * user using the InputStream, System.in.
 */
class MyCallbackHandler implements CallbackHandler {

    /**
     * Invoke an array of Callbacks.
     *
     * <p>
     *
     * @param callbacks
     *            an array of <code>Callback</code> objects which contain the
     *            information requested by an underlying security service to be
     *            retrieved or displayed.
     *
     * @exception java.io.IOException
     *                if an input or output error occurs.
     *                <p>
     *
     * @exception UnsupportedCallbackException
     *                if the implementation of this method does not support one
     *                or more of the Callbacks specified in the
     *                <code>callbacks</code> parameter.
     */
    public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {

        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof TextOutputCallback) {

                // display the message according to the specified type
                TextOutputCallback toc = (TextOutputCallback) callbacks[i];
                switch (toc.getMessageType()) {
                case TextOutputCallback.INFORMATION:
                    System.out.println(toc.getMessage());
                    break;
                case TextOutputCallback.ERROR:
                    System.out.println("ERROR: " + toc.getMessage());
                    break;
                case TextOutputCallback.WARNING:
                    System.out.println("WARNING: " + toc.getMessage());
                    break;
                default:
```

```
                    throw new IOException("Unsupported message type: " +
toc.getMessageType());
                }

            } else if (callbacks[i] instanceof NameCallback) {

                // prompt the user for a username
                NameCallback nc = (NameCallback) callbacks[i];

                // ignore the provided defaultName
                System.err.print(nc.getPrompt());
                System.err.flush();
                nc.setName((new BufferedReader(new
InputStreamReader(System.in))).readLine());

            } else if (callbacks[i] instanceof PasswordCallback) {

                // prompt the user for sensitive information
                PasswordCallback pc = (PasswordCallback) callbacks[i];
                System.err.print(pc.getPrompt());
                System.err.flush();
                try {
                pc.setPassword(Password.readPassword(System.in));
                }
                catch (Exception e){
                    System.err.println("Error obtaining password ");
                    e.printStackTrace();
                }

            } else {
               throw new UnsupportedCallbackException(callbacks[i], "Unrecognized
Callback");
            }
        }
    }

}
```

This module defines two classes, SampleAuthentication and MyCallbackHandler. SampleAuthentication has an *authenticate* method and a *main* method to enable running this class from the command line. MyCallbackHandler has one method: *handle*. This class is used to communicate with the user, using a test-based prompt for the user name and password.

The *authenticate* method defines a new LoginContext passing in the name “RunSamples”, and a new instance of the MyCallbackHandler object. The newly created LoginContext’s login method is run. At this point, the Java 2 security system runs through the list of LoginModules that were defined in the configuration. Notice how the code does not have any mention of the actual class to use to do the login. To tie this sample application to the sample configuration. a few options need to be specified when starting Java.

```
java -Djava.security.auth.login.config=jaas.config SampleAuthentication
```

The parameter -Djava.security.auth.login.config tells JAAS where it can find the configuration file that defines which LoginModules to use. After compiling `SampleAuthentication`, it can be run as shown in Example 2-3.

Example 2-3 Execution of SampleAuthentication.

```
WTSC60:/u/mbuzzet/sg247610>/usr/lpp/java/J6.0/bin/java
-Djava.security.auth.login.config=sample_jaas.config -jar
sg24_7610_samples_jaas.jar
---------------------------------
OS/390 user id:mbuzzet
OS/390 password:
                [OS390LoginModule] authentication with RACF succeeded
                        userID = mbuzzet
                [OS390LoginModule] added OS390Principal and
OS390PasswordCredential to Subject
Login a success.
Subject contains : [OS390UserPrincipal: userName: mbuzzet]
---------------------------------
```

Example 2-3 shows a successful run of the `SampleAuthentication`. User `mbuzzet` entered a valid password and successfully authenticated with RACF. The LoginModule then created a OS390UserPrincipal, which represents a RACF user, and attached it to the Subject. Also, the LoginModule attached a OS390PasswordCredential to the Subject. The Subject is now fully formed and can be used throughout the Java code. The SampleAuthorization code merely displays that the login was successful and shows what the current Subject contains after authentication.

Example 2-4 shows an unsuccessful attempt to authenticate. Notice that there are a number of error values displayed. This is because the debug option was passed into the LoginModule from the configuration file detailed in Example 2-1. The stack trace also shows where and why the login attempt failed. In this case, the authentication attempt was cancelled due to an incorrect user password. Because stack traces should be read backwards, we can see that line 21 of the SampleAuthentication class created a LoginContext. From there, the system called down the OS390LoginModule's login method. This is where the error occurred.

Note: Many authentication systems do not differentiate between an incorrect password and an incorrect username. If they did, user names could be determined and a brute force attack could be used to gain unauthorized access.

The OS390Login module returns an invalid password whenever the supplied credentials do not match the credential store.

Example 2-4 Unsuccessful attempt to login

```
WTSC60:/u/mbuzzet/sg247610>/usr/lpp/java/J6.0/bin/java
-Djava.security.auth.login.config=sample_jaas.config -jar
sg24_7610_samples_jaas.jar
OS/390 user id:nobody
OS/390 password:

                [OS390LoginModule] authentication with RACF failed.
                Contents of PlatformReturned object:
      success = false
      errno = 143
      errno2 = 151782877
      errnoMsg = EDC5143I No such process.
                [OS390LoginModule]: aborted authentication attempt
LoginException: Incorrect user password
javax.security.auth.login.FailedLoginException: Incorrect user password
```

```
        at com.ibm.security.auth.module.OS390LoginModule.login(OS390LoginModule.
java:238)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:59)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:39)
        at java.lang.reflect.Method.invoke(Method.java:612)
        at javax.security.auth.login.LoginContext.invoke(LoginContext.java:795)
        at javax.security.auth.login.LoginContext.access$000(LoginContext.java:2
09)
        at javax.security.auth.login.LoginContext$4.run(LoginContext.java:709)
        at java.security.AccessController.doPrivileged(AccessController.java:251
)
        at javax.security.auth.login.LoginContext.invokePriv(LoginContext.java:7
06)
        at javax.security.auth.login.LoginContext.login(LoginContext.java:603)
        at com.ibm.redbooks.sg24160.jaas.SampleAuthentication.authenticate(Sampl
eAuthentication.java:30)
        at com.ibm.redbooks.sg24160.jaas.SampleAuthentication.main(SampleAuthent
ication.java:23)
```

2.3 Authorization

JAAS authorization combines the Java platform version 1.3 code source-based access controls with identity-based access control. After an identity is authenticated by JAAS as described in 2.2, “Authentication” on page 28, the Security Manager can make access control decisions. This constitutes the core of Java 2 Security.

Note: Java 2 Security can be turned on or off by setting a JVM property. The -Djava.security.manager option, when present, enables Java 2 Security.

Access decisions are determined by an AccessControlContext. The AccessControlContext is a Java class that represents an encapsulation of system resources. The AccessControlContext is a snapshot retrieved from the AccessController (see 1.4.3, “SecurityManager and AccessController” on page 15 for an explanation of the AccessController).

The AccessController class makes access decisions based on the current execution thread or environment that it encapsulates. AccessControlContext allows you to make access control decisions from within a different context. This may seem confusing at first, so imagine a scenario where you have a set of worker threads in the Java Virtual Machine. Each thread might need to verify that the caller has authorization to do some work. To do this, each thread can ask the AccessController for an AccessControlContext. This is done by calling the static method getContext defined in the AccessController class. This reference to a context can then be used to perform any type of permission checking.

Prior to JAAS, AccessControlContext attached the location of the code source and the verification of the digital signature of JAR files. The original implementation therefore defined security based on where the code was coming from. JAAS adds the Subject, complete with Credentials and a set of Principals to the AccessControlContext. As a result,

SecurityManager and AccessController can make decisions based not only on *where* the code came from, but also on *who* is executing the code.

It is important to note that the Java platform access control list exploitation is based on the idea of “least privileged.” A Java application’s thread of execution can include any number of different modules, packages and classes, all with different security characteristics. The intersection of these characteristics is given to the thread of execution. For example, if a program that was in an unsigned package calls a method from a package that was signed, the authority level is reduced to the least common denominator. The executing code will have the authority of an unsigned package.

When the Java Virtual Machine starts with a SecurityManager running, it has the authority of the identity that started the initial process. To allow JAAS authenticated users to have access to resources that are different from the initial identity, the Java platform provides the ability to change the Subject that is associated with the AccessControlContext. This is known as the “doAs” family of methods. These methods allow the calling code to associate a different Subject to an AccessControlContext when performing a privileged action. The Subject object has the following static methods:

- doAs(Subject, PrivilegedAction)
- doAs(Subject, PrivilegedActionException)
- doAsPrivileged(Subject, PrivilegedAction, AccessControlContext)
- doAsPrivileged(Subject, PrivilegedActionException, AccessControlContext)

The doAs methods use the current AccessControlContext and attach the Subject that is passed in. The doAsPrivileged allows the running code to provide a specific AccessControlContext. A PrivilegedAction is a Java interface that defines a single method, the run method. If the doAs of doAsPrivileged get passed a PrivilegedAction that may throw a check exception, then the PrivilegedActionException form should be used. This allows the exceptions to be propagated properly to the calling code.

Example 2-5 shows a very simple implementation of a PrivilegedAction. This class implements the run method. Basically, when this code is executed, it will check SAF to see if the AccessControlContext it is running under has UPDATE authority to the resource BPX.SERVER in the RACF FACILITY class.

If the authority is granted, it will display the message `PASSED SAFPermission FACILITY BPX.SERVER, Update`. Otherwise, it will display `FAILED SAFPermission FACILITY, BPX.SERVER, Update` and the exception that caused the failure.

Example 2-5 Sample SAF Permission

```
package com.ibm.redbooks.sg247610.jaas;

import java.security.AccessControlContext;
import java.security.AccessController;
import java.security.PrivilegedAction;

import com.ibm.security.auth.SAFPermission;

public class SampleSAF implements PrivilegedAction<Object> {
    public Object run() {

        // Get the current AccessControlContext
        AccessControlContext accessControlContext = AccessController.getContext();
```

```
        // Create the SAFPermission and do checkPermission.
        try {
            accessControlContext.checkPermission(new SAFPermission("FACILITY",
"BPX.SERVER", SAFPermission.__UPDATE_RESOURCE));
            System.out.println("PASSED SAFPermission FACILITY BPX.SERVER,  Update
");
        } catch (Exception e) {
            System.out.println("FAILED SAFPermission FACILITY, BPX.SERVER, Update
" + e);
        }

        return (Object) null;
    }
}
```

Example 2-6 combines the features of Example 2-4 and Example 2-5 to form a complete JAAS application. When started, the application will prompt for a user name and password. After a successful login, the application will check to see whether the new identity created as a result of the authentication has the UPDATE authority to BPX.SERVER.

Example 2-6 SampleAuthorization

```
package com.ibm.redbooks.sg247610.jaas;

import java.security.PrivilegedAction;
import javax.security.auth.Subject;

public class SampleAuthorization {
    Subject sampleSubject = null;


    public SampleAuthorization (){
        this.sampleSubject = SampleAuthentication.authenticate();
    }

    public void runSAF() {

        try {
            PrivilegedAction<Object> action = new SampleSAF();
            Subject.doAsPrivileged(this.sampleSubject, action, null);
        }

        catch (Exception e) {
            e.printStackTrace();
        }

    }

    public void runSimpleFile() {
        try {
            PrivilegedAction<Object> doIt = new SampleFile();
            Subject.doAsPrivileged(this.sampleSubject, doIt, null);

        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }

    public void runSimpleProperty() {
        try {
            PrivilegedAction<Object> doIt = new SampleProperty();
            Subject.doAsPrivileged(this.sampleSubject, doIt, null);

        } catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

To run the code in Example 2-6, the JVM needs to start with a few extra arguments. Because the application is using JAAS, we need a JAAS configuration file. We can reuse the one in Example 2-1. In addition to the JAAS configuration file, we need a policy file.

Policy file

A policy file is a flat text (ASCII) file. Policy files are used by the default SecurityManager to grant access to Java Permissions. When running with a security manager, if access to a resource is required but it is not defined by a policy file, then access is denied.

Access to policy files must be restricted in order to guarantee their integrity. Similar to files and libraries that make up the JVM, policy files should be kept read-only to almost all users, even the initial identity that started the JVM. We recommend creating a specific group of users that have write access to these files. This will prevent other users from changing the authorizations defined by the system programmer, the application developer, or the security administrator who created the policy files. It will also prevent unauthorized Java code from corrupting or changing what is defined in the policy file.

Example 2-7 contains the statements we need in the policy file for the sample application.

Example 2-7 Java policy file

```
grant {
/* permissions for JAAS */
   permission javax.security.auth.AuthPermission "createLoginContext.RunSamples";

/* permissions for SampleCallbackHandler */
   permission java.lang.RuntimePermission "writeFileDescriptor";
   permission java.lang.RuntimePermission "readFileDescriptor";

/* permissions for SampleAuthentication */
   permission javax.security.auth.AuthPermission "doAs";
   permission javax.security.auth.AuthPermission "getSubject";
   permission javax.security.auth.AuthPermission "doAsPrivileged";
};

/* permsissions for SampleSAF */
grant codeBase"file:/-" {
   permission com.ibm.security.auth.SAFPermission "*", "*";
```

```
};

/* permissions for SampleFile */
grant Principal com.ibm.security.auth.OS390UserPrincipal "mbuzzet" {
   permission java.io.FilePermission "foo.txt", "read";
   permission java.util.PropertyPermission "java.class.path", "read";
};
```

The syntax of a Java policy file is similar to the syntax of a JAAS configuration file. Permissions are granted by the use of a grant statement. Inside the brackets are a set of permission statements ended by a semicolon (;). The general structure of a policy file is depicted in Example 2-8.

Example 2-8 Policy file syntax

```
grant [SignedBy “signer_name”] [,CodeBase “URL”]
   [, Principal [principal_class_name] “principal_name”]
   [, Principal [principal_class_name] “principal_name”]... {
permission permission_class_name [ “target_name”]
   [, “action”] [SignedBy “signer_names”];
}
```

Example 2-7 shows that we are granting access to createLoginContext. Because the first grant statement is not followed by a modifier (such as a code path or a Principal name), it applies to every identity. This particular permission is used to allow the JAAS LoginModule to create a LoginContext. The “RunSamples” name tells the SecurityManager that this applies to context name RunSamples. In Example 2-2, we created a named LoginContext with the same name. If we wanted to authorize every application to be able to create the LoginContext, we could replace the RunSamples name with an asterisk (*).

The read and write FileDescriptor permission are used by the callback handler included in the authentication section. The doAs and doAsPrivileged permissions grant the caller the ability to call the doAs family of methods.

The second grant statement has a codeBase modifier. This is the older-style authorization, based on file locations. This particular statement gives access to all files. Access is given to all of the SAFPermission options.

The final grant statement shows how to authorize a specific principal to a set of resources. As shown, we are assigning the ability to read the file `foo.txt` and read the JVM system property `java.class.path` to user `mbuzzet` of the class `com.ibm.security.auth.OS390UserPrincipal`.

Instead of granting access based on a Principal or a codeBase, policy files also support a signedBy modifier. This modifier indicates the alias for a certificate that is stored in a key store. The digital signature of the code is verified against the public key of the alias after it is retrieved from the keystore. The values of signedBy can be a single alias or a comma-separated list of aliases.

Policy files also support property expansion. When a string like ${random.property} appears in a policy file, it will be expanded to the current value of the property. For instance, granting the read operation to a file in a user’s home directory would look something like the following:

```
permission java.io.FilePermission “${user.home}”, “read”;
```

If the user's name is Bill, and Bill's home directory is /home/bill, then the system will see the permission as:

```
permission java.io.FilePermission "/home/bill". "read";
```

Property expansion can reduce the overall complexity of a given policy file. It also can make applications more platform-independent. The property expansion mechanism automatically inserts the proper file system separators. In the preceding example, if Bill's home directory was located at C:\users\Bill, then the expanded line would have the proper file system separators.

Note: Property expansion does not work with nested properties. A property like ${random.${user.home}} will not be expanded and the entire permission line will be ignored. In addition, if the property is not defined, the line is ignored.

Running with the SecurityManager

Example 2-9 shows a sample run of the application with authorization turned on. There are a number of options to be passed, as listed in Table 2-2.

Table 2-2 Options to be passed for the SecurityManager

Option	Description
-Djava.security.auth.login.config=./loginmodules390.config	This is the JAAS LoginModule configuration file.
-Djava.security.manager	This option tells Java to use a SecurityManager, thus turning on authorization. This option is also used to tell the JVM which SecurityManager to use. If left blank, the default is assumed.
-Djava.security.policy=./java2_1.policy	This is the Java 2 policy file to use. It includes all of the grant statements.

Example 2-9 Successful use of SAFPermission

```
MBUZZET @ WTSC60:/u/mbuzzet/sg247610>/usr/lpp/java/J6.0/bin/java -Djava.securit
y.auth.login.config=sample_jaas.config -Djava.security.manager
-Djava.security.policy=sample_java2.policy -jar sg24_7610_samples_jaas.jar
-----------------------------------
OS/390 user id:mbuzzet
OS/390 password:

               [OS390LoginModule] authentication with RACF succeeded
                       userID = mbuzzet
               [OS390LoginModule] added OS390Principal and
OS390PasswordCredential to Subject
Login a success.
Subject contains : [OS390UserPrincipal: userName: mbuzzet]
-----------------------------------
Checking SAF authorization
PASSED SAFPermission FACILITY BPX.SERVER,  Update
-----------------------------------
Checking File authorization
PASSED FilePermission foo.txt read
 foo.txt does not exist
-----------------------------------
```

```
Checking Property authorization
PASSED PropertyPermission java.class.path read
 java.class.path = sg24_7610_samples_jaas.jar
```

The policy file gives authorization to check the SAFPermission. The user `mbuzzet` must also have the proper SAF permission.

Example 2-10 shows a session where the same policy file is in use, but a different set of credential are given to JAAS (and thus SAF). The user `ramaa` does not have the proper authority-defined SAF, but does have the proper authority defined in the Java policy file.

Example 2-10 Unsuccessful use of SAFPermission

```
MBUZZET @ WTSC60:/u/mbuzzet/sg247610>/usr/lpp/java/J6.0/bin/java
-Djava.security.auth.login.config=sample_jaas.config -Djava.security.manager
-Djava.security.policy=sample_java2.policy -jar sg24_7610_samples_jaas.jar
-----------------------------------
OS/390 user id:ramaa
OS/390 password:

               [OS390LoginModule] authentication with RACF succeeded
                       userID = ramaa
               [OS390LoginModule] added OS390Principal and
OS390PasswordCredential to Subject
Login a success.
Subject contains : [OS390UserPrincipal: userName: ramaa]
-----------------------------------
Checking SAF authorization
FAILED SAFPermission FACILITY, BPX.SERVER, Update


java.security.AccessControlException: Access denied (FACILITY BPX.SERVER 2)
-----------------------------------
Checking File authorization
FAILED FilePermission foo.txt read
-----------------------------------
Checking Property authorization
FAILED PropertyPermission java.class.path read
```

Example 2-10 shows how different security models and their implementations can operate together. It also shows that in order to protect an application, the entire stack must be taken into consideration. z/OS provides the Java platform with many proven security technologies. Java can utilize these technologies and give developers a highly secure platform to deploy mission-critical applications.

Policy Tool

Policy files can be very complex, but they provide great flexibility. For more information about this topic, you can refer to the “Default Policy Implementation and Policy File Syntax” document located at:

http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html

However, writing policy files by hand is cumbersome and can be error-prone. The Policy Tool, which ships with the Java Developer Kit (JDK™), provides a graphical interface on a

workstation. The tool is written in Java and shields developers from having to know the strict syntax of policy files.

Figure 2-2 shows the Policy Tool main window. It displays the name of the policy file, and all of the policy entries it includes.

Figure 2-2 The Policy Tool main window

Adding a policy entry is a simple operation, as shown in Figure 2-3. The Policy Tools entry dialog box allows developers to select the codeBase, signedBy, or a set of principals to associate permissions to.

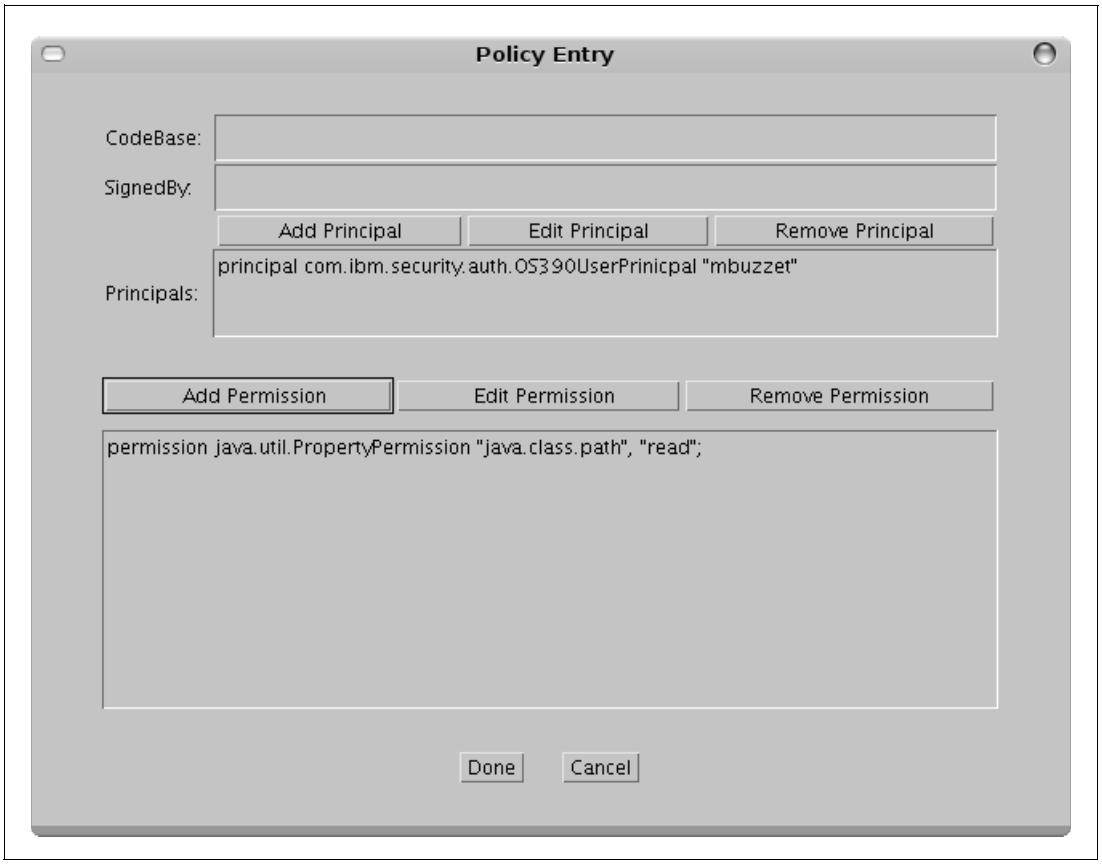


Figure 2-3 The Policy Tool entry dialog box

In this example, we add the permission to read the java.class.path property. Figure 2-4 shows the dialog box used to add a principal.

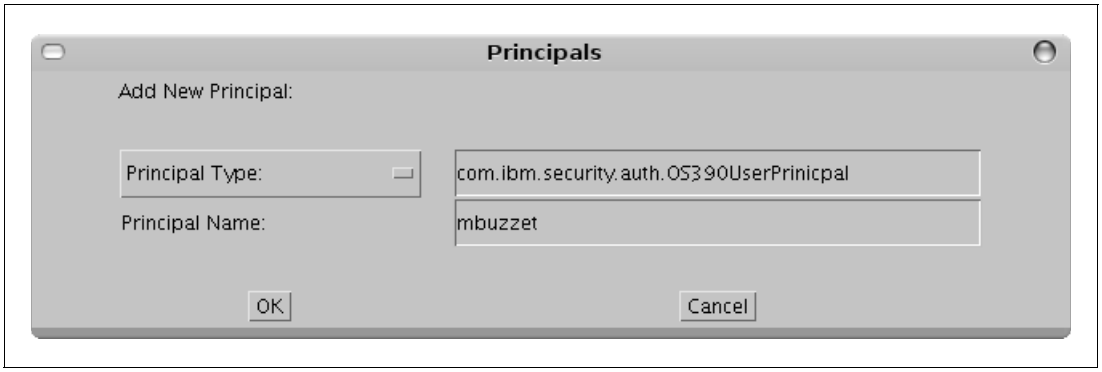


Figure 2-4 The principals dialog box

2.4 Performance issues

Because Java 2 Security implements an additional layer of security mechanisms on top of the operating system's own security mechanisms, the performance overhead of Java 2 security's authorization support might prove to be problematic. Whenever code is loaded or executed, the Java platform must ensure that the calling identity has authorization to perform the requested action. Java applications tend to utilize a large number of both built-in and third

party-supplied packages. Each of these packages could require an extraordinary amount of permission checks.

Many Java 2 Enterprise Edition (J2EE) Web and Enterprise Java Bean (EJB™) containers ship with Java 2 security turned off. With WebSphere Application Server, for instance, the documentation generally advises using against Java 2 security if performance is an issue. Instead, it suggests using thorough code reviews and setting proper access controls in the underlying operating system.

Part 2

Platform-level security with z/OS Java

This part addresses the z/OS security services that are made directly available to Java applications so they can benefit or even contribute to the management of the Identification, Authentication, and Authorization services provided in the z/OS platform.

© Copyright IBM Corp. 2008. All rights reserved.

3

Introduction to z/OS Resource Access Control Facility

This chapter provides a brief introduction to z/OS Security Server Resource Access Control Facility (RACF) External Security Manager to give readers a level set before progressing on to the following chapters.

For more general information about RACF, refer to:

- *z/OS Security Server RACF General User's Guide*, SA22-7685
- *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683

© Copyright IBM Corp. 2008. All rights reserved.

3.1 What is RACF

The Multiple Virtual Storage (MVS) operating system was designed in the 1970s to run on the System/370™ (S/370™), for the secure execution of multiple applications on one system by multiple users. RACF was introduced in 1976 to work on top of the robust hardware and operating system layers to provide applications with a common point of user authentication and access control services. At that point it was termed an “External Security Manager”, because security management was becoming “external” to the applications.

Since then, RACF has grown and been adapted to meet the specific needs and challenges of new security technologies supported by the OS/390 and z/OS operating systems. As of z/OS V1R10, RACF provides support for the following security services:

- User identification and authentication for z/OS users via password, PassTicket or password-phrase
- User identity mapping for z/OS users authenticated via X.509 V3 digital certificate or Kerberos ticket
- Resource access control for z/OS applications and components on the basis of the user’s identity or “groups of users” membership
- X.509 digital certificate management
- Kerberos user registry
- Remote security services via the z/OS LDAP server
- Support of the J2EE role security model
- Auditing of all security events detected by RACF or reported to RACF

3.2 RACF infrastructure for identification, authentication, and authorization

The RACF infrastructure is illustrated in Figure 3-1 on page 49. RACF maintains a user registry and authorization database that contains:

- Information pertaining to registered users, such as identities, passwords and other security-related attributes. Note that “user” here is a generic term that can designate an actual system user or a specific application that is running in the system under a “technical” identity. RACF also supports the concept of “groups of users” to designate a set of users with installation-related specific characteristics that allow you to treat them as a single entity.

  Information about users and groups of users are kept in database records called *profiles*. The label, or name, of a USER profile is the RACF identity of the user (user ID). The label of a GROUP profile is the group name.
- A definition of the resources to be protected, in *resource profiles*. Each resource profile contains a list (the *access list*) of users permitted to access the resource and, if needed, the access given by default to the resource. RACF also supports an optional Multilevel Security (MLS) security model. When MLS is active, access to a resource also depends on the *security label* assigned to the resource, in its profile, as well as its access requestor’s own security label.

  A RACF-protected resource is designated by its nature (the *class)* and its profile name.

The profiles in the RACF database are defined and maintained by RACF administrators or users declared as “owner” of the resources.

Note: RACF also supports controlling access for certain kinds of resources that are not registered with profiles in the RACF database. z/OS UNIX resources are examples of such types of resources.

However, RACF still creates the data structures where the access control information is kept (for example, for the z/OS UNIX file File Security Packet (FSP), which RACF creates in the file system). But it the resource manager ‘s responsibility to provide RACF with the information needed to allow it to make an access control decision and generate the relevant auditing data.

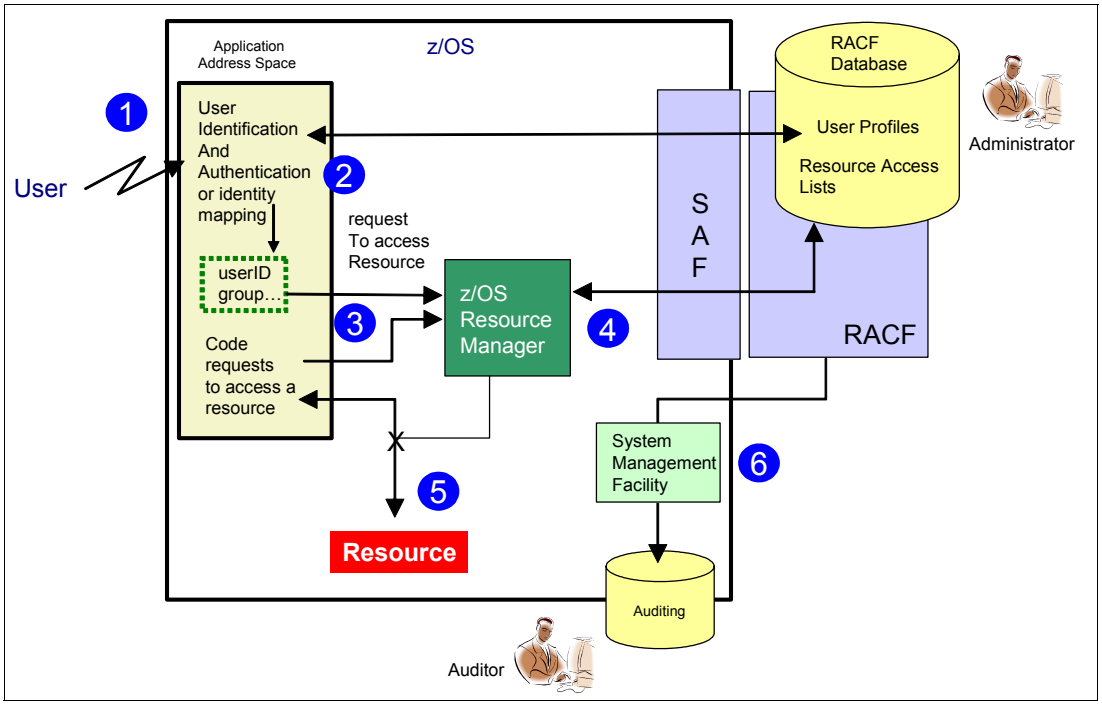


Figure 3-1 The RACF infrastructure

The following interactions occur in the infrastructure shown in Figure 3-1:

1. An application, or a z/OS component, requests the user to supply identification and authentication data. z/OS provides support for several forms of identification and authentication including a user identity and a password, or a digital certificate or a Kerberos ticket.

   If the user provides a user ID and a password to the requesting application, the application calls RACF to evaluate the validity of the user ID and the password.

   If the user authenticates with a digital certificate or a Kerberos ticket, the application must proceed with validating the certificate or ticket. After validation, the application calls RACF to obtain a RACF user ID that maps to the certificate or the Kerberos ticket.

   RACF responds to the request with Return and Reason codes, and with a RACF user ID in the case of a successful mapping request. These codes indicate whether or not the operation was successful in RACF.

2. The application now runs under a RACF user ID that is used to access a resource in the system, along with the list of user groups that this specific user ID belongs to.

3. In z/OS, access to resources is mediated by specialized z/OS components called *resource managers*. It is the role of the resource manager to call RACF to verify whether the user can be granted access to the resource.
4. The resource manager also grants or denies access to the application for the resource, according to the response from RACF.
5. Authentication accesses to resources and other security events can be recorded in an audit trail comprised of z/OS SMF records.

Note that specific RACF users can be given the AUDITOR attribute and can define auditing criteria. RACF can enforce a separation of duties with respect to security administration and auditing by allowing the administrator to change access control rules and user definitions, but not to change certain auditing controls.

Conversely, the auditor can change auditing controls but cannot make administrative changes to RACF profiles. In other words, administrators cannot prevent auditors from auditing them, and auditors cannot authorize themselves to resources.

3.2.1 The System Authorization Facility interface

System Authorization Facility (SAF) is the interface provided in z/OS for applications to ask for External Security Manager services. SAF can be invoked in two ways:

- By using the RACROUTE macro instruction in the program that needs to call RACF functions.
- By using RACF Callable Services. These callable services are usable by applications running in a variety of execution environments, such as Language Environment or Java. Thus, these applications can also benefit from the External Security Manager centralized security implementation.

SAF also allows customization of service requests processing by allowing the installation of a customer exit module that takes control before the request is forwarded to the External Security Manager.

Finally, SAF allows you to install and use a External Security Manager other than RACF with minimum changes to the applications. For example, the CA ACF2 or Top Secret are such alternate products.

3.2.2 RACF user, group, and resource profiles

RACF allows you to define the users who can access protected resources. It records the information about the users in user profiles, and maintains this information in the RACF database (along with all other profiles).

Figure 3-2 on page 51 illustrates a USER profile. Each RACF-defined user has a USER profile.

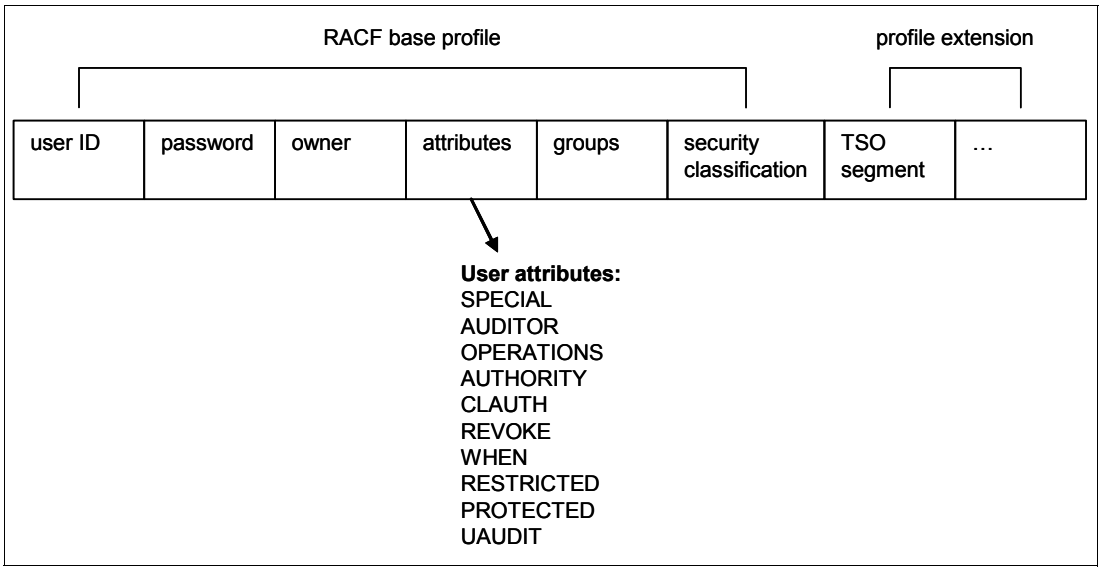


Figure 3-2 User profile

The name of the profile is the user ID itself. The password entry is kept one-way encrypted in the USER profile. The owner of a profile can change the profile. The user attributes define the responsibilities, authorities, or restrictions that a specific user has while defined to the system. Every user belongs, at a minimum, to one default group. The security classification determines the user's ability to access resources classified at security levels. Segments are optional extensions to the RACF profile and are used to save information for the resource managers like TSO or CICS.

As previously mentioned, with RACF, all defined users belong to at least one group, known as a *default group*. A group is a collection of RACF users who share common access requirements to protected resources or who have similar attributes within the system. Groups are intended for administrative convenience. Note that RACF groups form a hierarchical or “tree” administrative structure, where each group is “owned” by a superior group. Groups can also “own” resources, as well as users and other groups.

Figure 3-3 shows a GROUP profile in RACF.

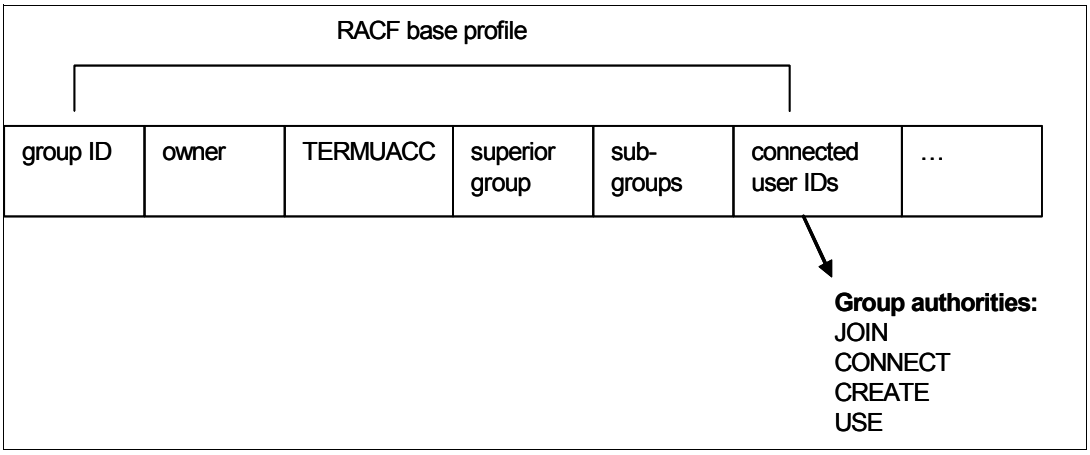


Figure 3-3 Group profile

A RACF user can be a member of more than one group. In RACF terminology, the users are connected to that group. A group owner, usually the user who defined the group to RACF, can define and control the other users connected to the group. The group owner can also delegate various group administrative responsibilities and authorities to other users connected to the group. The connect information in a user profile describes what the user can do when connected to that group, by including group authorities and group-related user attributes, as well as other information.

Resources, that is datasets or general resources, are protected using resource profiles. Figure 3-4 shows such a resource profile.

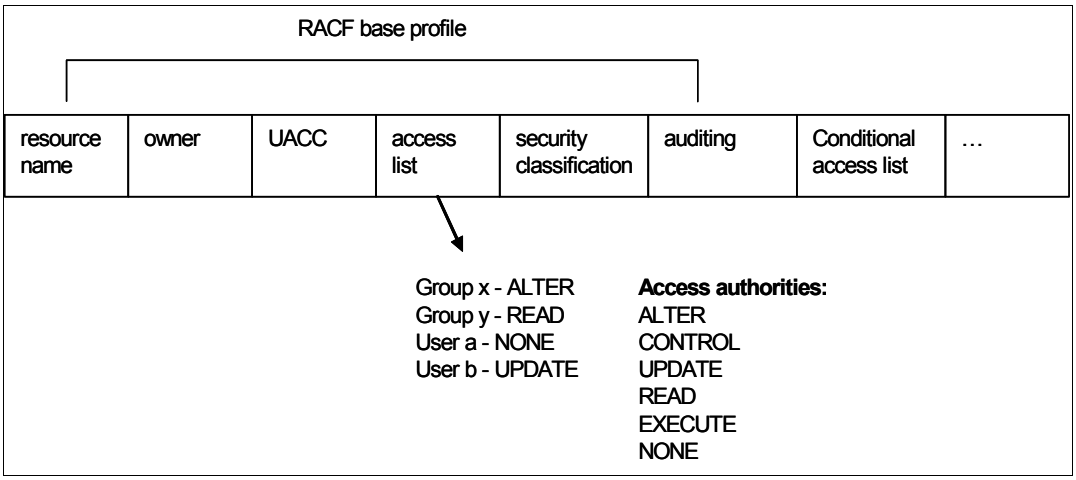


Figure 3-4 Resource profile

The name of the profile equals the name of the resource, and generic names are allowed. Using generic names, one profile can protect multiple resources. The owner of a profile can modify it. The universal access authority (UACC) describes in which way everyone is allowed to access the resource. It is usually recommended to set the UACC to NONE. The access list describes which user or group can access the resource beyond the access level set in UACC.

3.2.3 RACF commands

RACF administration means adding, deleting, modifying, and listing profiles in the RACF database. Figure 3-5 on page 53 shows the appropriate commands.

The administration can be performed using TSO bottom line commands or ISPF panels.

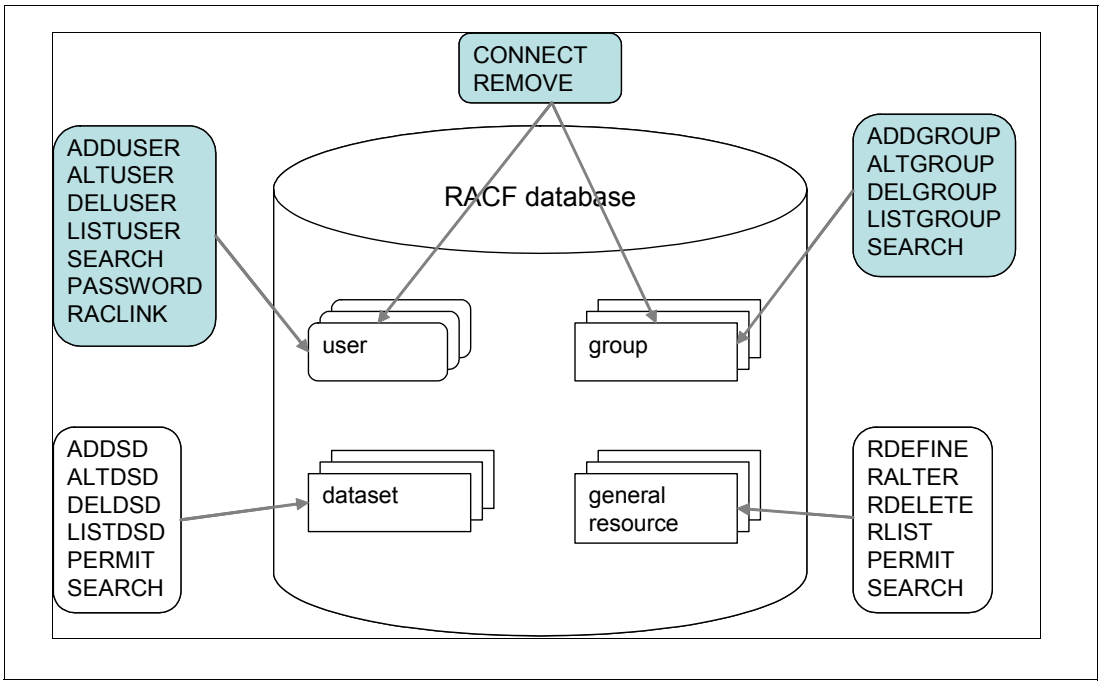


Figure 3-5 RACF commands

The highlighted boxes show the RACF commands for users and group administration.

The commands ADDUSER, ALTUSER, DELUSER, LISTUSER, SEARCH, PASSWORD and RACLINK are used to administer users in RACF, along with ADDGROUP. The commands ALTGROUP, DELGROUP, LISTGROUP, and SEARCH are used to administer groups. The commands CONNECT and REMOVE are used to create or delete connections between users and groups.

A user with the SPECIAL or group-SPECIAL attribute can issue all of the commands. A user with the AUDITOR or group-AUDITOR attribute can use only commands to search and list profiles, and to record RACF activities and events.

3.3 Accessing RACF using the LDAP protocol

The z/OS LDAP server supports providing access to RACF profiles or services to remote LDAP clients. The LDAP server must be running in the z/OS instance that hosts the target RACF system. The following back-ends are used, depending on the function requested by the client:

- The SDBM back-end allows access to USER and GROUP profiles in the RACF database, and the connection or removal of RACF users to or from RACF groups.

  The SDBM back-end is also used for Java Security Administration (JSec), as explained in Chapter 5, “Java Security Administration” on page 61.
- The GDBM back-end is used to indicate to an LDAP client that a change was made to a USER or GROUP profile, or to the connection of a user to a group, in the RACF database.
- The extended operation (EXOP) ICTX back-end interfaces with RACF for Remote Authorization, Remote Auditing, and Identity Cache services.

RACF supports the *Password Enveloping* function that allows an authorized LDAP client to securely extract a RACF user password out of the RACF database using the LDAP protocol. In z/OS V1R10, the function is expanded to provide Password Phrase enveloping.

3.3.1 Administering RACF users and groups through LDAP

The z/OS LDAP and RACF infrastructure which is used in that case is shown in Figure 3-6. It uses the LDAP server and the SDBM back-end. Note that it is sufficient to activate the SDBM back-end only to provide the required functions.

The SDBM back-end requires the user to authenticate, with an LDAP-authenticated “bind” performed by presenting a RACF user ID and password which are then verified by RACF.

After the user has been authenticated, the SDBM actually converts the client’s LDAP commands into RACF TSO commands, which are run with exactly the same RACF privileges as if the user were working from a TSO terminal. The responses are likewise converted so that they are issued in the proper LDAP syntax and format.

Note that there are no specific controls to access this function beside the user being successfully authenticated by RACF and owning the proper RACF privileges to operate on USER and GROUP profiles.

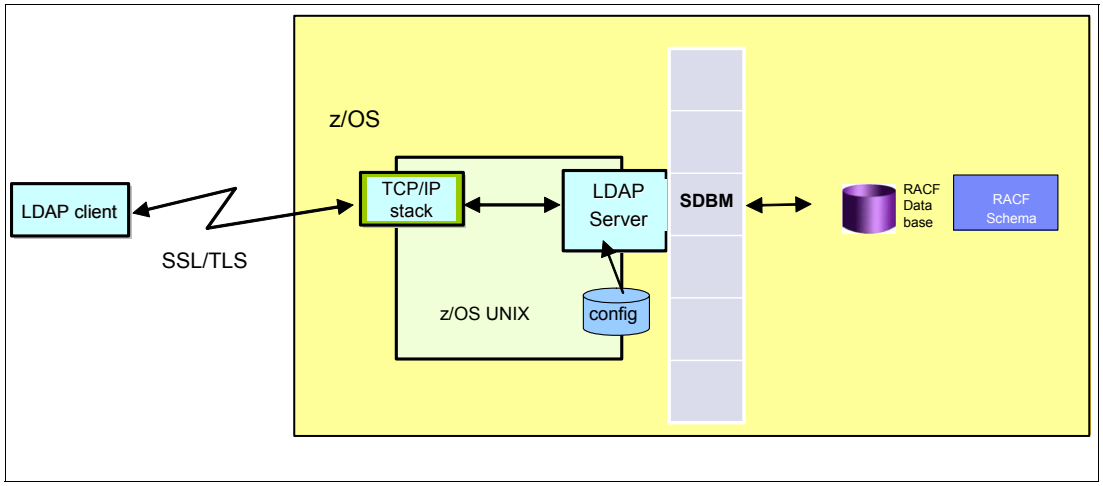


Figure 3-6 Administering RACF users and groups with LDAP

4

System Authorization Facility interfaces in z/OS Java

This chapter describes the System Authorization Facility (SAF) interfaces available to Java applications running on z/OS. It provides detailed information about the provided classes and how to use them, along usage examples.

© Copyright IBM Corp. 2008. All rights reserved.

4.1 System Authorization Facility interfaces in Java - overview

SAF interfaces in Java can handle access control to z/OS resources. They provide an additional set of security APIs. These APIs are implemented through Java classes wrapping z/OS UNIX Services via JNI. The z/OS UNIX Services are in turn handled by a Security Server for z/OS that implements SAF interfaces (such as IBM RACF).

Applications that use these APIs do not have to be APF authorized.

The classes provided are:

- PlatformAccessControl
- PlatformThread
- PlatformSecurityServer
- PlatformAccessLevel
- PlatformReturned
- PlatformUser

The methods of these classes allow a Java application to:

- Check whether the Security Server or a specific security server class is active
- Extract the user ID in effect for the current running thread
- Check the user ID in effect for access rights to a resource
- Authenticate a user ID and password
- Check whether a user ID is a member of a group
- Change a user's password

Because the SAF classes were first released with Java 1.1.6, they can be used to implement code-based Java 1 security.

Important: It appears that the use of SAF classes is not compatible, due to the exploitation of JNI, with an environment where Java 2 Security is turned on.

4.2 Installation of SAF classes

The SAF classes are packaged in com.ibm.os390.security.*. With Java 5.0, this package comes with the jar file security.jar which is installed under the /usr/lpp/java/lib/ directory when you have installed the JVM for z/OS. With Java 6.0, this package is in the RACF.jar which is also placed in the /usr/lpp/java/lib/ directory.

4.3 The classes in detail

The com.ibm.os390.security.* package consists of one interface and five classes.

To download the Java documentation of the SAF classes, refer to:

http://www-03.ibm.com/servers/eserver/zseries/software/java/j5security.html

You can find a complete Java class containing all examples shown below in Appendix B, “SAF sample code” on page 221.

4.3.1 PlatformAccessLevel

With z/OS Security Server (RACF), permissions to resources are granted using the READ, UPDATE, CONTROL or ALTER access levels. PlatformAccessLevel is an interface used as a placeholder for these named constants. The constants are used by the accessLevel parameter of methods in the PlatformAccessControl class.

4.3.2 PlatformReturned

This class is a helper used by the PlatformAccessControl and PlatformUser classes. It provides an output structure which is filled with various error codes and values by the called z/OS security service, such as an access control check.

The heart of the SAF interfaces in Java are four final classes, each of them representing a platform entity of z/OS. Methods of each class represent an API to manipulate this platform entity.

Note: These methods come as static methods intended to be invoked without the need to instantiate an object of the class.

4.3.3 PlatformSecurityServer

This class represents the z/OS Security Server (such as RACF). Methods of this class are called to confirm the active status of the Security Server as shown in Example 4-1.

Example 4-1 isActive

```
if (PlatformSecurityServer.isActive())
   System.out.println("Security Server is active.");
else {
   System.out.println("Error.");
   return;
}
```

Methods of this class are also called to confirm that a specific class of resources is active. Note that the resource type is case sensitive, so you have to type it in upper case letters, as shown in Example 4-2.

Example 4-2 resourceTypeIsActive()

```
if (PlatformSecurityServer.resourceTypeIsActive("FACILITY"))
   System.out.println("FACILITY is active");
else {
   System.out.println("Error: FACILITY is not active");
}
```

4.3.4 PlatformAccessControl

This class represents the access control function of the z/OS Security Server. It can be used to check the permission of the current user to a specific resource. If the current thread has a security context, the thread user ID is checked in an access control check. If not, the user ID associated with the current process is checked.

This class actually wraps the __check_resource_auth_np service() function of the z/OS C/C++ Run Time Library. For authorization to use this function, the caller must have read permission to the BPX.SERVER Facility class, or if BPX.SERVER is not defined, the caller must be a superuser (UID=0).

Note that the resource type and the resource name are case sensitive, so you have to type them in upper case letters, as shown in Example 4-3.

Example 4-3 checkPermission

```
PlatformReturned pr =
PlatformAccessControl.checkPermission("FACILITY","BPX.SERVER",PlatformAccessLevel.
READ);
if (pr == null)
   System.out.println("User has READ access to the resource named BPX.SERVER of
   resource type FACILITY");
else {
   System.out.println("An error ocurred ...");
   System.out.println( "success: " + pr.success +
         "\nerrno: " + pr.errno +
         "\nerrno2: " + pr.errno2 +
         "\nerrnoMsg: " + pr.errnoMsg );
}
```

If the user is authorized for the specific resource, checkPermission() returns null; otherwise, pr is filled with error codes and messages.

If in this example the user has no READ access to the resource, pr would return the following output:

```
success: false
errno: 139
errno2: 154665176
errnoMsg: EDC5139I Operation not permitted.
```

The method checkPermission() can also be used to check whether a specific user has permission to a resource, as shown in Example 4-4. In this case, the user identity is specified as a parameter for this method.

Example 4-4 checkPermission for a specified user

```
PlatformReturned pr =
PlatformAccessControl.checkPermission("anyUser","FACILITY","BPX.SERVER",PlatformAc
cessLevel.READ );
if (pr == null)
   System.out.println("anyUser has READ access to the resource named BPX.SERVER of
   resource type FACILITY");
else {
   System.out.println("An error ocurred ...");
   System.out.println( "success: " + pr.success +
```

```
      "\nerrno: " + pr.errno +
      "\nerrno2: " + pr.errno2 +
      "\nerrnoMsg: " + pr.errnoMsg );
}
```

4.3.5 PlatformThread

This class wraps z/OS UNIX thread level functions. It represents a UNIX thread that is mapped onto an z/OS task (TCB). The method getUserName is useful to extract the user associated with the current thread, as shown in Example 4-5. The method getUserName wraps the BPX1ENV UNIX System Services Assembler callable service.

Example 4-5 getUserName

```
String currentUser = PlatformThread.getUserName();
System.out.println("CurrentUser is:"+currentUser);
```

4.3.6 PlatformUser

This class represents a z/OS user ID, which is the basis for z/OS platform authentication and access control. The parameters for user, password, and group are *not* case-sensitive.

The authenticate method validates a user ID and a password as shown in Example 4-6. It wraps the _passwd() function.

Example 4-6 authenticate

```
PlatformReturned pr = PlatformUser.authenticate("anyUser","password");
if (pr == null)
   System.out.println("Password check was successful");
else {
   System.out.println("An error ocurred ...");
   System.out.println( "success: " + pr.success +
         "\nerrno: " + pr.errno +
         "\nerrno2: " + pr.errno2 +
         "\nerrnoMsg: " + pr.errnoMsg );
}
```

The isUserInGroup method checks whether a user is a member of a specific user group, as shown in Example 4-7.

Example 4-7 isUserInGroup

```
boolean isMember = PlatformUser.isUserInGroup("anyUser,anyGroup");
if (isMember)
   System.out.println("anyUser is a member of anyGroup");
else {
   System.out.println("anyUser is not a member of anyGroup");
}
```

The changePassword method is used to change a user's password. The user ID and the current password are validated before the password is changed to the new value; see Example 4-8 on page 60.

Example 4-8 changePassword

```
PlatformReturned pr = PlatformUser.changePassword("anyUser","oldPwd","newPwd");
if (pr == null)
   System.out.println("Password change successful.\n");
else {
   System.out.println("An error ocurred ...");
   System.out.println( "success: " + pr.success +
         "\nerrno: " + pr.errno +
         "\nerrno2: " + pr.errno2 +
         "\nerrnoMsg: " + pr.errnoMsg );
}
```

Note: The functions wrapped by the methods of PlatformUser and PlatformAccessControl require that all modules within the address space must be loaded from a program controlled library when the BPX.DAEMON resource is defined in the RACF FACILITY class. Ask your system administrator to properly assign the program controlled status to the required libraries.

When writing this chapter we found out that the following libraries had also to be program controlled:

- Program EAGRTPRC from library REXX™.SEAGALT
- Program EX from library SYS1.CMDLIB
- Program ISPQRY from library ISP.SISPLOAD
- Program ISPCFIGU from library SYS1.USER.LOAD
- HFS program /bin/fomtlinp

If you are using the SAF interface with JZOS batch launcher, then you also have to set the JZOS PDS load libraries program controlled.

Program Control and APF authorization are explained in 1.5, “Java and z/OS security” on page 17.

5

Java Security Administration

This chapter explains what the Java Security Administration (JSec) API is, and how it can be exploited in a context where z/OS RACF is the user registry.

With this Java API, users who have the proper RACF privileges can develop their own Java security administration applications to define users and groups in RACF. Furthermore, you can query whether a user belongs to a group, or obtain other user information from the security repository from within a Java application.

© Copyright IBM Corp. 2008. All rights reserved.

5.1 Overview of Java Security Administration

Java Security Administration provides a Java interface to allow administration of users and groups in security repositories. It also provides a RACF-specific implementation. The support for z/OS version 1 release 9 and above includes the ability to query users and groups from z/OS RACF or other non-z/OS security mechanisms, or interface with Java programs, as a call or a check of security credentials.

This support provides two components:

- A generic Java interface, which could be used with any security provider
- A RACF-specific implementation

Figure 5-1 illustrates how a Java application accesses an active instance of RACF. The Java application uses the JSec API that is running in a JVM and using JNDI to communicate over TCP/IP to an LDAP server, with the SDBM back-end active, in the z/OS image hosting the RACF instance.

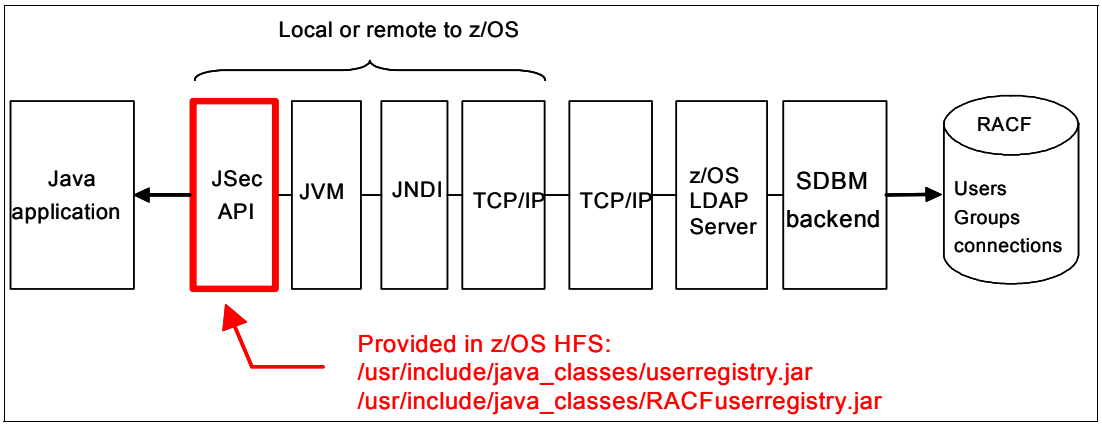


Figure 5-1 Java application accessing an active RACF instance

The API is designed to be extensible for future RACF enhancements or for use with other security repositories. The generic interface would need another security provider-specific implementation in order to be used with another security repository.

The JSec API provides for easy mapping between this interface and the user interface constituted with the RACF TSO commands: ADDUSER, ALTUSER, ADDGROUP, ALTGROUP, CONNECT and REMOVE.

Note: RACF uses the term CONNECT to add a user to a group. However, the Java interface uses terms such as add member and membership attributes, which correspond to the options on the CONNECT command.

The implementation details of Java security administration and description of classes and methods used are documented at:

`http://www.ibm.com/servers/eserver/zseries/zos/racf/racfjsec/doc/index.html`

A compressed version of the javadoc can be downloaded using the following link:

`ftp://ftp.software.ibm.com/eserver/zseries/zos/racf/jsec/JSec_javadoc.zip`

5.2 Installation

The requirements for using Java Security Administration are:

- z/OS version 1 release 9 or above
- Java (IBM 31-bit SDK for z/OS Java 2 Technology Edition, V5)
- The z/OS Integrated Security Services LDAP Server or the IBM Tivoli Directory Server for z/OS configured and running with the SDBM back-end.

  Refer to *IBM Tivoli Directory Server Administration and Use*, SC23-5191, or *z/OS Integrated Security Services LDAP Server Administration and Use*, SC24-5923, for setup details.

> **Note:** The JSec API works with both of these LDAP servers, but there is a limitation to the z/OS Integrated Security Services LDAP. Its SDBM version is limited to get a maximum of 4096 lines as a response to a RACF command. This may be an issue only in very specific cases like listing the users or groups in a large RACF DB.
>
> The new IBM Tivoli Directory Server in z/OS 1.8 has no such limitation.

The Java security code is packaged into two jar files in HFS:

- /usr/include/java_classes/userregistry.jar (this contains the Java interface)
- /usr/include/java_classes/RACFuserregistry.jar (this contains the RACF-specific implementation)

5.3 Java classes used

Table 5-1 lists and describes which Java classes are used to implement the JSec API.

Table 5-1 Overview of Java classes used

Java interface or class	Description
java.security.Principal	Interface which can be used to represent any entity, such as an individual, a company, or a logon user ID.
java.security.acl.Group	Interface which represents a group of principals.
javax.naming.directory.BasicAttribute	Class which provides a name/value or name/values pairing. Attribute names for the JSec API follow the format of segmentname_keyword, such as OMVS_UID. Attributes are case-insensitive.
javax.naming.directory.BasicAttributes	Class which is a collection of BasicAttribute objects.
javax.naming.directory.ModificationItem	Class which is used to modify users, groups, or memberships.

5.4 Interface definitions

As mentioned in 5.1, “Overview of Java Security Administration” on page 62 the JSec API consists of two different components. The following paragraphs provide information about the

Java interfaces and classes in `com.ibm.security.userregistry.*` that can be used for any security provider.

5.4.1 User

The interface `ibm.com.security.userregistry.User` extends `ibm.com.security.Principal`. This interface offers methods to:

- Get the attributes for the user from the security repository using `getAttributes()`
- Get a list of all user groups the user is a member of, using `getGroups()`
- Get the name of the user or its userID using `getName()`

5.4.2 UserGroup

The interface `ibm.com.security.userregistry.UserGroup` extends `ibm.com.security.acl.Group` and is intended to be a group of individual users. Implementations of this interface can be designed to support adding groups as members of other groups or not.

Note: RACF does not support groups being members of other groups.

This interface offers methods to:

- Add a member to the group using `addMember()`
- Get the attributes for the group from the security repository using `getAttributes()`
- Get the membership attributes for a user in the group using `getMembershipAttributes()`
- Get the name of the group using `getName()`
- Check if a user is member of the group or not, using `isMember()`
- Get a list of all members in the group using `members()`
- Modify membership attributes for a user in the group in the security repository using `modifyMembershipAttributes()`
- Remove a user from the group using `removeMember()`

5.4.3 SecAdmin

A very important interface is `com.ibm.security.userregistry.SecAdmin`. This interface is the primary interface in the security administration of users and groups. It provides the following methods:

- `createUser()` and `createGroup()` to define a new user or group in the security repository
- `deleteUser()` and `deleteGroup()` to delete an existing user or group from the security repository
- `getUser()` and `getGroup()` to instantiate a User or UserGroup object to work with in the application
- `modifyUser()` and `modifyGroup()` to change, add or delete attributes for an existing user or group

5.4.4 SecAdminException

This class is implemented for all exceptions thrown by the interfaces previously defined.

5.5 RACF implementing classes

The second component of JSec is the RACF-specific implementation of the Java interfaces defined in 5.4, “Interface definitions” on page 63. These implementing classes provide extra methods which are described in the following sections.

All classes are packaged in `com.ibm.eserver.zos.userregistry.*`.

5.5.1 RACF_User

The class `ibm.com.eserver.zos.racf.userregistry.RACF_User` implements `com.ibm.security.userregistry.User`. The RACF_User object has no constructor. That is, a User object must be obtained by calling the `getUser()` or `createUser()` method of `RACF_SecAdmin`. To use the methods provided in this class, the User object must be cast to a RACF_User object. An example would be:

```
((RACF_User) ibmuser)
```

This class offers two utility methods to:

- Obtain user attributes and their properties in a BasicAttributes object using `attributesInfo()`
- Display an HTML table of user attributes and their properties using `attributesHTML()` as shown in Table 5-1 on page 63

  This method returns a String which has to be put in a Web browser in order to actually see the HTML table. For each attribute, the table displays the name and a description, whether it can be modified, whether it is a segment, whether it is a Boolean attribute, and whether it can have multiple values.

  A Boolean attribute is an attribute that does not have a particular value. The user either has this attribute or not. An example would be BASE_SPECIAL. A user either has the SPECIAL attribute or it does not. This can be contrasted to, for example, OMVS_UID where the caller would assign a specific value to the OMVS_UID attribute.

Table 5-2 Sample output from RACF_User.attributesHTML()

User attributes					
Attribute name	**Description**	**Modifiable**	**Segment**	**Boolean attribute**	**Multi-value attribute**
BASE_ADSP	All permanent tape and DASD data sets created by user are automatically RACF-protected by discrete profiles.	Yes	No	Yes	No

User attributes					
Attribute name	**Description**	**Modifiable**	**Segment**	**Boolean attribute**	**Multi-value attribute**
BASE_AUDITOR	Indicates user has full responsibility for auditing the use of system resources, and is able to control the logging of detected accesses to any RACF-protected resources during RACF authorization checking and accesses to the RACF database.	Yes	No	Yes	No
BASE_CATEGORY	Names of installation-defined security categories, which must be defined as members of the CATEGORY profile in the SECDATA class.	Yes	No	No	Yes
BASE_CLAUTH	Classes in which user is allowed to define profiles to RACF for protection. Classes can be USER, and any resource classes defined in the class descriptor table.	Yes	No	No	Yes
BASE_CREATED	The date this user was defined to RACF.	No	No	No	No
Many more	These are the first five attributes of over 100.				

All possible user attributes and descriptions are listed in Appendix D, “JSec attributes” on page 235.

5.5.2 RACF_Group

The class `ibm.com.eserver.zos.racf.userregistry.RACF_Group` implements `com.ibm.security.userregistry.UserGroup`. The RACF_Group object has no constructor. That is, a UserGroup object has to be obtained by calling `getGroup()` or `createGroup()` method of `RACF_SecAdmin`. To use the methods provided in this class, the UserGroup object has to be cast to a RACF_User object. An example would be:

```
((RACF_Group) ibmgroup)
```

This class offers four utility methods:

- `attributesInfo()` and `membershipAttributes()` both return a BasicAttributes object containing either group attributes and their properties or membership attributes and their properties.
- `attributesHTML()` and `membershipAttributesHTML()` both return an HTML table of group or membership attributes and their properties.

  All possible group and membership attributes and their description are listed in Appendix D, “JSec attributes” on page 235 under “Group attributes” on page 250 and “Membership attributes” on page 251.

5.5.3 RACF_SecAdmin

The class `ibm.com.eserver.zos.racf.userregistry.RACF_SecAdmin` implements `com.ibm.security.userregistry.SecAdmin`. This class offers:

- A method to create a user with the same attributes as an existing user using `clone_user()`. Actually not all attributes are copied from the existing user, for different reasons.
  - BASE_PASSWORD and PROXY_BINDPW are not duplicated for security reasons.
  - BASE_UAUDIT is not allowed to be set on a user ID creation.
  - KERB_KERBNAME, LNOTES_SNAME and NDS_UNAME must be unique values across the system.
  - Each user should have a unique OMVS_UID.
  - `clone_user()` does not make the new user a member of any group other than the default group.
- Utility methods to sort attributes alphabetically by name using `BasicAttributesToTreeMap()` and to display all attributes, one on each line in alphabetical order using `displayAttributes()`

5.5.4 RACF_remote

The RACF_remote object defines the connection parameters of a remote or local RACF to be accessed via the z/OS LDAP server and SDBM back-end. A RACF_remote object is constructed with five strings:

- The connection URL for the z/OS LDAP server

  Like all URLs, this follows the convention: prototype://hostname:portnumber. An example of a connection URL would be:

  `"ldap://wtsc60.itso.ibm.com:389"`

- The connection mode

  It should be either set to “simple” for using LDAP or to “secure” for using LDAPS (that is, LDAP over an SSL/TLS connection). If it is null, “simple” will be used.

- The connection principal

  This is the RACF user ID to be used for LDAP bind. All RACF operations will be done using this user’s authority after a successful bind.

- The connection credentials

  This is the password of the RACF user ID used as connection principal.

- The connection suffix

  This is the SDBM back-end suffix; that is, the root of the LDAP naming space assigned to the RACF directory. The suffix appears as the last part of the distinguished name for any RACF “LDAP entry”.

Note: Refer to *IBM Tivoli Directory Server Administration and Use for z/OS*, SC23-5191, or *z/OS Integrated Security Services LDAP Server Administration and Use*, SC24-5923, for information about installing and setting up the related functions.

5.6 Usage and invocation

Because JSec is native Java code, it can be run on a variety of platforms. It can be run on z/OS or downloaded and run on any Java-capable platform. Similarly, it can be invoked from Java code on any machine with the Java Virtual Machine and a TCP/IP connection. Also, the z/OS system with RACF database must have the LDAP server with the SDBM back-end configured and running.

Note: Regardless of whether you are running the code on or off platform, JSec always uses LDAP as a connection layer.

To use the native Java interface with another security manager or on a system other than z/OS, you need to download the two jar files provided in the HFS file using FTP. Afterwards, you must set the classpath to the location where you placed the downloaded files. For example, on a personal computer this would be:

```
set CLASSPATH=.\userregistry.jar;.\RACFuserregistry.jar;
```

5.6.1 Running Java security code on z/OS

To run Java security code on z/OS, you need to set CLASSPATH to pick up jar files, such as:

```
export CLASSPATH=$CLASSPATH:/usr/include/java_classes/userregistry.jar:
/usr/include/java_classes/RACFuserregistry.jar
```

5.7 Sample code

The following link contains a zip file where you can access and use various code samples. Also in this zip file is a file called ShowAttributes.html, which contains all User, Group and Membership attributes. It is an important reference to have as you begin coding.

ftp://ftp.software.ibm.com/eserver/zseries/zos/racf/jsec/JSec_Webpages.zip

The examples contained in this zip file are also listed in Appendix D, “JSec attributes” on page 235. An overview of all attributes for users, groups, and memberships is provided in Appendix D, “JSec attributes” on page 235.

5.7.1 Simple program

Example 5-1 illustrates a very simple program to obtain the attributes for a user, as the TSO RACF command LISTUSER would provide.

Example 5-1 Simple program

```
01    import javax.naming.directory.BasicAttributes;
02    import com.ibm.eserver.zos.racf.userregistry.*;
03    import com.ibm.security.userregistry.SecAdmin;
04    import com.ibm.security.userregistry.User;
05
06    public class Sample {
07
08       public static void main(String[] args)
09       {
```

```
10          RACF_remote remote = new RACF_remote("ldap://wtsc60.itso.ibm.com:389",
11             "simple",
12             "IBMUSER",
13             "SECRET",       // password during testing
14             "o=itsoracf");
15
16          try
17          {
18             SecAdmin racfAdmin = new RACF_SecAdmin(remote);
19             if (racfAdmin != null)
20             {
21                User ibmuser = racfAdmin.getUser("IBMUSER");
22                BasicAttributes ibmuser_attr = ibmuser.getAttributes();
23                System.out.println("Attributes returned for IBMUSER are: ");
24                RACF_SecAdmin.displayAttributes(ibmuser_attr);
25             }
26          }
27          catch (Exception e)
28          {
29             System.out.println("Exception in Sample.java " + e.getMessage() +
               "\n");
30             e.printStackTrace();
31          }
32       }
33    }
```

The first four lines show the necessary import statements.

The RACF_remote object containing the connection information is created in line 10. It includes the hostname, the information that a simple, unsecured connection is used, the user ID and passwords well as the target SDBM suffix.

Note: The user ID provided in the RACF_remote object is the user ID that effectively executes the RACF commands. It is therefore important to ensure that this user ID possesses the proper RACF privileges.

If an incorrect password is entered, line 29 will produce the following error message:

```
com.ibm.security.userregistry.SecAdminException: Error initially connecting to
LDAP/SDBM.  Original error: [LDAP: error code 49 - R000104 The password is not
correct or the user id is not completely defined (missing password or uid)
(srv_authenticate_native_password)]
```

At line 18, the RACF_remote object is provided as a parameter to the RACF_SecAdmin constructor. This means that a connection to the LDAP server is required.

If the RACF_SecAdmin object was created successfully (line 19), then the getUser method returns a User object (line 21) and the getAttributes method returns the user's attributes (line 22).

Finally, these attributes are displayed in line 24 using the RACF_SecAdmin.displayAttributes method.

The output should look like in Example 5-2 on page 70.

Example 5-2 Output of simple program

```
Attributes returned for IBMUSER are:
BASE_CREATED: 06/06/08
BASE_DAYS: SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
BASE_DFLTGRP: SYS1
BASE_LAST-ACCESS: 06/25/08/13:02:32
BASE_OPERATIONS: No values <-- Boolean attribute
BASE_OWNER: IBMUSER
BASE_PASS-INTERVAL: 30
BASE_PASSDATE: 06/17/08
BASE_PASSWORD: Password Exists
BASE_SECLABEL: SYSMULTI
BASE_SPECIAL: No values <-- Boolean attribute
BASE_TIME: ANYTIME
BASE_USERID: IBMUSER
OMVS_PROGRAM: /bin/sh
OMVS_UID: 0
```

Most attributes have one or more values. For example, the first attribute, BASE_CREATED, which is the date the user was created, has a value of 06/06/08.

The second attribute, BASE_DAYS are the days of the week the user can access the system. It has seven values (‘SUNDAY’, ‘MONDAY’, ‘TUESDAY’, ‘WEDNESDAY’, ‘THURSDAY’, ‘FRIDAY’, ‘SATURDAY’).

However, there are Boolean attributes, where the user either has the attribute or does not have the attribute, such as ‘BASE_SPECIAL’. When displayed in the preceding example, you see:

```
BASE_SPECIAL: No values
```

This indicates the user has the BASE_SPECIAL attribute, meaning the user is allowed to issue all RACF commands with all operands except the operands that require the AUDITOR attribute. If the user did not have the BASE_SPECIAL attribute, then you would not see BASE_SPECIAL at all when displaying the attributes.

5.7.2 Create groups and members

Example 5-3 illustrates how to create groups and users in RACF using the JSec API. This refers to the TSO RACF commands ADDGROUP, ADDUSER and CONNECT. Furthermore, it explains how to modify attributes for existing users. It also explains how to delete users from the security repository, which refers to the RACF commands ALTUSER and REMOVE.

First, the import statements on line 1 to 4 are again required. Likewise, you must create the RACF_remote object to connect to the LDAP server (line 14).

Example 5-3 Create groups and users

```
01    import java.util.Enumeration;
02    import javax.naming.directory.*;
03    import com.ibm.eserver.zos.racf.userregistry.*;
04    import com.ibm.security.userregistry.*;
05
06    public class CreateGroupsAndMembers {
07
08       public static void main(String[] args)
```

```
09      {
10         SecAdmin racfAdmin = null;
11         UserGroup savants = null;
12         User savant;
13
14         RACF_remote remote = new RACF_remote("ldap://9.12.4.18:389",
15               "simple",
16               "IBMUSER",
17               "SECRET",       // password during testing
18               "o=itsoracf");
19
20         try
21         {
22            racfAdmin = new RACF_SecAdmin(remote);
23         }
24         catch (SecAdminException e)
25         {
26            System.out.println("Unable to connect to specified RACF database.
              "+e.getMessage());
27            return;
28         }
29
30         try
31         {
32            savants = racfAdmin.createGroup("savants", null);
33            System.out.println("We just created a group called savants.");
34         }
35         catch (SecAdminException e)
36         {
37            System.out.println("Unable to create group 'savants'.
              "+e.getMessage());
38            return;
39         }
40
41         try
42         {
43            System.out.println("Now we are going to add some members.");
44            savants.addMember(racfAdmin.createUser("Ohm",null));
45            savants.addMember(racfAdmin.createUser("Humboldt",null));
46            savants.addMember(racfAdmin.createUser("Curie",null));
47            savants.addMember(racfAdmin.createUser("Roentgen",null));
48            savants.addMember(racfAdmin.createUser("Bashful",null));
49            savants.addMember(racfAdmin.createUser("Bohr",null));
50            savants.addMember(racfAdmin.createUser("Einstein",null));
51         }
52         catch (SecAdminException e)
53         {
54            System.out.println("Exception trying to add members to group
              'savants'."+e.getMessage());
55            return;
56         }
57         System.out.println("savants Members:");
58         for (Enumeration ae = savants.members(); ae.hasMoreElements();)
59         {
60            User user = (User)ae.nextElement();
```

```
61             System.out.println(user.getName());
62          }
```

When you are connected to the SDBM back-end, you can create a new RACF group named SAVANTS with default attributes on line 32. Therefore, the SecAdmin.createGroup method is used. In line 44 to 50, seven members with default attributes are created and added to group SAVANTS. The lines 58 to 61 will display the names of all members of the group SAVANTS.

If the user ID provided in the RACF_remote object has insufficient authorizations to add groups to the security repository, the following error message will be produced by the code in line 37.

```
Unable to create group 'savants'. Error creating Group.  Original error: [LDAP:
error code 80 - ICH00007I INSUFFICIENT AUTHORITY TO SUPERIOR GROUP.]
```

If the group that you want to add already exists, the code in line 37 will produce the following error message:

```
Unable to create group 'savants'. Group savants already exists.
```

The same applies for adding users if there is insufficient authorization or a user already exists.

Assuming that the user Einstein is the leader of the group, that user should have the BASE_SPECIAL attribute. Example 5-4 illustrates how to modify the membership attributes of user Einstein.

Example 5-4 Modify attributes

```
63          try
64          {
65             System.out.println("Einstein is leader of the group, should be
                  SPECIAL.");
66             ModificationItem mods[] = new ModificationItem[1];
67             mods[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE,
68                   new BasicAttribute("BASE_SPECIAL"));
69             savant = racfAdmin.getUser("Einstein");
70             savants.modifyMembershipAttributes(savant,mods);
71          }
72          catch (SecAdminException e)
73          {
74             System.out.println("Error modifying membership attributes
               "+e.getMessage());
75             return;
76          }
77          try
78          {
79             BasicAttributes member_at =
                  savants.getMembershipAttributes(savant);
80             System.out.println("Membership attributes returned for Einstein
                  are: ");
81             RACF_SecAdmin.displayAttributes(member_at);
82
83             savant = racfAdmin.getUser("Bohr");
84             member_at = savants.getMembershipAttributes(savant);
85             System.out.println("Membership attributes returned for Bohr are:
               ");
```

```
86             RACF_SecAdmin.displayAttributes(member_at);
87          }
88          catch (SecAdminException e)
89          {
90             System.out.println("Error retrieving membership attributes
91             "+e.getMessage());
92             return;
93          }
```

How to declare, define, and use a ModificationItem array is shown in lines 66 to 70. The lines 79 to 86 get the membership attributes of the users Einstein and Bohr and display them.

The output from this running code is shown in Example 5-5. You can see that Einstein has an attribute BASE_SPECIAL. The user Bohr, which was created with default attributes and had not been modified, does not.

Example 5-5 Output of membership attributes

```
Membership attributes returned for Einstein are:
BASE_AUTHORITY: USE
BASE_CONNECT-DATE: 06/25/08
BASE_CONNECTS: 0
BASE_OWNER: IBMUSER
BASE_SPECIAL: No values
BASE_UACC: NONE
Membership attributes returned for Bohr are:
BASE_AUTHORITY: USE
BASE_CONNECT-DATE: 06/25/08
BASE_CONNECTS: 0
BASE_OWNER: IBMUSER
BASE_UACC: NONE
```

Finally, Example 5-6 illustrates how to remove user Roentgen from the group and delete the user ID Roentgen from the security repository.

Example 5-6 Delete user

```
94          try {
95             savant = racfAdmin.getUser("Roentgen");
96             savants.removeMember(savant);
97             racfAdmin.deleteUser("Roentgen");
98             System.out.println("savants after removing member Roentgen:");
99             for (Enumeration ae = savants.members();
100            ae.hasMoreElements();)
101            {
102               User user = (User)ae.nextElement();
103               System.out.println(user.getName());
104            }
105         } catch (SecAdminException e) {
106            System.out.println("Error deleting Roentgen from savants
               "+e.getMessage());
107         }
108      }
109   }
```

Line 96 shows how to remove Roentgen from the group SAVANTS using the UserGroup.removeMember method. To delete Roentgen from RACF, the SecAdmin.deleteUser method is used (line 97). The lines 99 to 103 will print out all members of the group SAVANTS, and Roentgen should not appear there anymore.

5.7.3 Change password

Example 5-7 illustrates how to change a password for an existing user.

First, the necessary import statements on line 1 to 3 are required. Then you must create the RACF_remote object again to connect to the z/OS LDAP SDBM back end (line 12). After that, connect to the RACF database using the SecAdmin object (line 20).

Example 5-7 Change password

```
01    import com.ibm.eserver.zos.racf.userregistry.*;
02    import com.ibm.security.userregistry.*;
03    import javax.naming.directory.*;
04
05    public class ChangePassword {
06
07       public static void main(String[] args)
08       {
09          SecAdmin racfAdmin = null;
10          User catuser = null;
11
12          RACF_remote remote = new RACF_remote("ldap://9.12.4.18:389",
13                "simple",
14                "IBMUSER",
15                "SECRET",       // password during testing
16                "o=itsoracf");
17
18          try
19          {
20             racfAdmin = new RACF_SecAdmin(remote);
21          }
22          catch (SecAdminException e)
23          {
24             System.out.println("Unable to connect to specified RACF database.
               "+e.getMessage());
25             return;
26          }
27
28          BasicAttribute pwd = new BasicAttribute("base_password");
29          pwd.add("wuff");
30          pwd.add("noexpired");
31
32          try {
33             System.out.println("Changing password for Cat");
34             ModificationItem mods[] = new ModificationItem[1];
35             mods[0] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE,pwd);
36             racfAdmin.modifyUser("cat",mods);
37
38          } catch (SecAdminException e){
39             System.out.println("Error changing password "+e.getMessage());
```

```
40              return;
41           }
42        }
43     }
```

To change a user's password, you must first create a new BasicAttribute object (line 28). In line 29 and 30, store the new password and the information noexpired in the BasicAttribute object.

Then create a ModificationItem which contains the information that an attribute has to be replaced and the new BasicAttribute (line 35). Now you can modify the user Cat with this ModificationItem (line 36).

6

RACF PassTicket generation and evaluation by z/OS Java applications

This chapter explains how Java applications on z/OS can generate and evaluate the RACF password substitute known as PassTicket.

© Copyright IBM Corp. 2008. All rights reserved.

6.1 PassTicket overview

The RACF PassTicket is a single-use password substitute that is generated by a program or function and used instead of a password during logon to a z/OS application. It is an alternative to the RACF password that removes the need to send RACF passwords across the network in clear text where they could be compromised. A PassTicket is only valid for 10 minutes after it is created, and may be used only once to log on to the application. After use, a PassTicket is no longer accepted as valid by the RACF instance running in the system[1].

This makes it more difficult to use a PassTicket that has been stolen from an insecure network. The short life span means that an attacker must be ready to use the PassTicket within minutes of its capture (before it expires), and in many cases, the stolen PassTicket was already on its way to being used, and will no longer be valid by the time the attacker attempts to connect to the application.

PassTicket functionality first appeared in RACF in the mid-1990's This functionality was updated in z/OSv1R7 with the addition of callable services and Java interfaces to those callable services, extending support of PassTicket operations to Java code running on z/OS.

A PassTicket is the result of a cryptographic algorithm applied to a user ID, an application name, a time stamp (current time) and a secret symmetric key. A PassTicket is an 8-character value consisting of EBCDIC A-Z, 0-9, and @ # $characters.

The two operations pertaining to PassTickets are generation (creation of a PassTicket using user ID, application name, and key) and evaluation (determining if a PassTicket is valid, given the user ID, application name, and key). The original password for the user is not used or required in either PassTicket operation.

PassTicket generation encodes the current time using the user ID, application name, and a secret key. During evaluation the user ID, application name, and secret key are used to decode the time stamp from the PassTicket. If this time stamp is within 10 minutes of the current time, it is a valid PassTicket.

The user ID is a z/OS user ID, with a length of 1-8 containing EBCDIC characters that are valid for RACF user IDs. The application name is also a 1 to 8 character EBCDIC string. The time stamp is automatically retrieved by the system at generation and evaluation time. Both the user ID and application name are case sensitive and should be presented in upper case to the PassTicket functions.

It is vital that the secret symmetric keys be protected wherever they are stored. If a key is compromised, anyone with knowledge of the key can generate working PassTickets for other users. When these keys are defined in RACF, there is the option to encrypt them using Integrated Cryptographic Service Facility (ICSF) as an additional layer of protection. This is described in *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683.

IBM only provides interfaces to generate and evaluate PassTickets on the z/OS platform. However, the algorithm to generate PassTickets is documented in the RACF publications with enough detail to allow its implementation in most computing languages and platforms. Any non-z/OS implementation of PassTicket operations must also deal with the protection of the secret key.

[1] There are exceptions to the single-use protection of Passtickets; refer to 6.8, "Miscellaneous PassTicket considerations" on page 88 for more information about this topic.

An additional function, the secured signon session key, is closely related to the PassTicket functionality. It gives z/OS applications the ability to generate an 8-byte key from an existing PassTicket, which can be used as a symmetric encryption key.

Due to the lack of Java interfaces to this function, this functionality is beyond the scope of this book and is not covered.

6.2 Documentation

Description and use of PassTickets is documented across multiple RACF publications. This can make it difficult to find the appropriate information at times.

In the RACF publications, PassTicket support is usually described as the Secured Signon Function. When searching the IBM publications for information about PassTicket, a search for Secured Signon may yield additional results.

Table 6-1summarizes the nature of the information on PassTicket given in the various RACF publications.

Table 6-1 PassTicket information in the RACF documentation

Publication	Topic
z/OS Security Server RACF Security Administrator's Guide, SA22-7683	PassTicket overview.
	Details about how to configure RACF to define and use PassTickets.
	PassTicket application profile definition information, including options for the protection of the secret key.
	Discussion of how to derive RACF profile names, including how to determine the application names of various z/OS subsystems and the user ID and group scoping functionality.
	Overview of how RACF processes PassTickets.
	Bypassing replay protection, when is it appropriate, and how to do it.
	Troubleshooting.
z/OS Security Server RACF Macros and Interfaces, SA22-7682	Describes various IBM supported mechanisms that can be used to generate PassTickets on z/OS.
	Details about the PassTicket generation algorithm which can be used to implement non-z/OS PassTicket generation code.
	This book contains the only mention of the Java PassTicket interfaces and a pointer to the location of the documentation which is found in the UNIX file system.
z/OS Security Server RACF System Programmer's Guide, SA22-7681	Another overview of PassTickets.

Publication	Topic
z/OS V1R9.0 Security Server RACF Callable Services, SA22-7691	Discussion about the R_gensec and R_ticketserv callable services which generate and evaluate PassTickets. This includes return codes from these functions.
z/OS V1R9.0 Security Server RACF Auditor's Guide	Documents audit records created by the R_ticketserv and R_gensec services when they perform PassTicket operations.
z/OS Security Server RACF Diagnosis Guide, GA22-7689	Details about using SAF Trace.

6.3 RACF configuration

To use PassTicket functionality from Java programs running on z/OS, RACF must be configured to activate the PassTicket function. This is accomplished with the SETROPTS command. The PTKTDATA class of profiles must be activated in RACLIST. To make it easier to define profiles which protect the PassTicket functions, set to allow generic profiles.

```
setropts classact(ptktdata)
setropts raclist(ptktdata)
setropts generic(ptktdata)
```

These settings may be verified with the SETROPTS LIST command.

6.3.1 RACF PassTicket Application Profile definition

For each application to be accessed using a PassTicket, a profile in the PTKTDATA class must be defined. The profile associates a secret PassTicket key with a particular application name on a particular system.

To define the application to RACF, create a profile using the RDEFINE command. After it is defined, it may be altered as needed with the RALTER command.

```
RDEFINE PTKTDATA profile-name SSIGNON(key-description) UACC(access-authority)
```

Where

- `PTKTDATA` specifies the PassTicket data class of profile.
- *`profile-name`* is the name of the application to be defined.
- *`key-description`* defines the secured signon application key and specifies the method RACF is to use to protect it in the RACF database on the host.
- `access-authority` is the universal access authority to be associated with the resource protected by this profile. There is no reason to specify anything other than NONE. Authorization to use the PassTicket services for this application is protected by other profiles.

6.3.2 Determining application name

One difficulty with the use of PassTickets is determining the correct application name to use. The application name is the link to the secret key, and is provided when the PassTicket is generated. However, when logging on, only the user ID and PassTicket is provided by the

client and it is up to z/OS to determine the application name, based on how the user connected to the system.

In some cases, it is easy to determine the application name. The PassTicket is used to log on to a custom z/OS application which directly specifies the application name in a call to RACROUTE REQUEST=VERIFY using the APPL= parameter. In this case, the generator and evaluator can decide in advance which application name to use and code the same application name during generation and evaluation.

Logging on to existing applications might be problematic because the user needs to determine which application name will be used by RACF during the PassTicket evaluation process. Some applications have simple constant names, but others require scrutiny to determine which name to use. *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683, discusses in detail how to select application names in the chapter “Using the Secured Signon Function.”

In addition to the applications that are documented in the RACF publication, OMVSAPPL is used, as an application name, by UNIX System Services applications that call _passwd() and _pthread_security_np(). In z/OS V1R10, the new UNIX System Services functions _passwd_appl() and _pthread_security_np_appl() have been created. They allow an application to be specified directly on the call, overriding the default OMVSAPPL.

WebSphere Application Server uses CBS390 as the default application name, but this can be overridden by using the security.zOS.domainName property.

6.4 PassTicket evaluation versus logging on using a PassTicket

The PassTicket Java services allow callers to both generate and evaluate PassTickets. Normally, a user logs on to an application using a RACF user ID and PassTicket. This log on request to the application drives a RACF function which evaluates the PassTicket, creates a security context, and associates the current unit of work with that user ID.

The PassTicket evaluation function is meant to be used to evaluate PassTicket for users who do not exist in the RACF database (for example, temporary or generated user IDs). However, it can also be used with RACF-defined users. There is no revocation of users because of failed PassTicket attempts, so the user must take great care in granting access to the PassTicket evaluation function. The evaluation service will only verify a PassTicket and not a password or password phrase.

The PassTicket evaluation service only evaluates that a PassTicket is computationally valid for a given user ID and application. It does not actually create any kind of z/OS security context for that user as a regular logon would do. Basically, it insures that the PassTicket originated from a system which has possession of the shared secret key.

6.4.1 Permission to use PassTicket services from Java

Additional profiles are created in RACF to define which users are allowed to perform PassTicket operations for a given application and target user ID. A target user ID is a user ID for whom a PassTicket is to be generated or evaluated.

These authorization profiles are all defined in the PTKTDATA class and can be defined using RDEFINE and changed using RALTER.

```
RDEFINE IRRPTAUTH.application.target-user UACC(NONE)
```

Where:

- *application* is the application name that this profile protects. Another profile in the PTKTDATA class must exist with this application name and contain a key in the SSIGNON segment.
- *target-user* is the user ID for whom a Passticket operation is to be performed.
- UACC(NONE) defines that, by default, no one may perform PassTicket operations.

Users with UPDATE access to a IRRPTAUTH profile are allowed to generate PassTickets for target-user for use with the application *application*. Generics are allowed in these profiles, so profiles that cover multiple users or multiple applications can be defined.

Users with READ access to a IRRPTAUTH profile are allowed to evaluate PassTickets for target-user for use with application *application*.

6.5 PassTicket Java

There are four files shipped with z/OS that implement the Java PassTicket support. These are all shipped in the UNIX file system. The Java support for PassTicket operations was shipped as part of RACF in z/OS V1R7. In z/OS V1R9, the code was moved to the SAF component of z/OS; however, no changes were made to any part names, or file locations. No changes to existing Java code need to be made.

- /usr/include/java_classes/IRRRacf.jar - This is the main jar file for the Java PassTicket support. This file must be included in java CLASSPATH when compiling and executing programs that use PassTicket services.
- /usr/include/java_classes/IRRRacfDoc.jar - Javadoc™ documentation for the PassTicket support. This file should be transferred to a workstation, un-jarred and viewed in a Web browser.
- /usr/lib/libIRRRacf.so - The Native DLL code used when a 31-bit JVM calls the Java PassTicket services. This file must be found in the UNIX $LIBPATH when running a program that uses Java PassTicket services.
- /usr/lib/libIRRRacf64.so - The Native DLL code used when a 64-bit JVM calls the Java PassTicket services. This file must be found in the UNIX $LIBPATH when running a program that uses Java PassTicket services.

6.5.1 Using Javadoc documentation

The Java PassTicket services are documented using standard Javadoc. The simplest way to view this data is to transfer it to a workstation, using FTP or equivalent. Be sure to use BINARY transfer mode when using FTP. After the IRRRacfDoc.jar file has been transferred, it can be expanded and viewed in a Web browser.

The simplest way to expand IRRRacfDoc.jar is to use the jar utility, as shown in Figure 6-1 on page 83.

```
>jar -xvf IRRRacfDoc.jar
  created: META-INF/
 inflated: META-INF/MANIFEST.MF
  created: doc/
 inflated: doc/allclasses-frame.html
 inflated: doc/allclasses-noframe.html
  created: doc/com/
  created: doc/com/ibm/
  created: doc/com/ibm/eserver/
  created: doc/com/ibm/eserver/zos/
  created: doc/com/ibm/eserver/zos/racf/
```

Figure 6-1 Using the jar utility to expand IRRRacfDoc.jar

If the jar utility is unavailable, the file can be processed as a zip file after renaming it to IRRRacfDoc.zip.

From here, the index.html file in the document directory is opened using a Web browser and the documentation is viewable.

6.5.2 PassTicket generation from Java on z/OS

After RACF is configured to support PassTicket, the simple Java program shown in Figure 6-2 on page 84 can be used to generate a PassTicket.

```
import com.ibm.eserver.zos.racf.IRRPassTicket;
import com.ibm.eserver.zos.racf.IRRPassTicketGenerationException;

public class ptGen {
    public static void main(String args[]) {
        IRRPassTicket x;
        // Sample user and appl info..
        String userid="IBMUSER";
        String appl="SAMPAPPL";

        // New instance of passticket class
        try {
            // Get new PT object
            x= new IRRPassTicket();
            System.out.println("User="+userid+" appl="+appl);
            // Generate new PassTicket.
            String pt= x.generate(userid,appl);
            System.out.println("New PassTicket: " + pt);
            // Catch failure.
        } catch (IRRPassTicketGenerationException bx) {
            System.out.println("Generation Exception caught: " + bx);
            bx.printStackTrace();
        }
    }
}
```

Figure 6-2 Program to generate a PassTicket

Compile

```
> javac -classpath /usr/include/java_classes/IRRRacf.jar ptGen.java
```

Execute

```
>java -classpath /usr/include/java_classes/IRRRacf.jar:. ptGen
User=IBMUSER appl=SAMPAPPL
New PassTicket: R4CHQH1V
```

In practice it would be expected that the generated PassTicket be used to log on to a remote server and not to be displayed on the screen.

Errors and exceptions

If an error occurs in the underlying PassTicket service, a Java exception is thrown. This exception contains the return and reason codes from the failing service.

```
>java -classpath /usr/include/java_classes/IRRRacf.jar:. ptGen
User=IBMUSER appl=BADAPPL
Generation Exception caught: SafRc=8, racfRc=8 racfRsn=16
SafRc=8, racfRc=8 racfRsn=16
        at
com.ibm.eserver.zos.racf.IRRPassTicket.generate(IRRPassTicket.java:226)
        at ptGen.main(ptGen.java:17)
```

These return codes are found in *z/OS Security Server RACF Callable Services*, SA22-7691. If a 31-bit JVM is used, the failing service is R_ticketserv (IRRSPK00). If a 64-bit JVM is used, the failing service is R_gensec (IRRSGS64).

PassTicket generation failures

Generally, if the PTKTDATA profile which defines the application and contains the key is present, along with the IRRPTAUTH.*application* profile, then PassTicket generation succeeds.

If neither profile is present, the first detected failure will be an authorization-related failure, due to RACF not finding the IRRPTAUTH profile.

Therefore, if a PassTicket generation failure is encountered, the user should check the relevant PTKTDATA profiles in RACF to solve the problem.

6.6 PassTicket evaluation from Java on z/OS

After RACF is configured to support PassTicket and a PassTicket is generated, the simple Java program shown in Figure 6-3 can be used to illustrate how to evaluate it. This program invokes the evaluation of the PassTicket and there is no creation of any sort of security context for the user.

```
import com.ibm.eserver.zos.racf.IRRPassTicket;
import com.ibm.eserver.zos.racf.IRRPassTicketEvaluationException;

public class ptEval {
    public static void main(String args[]) {
        IRRPassTicket x;
        // Sample user and appl info..
        String userid="IBMUSER";
        String appl="SAMPAPPL";
        try {
            // Get new PT object
            x= new IRRPassTicket();
            System.out.println("User="+userid+" appl="+appl+" pt="+args[0]);
            // Evaluate PassTicket.
            x.evaluate(userid,appl,args[0]);
            System.out.println("PassTicket is valid");
            // Catch failure.
        } catch (IRRPassTicketEvaluationException bx) {
            System.out.println("PassTicket evaluation failed." + bx);
        }
    }
}
```

Figure 6-3 PassTicket evaluation Java program

Compile

```
javac -classpath /usr/include/java_classes/IRRRacf.jar ptEval.java
```

Execute (use another program to first generate PassTicket)

```
>java -classpath /usr/include/java_classes/IRRRacf.jar:. ptGen
User=IBMUSER appl=SAMPAPPL
New PassTicket: 6IJX0888
>java -classpath /usr/include/java_classes/IRRRacf.jar:. ptEval 6IJX0888
User=IBMUSER appl=SAMPAPPL pt=6IJX0888
PassTicket is valid
…
```

If the PassTicket is evaluated a second time, the evaluation fails because the PassTicket may not be reused.

```
…
>java -classpath /usr/include/java_classes/IRRRacf.jar:. ptEval 6IJX0888
User=IBMUSER appl=SAMPAPPL pt=6IJX0888
PassTicket evaluation failed.SafRc=8, racfRc=16 racfRsn=32
```

PassTicket evaluation failures

PassTicket evaluation failures are more complex to diagnose than generation failures. Generation failures usually occur due to configuration problems in the PTKTDATA profiles. There are many causes of evaluation failures and, unfortunately, the return codes from the RACF services are not always descriptive.

Possible causes include:

- The PassTicket is just a bad PassTicket. It was corrupted or copied incorrectly.
- The PassTicket could be expired.
- The system clock on the generation and evaluation systems are out of sync by more than a few minutes. Clocks must be in sync with respect to GMT, not local time.
- The PassTicket has already been used.
- The application name is not defined on the evaluation system
- There is a key mismatch in the PTKTDATA profile for the application between the generation and evaluation systems.
- The user ID, application name, and PassTicket are all case sensitive. It is best to put everything into uppercase before sending it to both generate and evaluate.
- The value of a PassTicket is closely tied to the system clock with a one-second resolution. This means that, for any given user ID, key, application name, and time, there is only one possible PassTicket generated within a one-second interval.

  If multiple PassTickets are generated for the same user ID and application name within the same second interval, they will be all the same. After the first is evaluated, subsequent evaluations will fail due to PassTicket replay. Applications must either time themselves to insure that only one PassTicket is generated in any given second, or bypass the replay protection[2].

[2] The single use restriction of PassTickets can be bypassed. Refer to 6.8, “Miscellaneous PassTicket considerations” on page 88.

PassTicket logon problems

When a user logs on to an application with a PassTicket, there is an additional level of complexity due to the fact that it is RACF which determines the application name to use during evaluation. A few more things can go wrong.

- The user unexpectedly matched a user/group scoping application profile in the PTKTDATA class, and the key does not match the key on the generating system. [3]
- An ICHRIX01 user exit has changed the application name.
- RACF has picked an application name different from what was used in generation.

6.7 Audit

The Java PassTicket interfaces create audit records if audit settings are activated on PTKTDATA application profiles which contain the key. It is good practice to log the generation of PassTickets because, in effect, this action allows one user the ability to gain access as another user. Audit can be set as follows:

```
RALTER PTKTDATA OMVSAPPL AUDIT(SUCCESS(UPDATE))
```

Audit records show when a PassTicket is generated or evaluated, the user ID which performed the operation, the user ID for whom the PassTicket operation was performed, and the application name used. Figure 6-4 shows the XML format of SMF audit records, as provided by the RACF SMF data unload utility, that are relevant to PassTicket generation.

```
<event>
  <eventType>PTCREATE</eventType>
  <eventQual>SUCCESS</eventQual>
  <timeWritten>09:39:59.53</timeWritten>
  <dateWritten>2008-06-30</dateWritten>
…
  <details>
…
    <evtUserId>SRVRUSER</evtUserId>
    <evtGrpId>SYS1</evtGrpId>
…
    <term>093831A6</term>
    <jobName>SERVER7</jobName>
…
     <application>OMVSAPPL</application>
     <targetUser>TESTUSER</targetUser>
     <userName>USER NAME</userName>
…
   </details>
 </event>
```

Figure 6-4 XML output from SMF Unload showing an audit record generated when a PassTicket is generated

Note that these additional audit records are *not* created when standard z/OS logon processing is used to log on with a PassTicket. They are only created when a PassTicket is generated or evaluated.

[3] Refer to 6.8, “Miscellaneous PassTicket considerations” on page 88 for more information about scoping.

In addition, audit records may be cut when authorization to the PassTicket services is checked, depending on the audit setting set in the IRRPTAUTH profiles in the PTKTDATA class.

6.7.1 Tracing and debug

Both the Java services and the underlying RACF services have support in them for tracing which may be useful in debugging problems. Generally, the user is instructed by IBM support to activate the appropriate trace mechanism and send the report to IBM in the event a problem is encountered. Two types of tracing can be activated.

Java and Native code trace

Java and native code trace is used to trace the Java and native code which calls the RACF services. It is activated by adding a line to the trace options in the Java logging.properties file.

```
com.ibm.eserver.zos.racf.level = ALL
```

Trace records will be issued as configured in the logging.properties file.

SAF Trace

The Java native code calls SAF R_ticketserv or R_gensec callable services. These calls may need to be traced in order for IBM support to debug problems. These services are traced with the SAF Trace facility. R_ticketserv is service number 43. R_gensec is service number 50.

SAF Trace can also be used to help diagnose problems when PassTickets are used to logon to z/OS. Trace of the initACEE callable service (number 38) will trace some logon invocations.

Trace of the RACROUTE REQUEST=VERIFY and VERIFYX (RACROUTE call numbers 4 and 9) will trace call to the lowest level RACF logon service.

SAF Trace is documented in *z/OS Security Server RACF Diagnosis Guide*, GA22-7689. Refer to this document for more detailed information about setting up and using SAF Trace.

6.8 Miscellaneous PassTicket considerations

This section provides additional considerations of interest to PassTicket users experiencing practical problems because of the replay protection of PassTickets, or who wish to scope the use of PassTickets by user and group.

6.8.1 One-time use of PassTickets and bypassing replay protection setting

PassTickets are considered to have one-time use because used PassTickets are stored in system memory and compared against new PassTickets entering the system for evaluation. If a match is found, an otherwise valid PassTicket is rejected. However, because the log of used PassTickets is stored in memory, a PassTicket used on one system is still valid on a different system, even if the systems are members of the same sysplex.

Sometimes it is necessary to allow PassTickets to be used more than once. This bypassing of replay protection can be configured on an application by application (key) basis as explained in *z/OS Security Server RACF Security Administrator's Guide,* SA22-7683, in the chapter “Protecting General Resources.”

6.8.2 Scoping PassTicket Logon by user and group

It is possible to further scope application names and keys by RACF user ID and RACF group. This scoping allows for different keys to be used for the same application, based on the userID and connect group name.

Profiles supporting this scoping have one of the following forms:

- applicationName (no scoping)
- applicationName.groupName
- applicationName.userid
- applicationName.groupName.userid

The purpose of this is to scope the PassTicket generation ability of other systems which share the secret key. By allowing the scoping by user and group, a remote PassTicket generating system is only given the ability to generate PassTickets for a specific subset of users of a given application. An example of PassTicket generation scoping is shown in Figure 6-5.

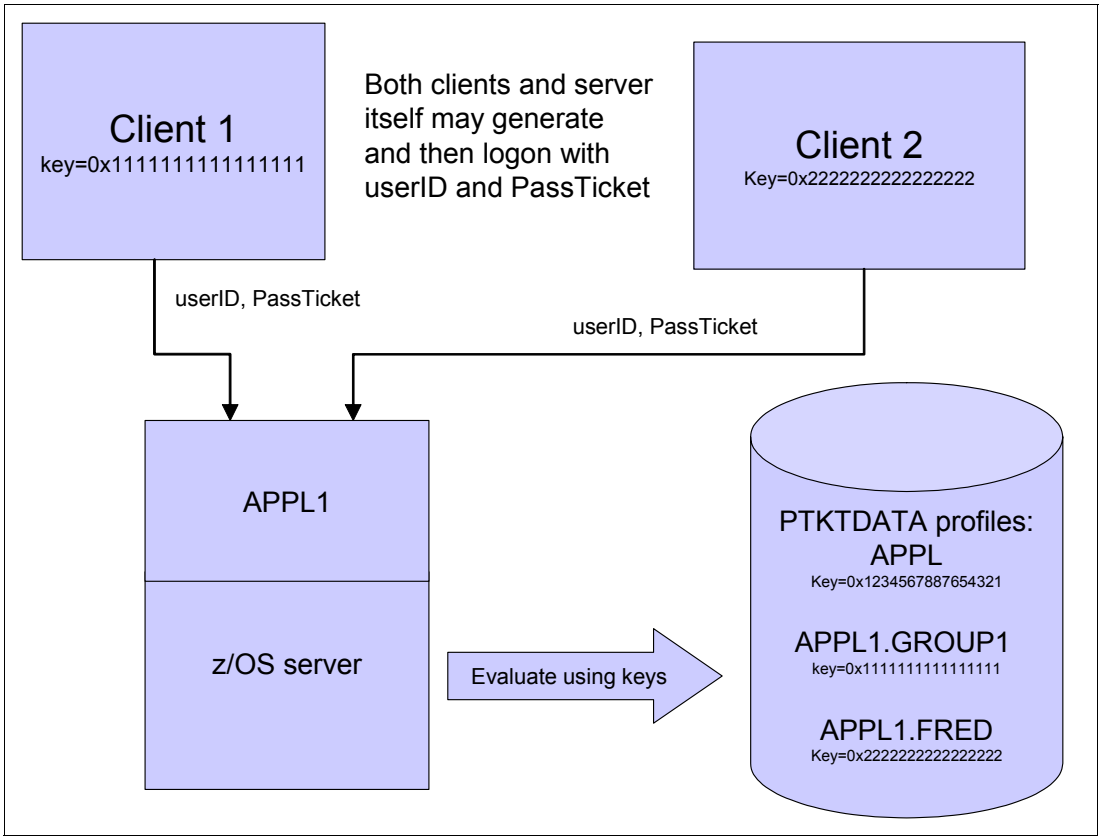


Figure 6-5 PassTicket scoping example

Figure 6-5 illustrates that Client 2 is only able to create a PassTicket for user FRED for APPL1 because it only has the key for APPL1.FRED. Client 1 may create a PassTicket for APPL1, but only for users connected to RACF group GROUP1. The server has access to all keys and may create PassTickets for all users to access APPL1.

Note that the client must take care in selecting the proper key for the proper user. For example, if FRED belongs to GROUP1, Client 1 is unable to generate a PassTicket for FRED because user FRED most closely matches the APPL1.FRED and this key will be used when

the PassTicket is evaluated. There is no “fallback” to less-specific profiles during the PassTicket evaluation process.

Even more confusing is that this profile matching mechanism is only used during the evaluation during logon of the PassTicket. The z/OS services that are used to generate a PassTicket only use the secret key stored in the 'applicationName' profile, and never the group or user qualified profiles. So, to make use of this feature, the key generation must take place on a different system from the one where the PassTicket is evaluated.

7

z/OS Enterprise Identity Mapping for Java applications

This chapter describes how to use Enterprise Identity Mapping (EIM) support via Java APIs. It provides a step-by-step example of EIM administration and mapping lookup, along with a demo Java application that utilizes EIM, Java Passtickets, and JSec. This section contains:

- A step-by-step example of EIM administration and lookup
- The description of a demo application
- The installation prerequisites and system setup information

For detailed information about EIM for z/OS, refer to *z/OS Integrated Security Services Enterprise Identity Mapping (EIM) Guide and Reference*, SA22-2875.

© Copyright IBM Corp. 2008. All rights reserved.

7.1 Enterprise Identity Mapping introduction

This section discusses the concept of Enterprise Identity Mapping in terms of:

- What problem it addresses
- Benefits of using EIM
- EIM implementation

7.1.1 The problem

Today's network environments consist of a complex group of systems and applications, resulting in the need to manage multiple user registries. These user registries are intended to be used by applications to achieve user identification and authentication.

The authenticated user ID is eventually used for access control as performed by the application itself, or the middleware it runs on, or by the local operating system. In today's typical heterogeneous configurations, this ends up in the situation shown in Figure 7-1.

In this figure, the many different platforms in the installation are represented with the local identity they have been set up with for the user John N Smith. For instance John N Smith is known as JohnSM in the AIX user registry, implying that John N Smith must be identified as JohnSM by applications running on his behalf on the AIX platforms. Also in this example, applications running on behalf of John N Smith on the z/OS platform must use the SAF user ID JOHN.

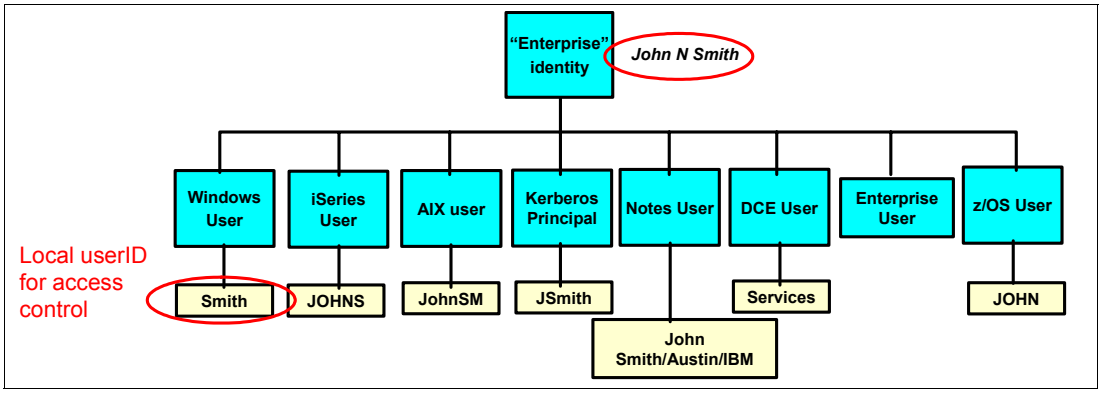


Figure 7-1 An heterogeneous configuration

Typically, these multiple user registries have a design and implementation specific to each type of platform or even application, because some of them may have been implemented at the application level. They have grown over time, along with the user population and the introduction of new systems and applications. This can result in significant administrative challenges that affect users, administrators, and application developers when it comes to keeping track of a single entity and its many representations in the installation.

In addition, system-specific local user IDs and registries, with their underlying mechanisms, are not going to unify in a common format across all vendors, at least for the foreseeable future. A user authenticates to an installation using a "network identity", which would be John N Smith in this example. It would then be left to the involved applications to map this network identity to the locally meaningful representation (user ID) of this user.

Enterprise Identity Mapping has been introduced as part of the IBM On Demand Initiative. It is an IBM system infrastructure technology that allows administrators and application developers to address this problem more easily and inexpensively than previously possible. By providing centralized and protected identification facilities, it also contributes to enabling products and applications to run in a more secure way in today's extremely open environments, as expected for the On Demand operating infrastructure.

EIM accomplishes this by providing a central place to store mappings between user IDs that are defined in different registries in an installation. These mappings indicate:

- A relationship between user IDs, that is, user IDs that belong to the same person or entity within the enterprise, or
- A transformation of one or more user IDs to an application-specific user ID

In EIM terminology, the unique name given at an enterprise level for a user or entity is the "EIM Identifier". EIM actually defines associations between an EIM identifier and user IDs in registries that are part of operating system platforms, applications, and middleware.

Applications, typically servers, can then use an EIM API to find a mapping that transforms the installation-level user ID initially used for authentication to a local user ID. The local user ID can, in turn, be used to access local resources.

Important: As the name implies, EIM is used to provide identity mapping only. EIM is not used to provide user authentication itself, which must be performed ahead of identity mapping. It is the application deployment strategy's responsibility to decide on how user authentication should be performed and how interapplication trust can be implemented. Kerberos is an authentication mechanism that may nicely fit the authentication needs in an EIM environment.

7.1.2 Benefits of using EIM

Using the EIM approach conveys the following benefits:

- The application server does not need to invent yet another user registry. Instead, it can use existing protocols for authenticating users and existing resource access managers to control the use of resources
- The application server does not need to store these mappings in side files.
- The mappings can be used by more than one application.
- The mappings can span different platforms in the enterprise.
- Multi-tier applications can be written that will not ask the user for a platform-specific identity. They can get this identity without going into an identification and authentication process that may expose passwords.

Note: EIM is often mistaken for a user ID management solution similar to products like the Tivoli Identity Manager. An identity management solution is able to work with all aspects of a user ID, including passwords and user authorities. EIM only contains the names of user IDs and none of the other attributes. It has insufficient information for authentication or authorization.

7.1.3 EIM implementation concepts

A simplified view of the EIM conceptual implementation is shown in Figure 7-2 on page 95. In this figure, the installation is composed of a z/OS, two i5/OS® (formerly OS/400®) and one AIX systems. The user ID mapping is kept in an LDAP directory called an “EIM Domain Controller” accessible by remote EIM clients.

In this example, John Smith (whose EIM identifier is “John N Smith”) is running a client application on his workstation. The application is eventually to send a request to a server running in SYSA. Both the client application and server are using Kerberos authentication.

1. John Smith has authenticated to the Kerberos authentication server (not shown in the picture) and now requests a service ticket to the Kerberos Ticket Granting Server to get access to the server on SYSA.
2. The Kerberos service ticket is delivered to the client application and contains John Smith’s Kerberos principal name that we assume for this example to be “John Smith@DomServer”.
3. John Smith’s client application requests a service on SYSA, using the Kerberos service ticket to authenticate. The Kerberos enabled server on SYSA decrypt and parses the service ticket and extracts John Smith principal name “John Smith”, which is expected to be meaningless to the local i5/OS user registry because it has not been implemented with Kerberos semantics. Note that the successful ticket parsing process is also a proof of proper client authentication by the Kerberos authentication server.
4. The server application on SYSA then acts as an EIM client, using the local EIM API, and requests the EIM domain controller to map the principal name “John Smith” from source “DomServer” to a user ID in the target “SYSA” registry.
5. The EIM domain controller looks up its directory and finds the mapping information for the specified source and target identities. In this example, the principal name “John Smith” returns JS50852 for John N Smith’s SYSA local identity.

   Note: The EIM domain Controller actually proceeds by first mapping the “source” identity, “John Smith” in this example, to the EIM identifier “John N Smith.” It then maps the EIM identifier to the corresponding user ID in the “target” registry, that is SYSA’s registry.

6. The server application on SYSA then acknowledges John Smith authentication and identity mapping to the client application. It is expected that the SYSA server application is now to run on behalf of user JS50852.

In this model, it is expected that if the request had to be passed on to another tier for additional processing, SYSA would pass it with the initially authenticated identity, that is “John Smith” from “DomServer” to the other system, which in turn would invoke the EIM Domain Controller for a local mapping.

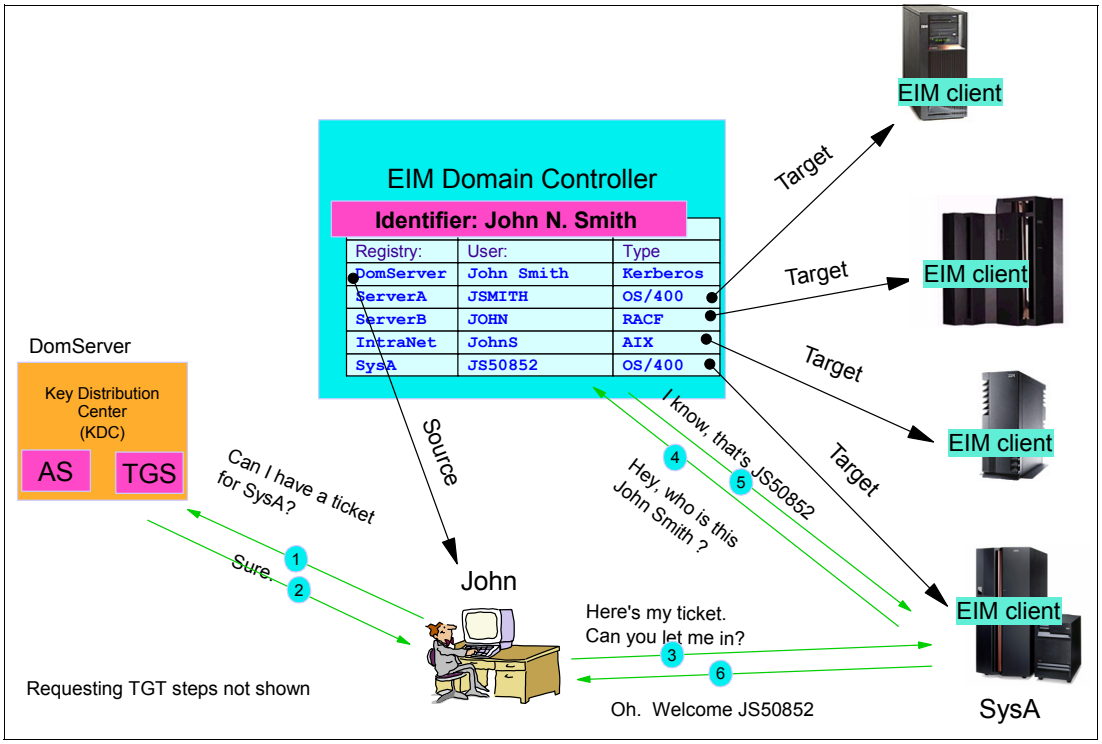


Figure 7-2 EIM conceptual implementation

7.1.4 EIM components

Following are the basic components of EIM:

EIM domain controller

This is an IBM Tivoli Directory server, for any platform, or IBM z/OS Integrated Security Services LDAP Server that contains one or more EIM domains.

The information in the domain controller are created and maintained with the eimadmin utility on z/OS. As of the time of writing, only the following eServer™ operating systems provide the eimadmin utility or equivalent administration tool:

- OS/400 and i5/OS
- z/OS
- AIX

EIM domain

This contains the mappings for an enterprise. It is located by the URL for the LDAP server. The name of the domain should be descriptive of the enterprise. An example of a EIM domain's URL is:

```
ldap://some.host/ibm-eimDomainName=My Business,c=fr
```

The EIM client

This is the code that implements the EIM lookup APIs and the administrative APIs that can be used for creating, modifying, displaying, and removing information from an EIM domain.

The EIM client uses the EIM API that is provided with the IBM system's operating systems or which are downloadable from the Internet.

The EIM client APIs are available for the following operating systems:

- OS/400, i5/OS
- z/OS
- AIX
- Windows®
- Linux

Note that the Windows and Linux clients have to be downloaded from:
`http://www-1.ibm.com/servers/eserver/security/eim/availability.html`

EIM identifier

This is the name that represents the unique name of an individual or entity within the enterprise. It can be a name or number or some combination of the two that represent a person or entity in the company. The EIM identifier is the anchor point for all mappings.

EIM registry

This is the logical representation of a user registry that exists on one of the systems in the network. Examples of user registries are RACF (or equivalent), LDAP (which contains bind IDs and passwords), Lotus® Notes®, and so on.

An EIM registry only contains user IDs and EIM information needed for mappings with EIM identifiers. It does not contain any of the other information, such as passwords or user attributes, that normally exist in a user registry.

EIM associations

These are the relationships between a use rID and an EIM identifier.

There are three kinds of associations:

source association - The registry user ID can be used as a source, that is a “starting point”, for a lookup operation. (It is assumed here that the source identity was properly authenticated by the application.)

target Association - The registry user ID can be returned as the target value for a lookup operation. It is expected that this user ID is intended to be used for access control by the application. The EIM domain controller also provides fields for application-specific information that can be used to refine the lookup results.

administrative association - the user ID is kept in the EIM Domain Controller for administration purpose only. However, it is not recognized as a source or target user ID in a mapping lookup. It is also not returned as a result of a mapping lookup.

User IDs can be defined with both source and target, and if desired, with administrative associations.

EIM policies

These are a special kind of association that can be used by an EIM lookup operation. If the EIM lookup API cannot locate a specific EIM association, the API will look for a policy.

The policy acts as a default or many-to-one mapping between a source registry and a target registry.

	There are three kinds of policies: ►A certificate filter policy associated with an x.509 registry ►A registry policy ►A domain policy
EIM lookup operations	These retrieve a mapping from the specified EIM domain. There are three kinds of lookups: **Get Target From Source** returns a user ID in the target registry when a source registry and source user ID is provided. **Get Target From Identifier** returns a user ID in the target registry when only the EIM identifier is provided. **Get Associated Identifier** returns the EIM identifier that has an association with the given registry and user ID.

Miscellaneous additional information

More details about EIM concepts can be found at the IBM Systems Information Center:

`http://www-1.ibm.com/servers/eserver/security/eim/`

An EIM domain controller can run on any platform where the IBM Tivoli Directory Server can be installed. To obtain a complete list of those platforms, go to the following Web site and search for IBM Tivoli Directory Server:

`http://www.ibm.com`

The z/OS Integrated Security Services LDAP Server can also serve as an EIM domain controller.

An application that uses EIM to locate mappings needs access to the EIM client. Depending upon the platform, it is either part of the operating system or available from IBM as a download.

There are two kinds of EIM clients; one is written in C/C++ and the second is written in Java. The IBM Systems Information center has the latest information about supported platforms and releases.

7.2 EIM Domain Controller overview

The EIM Controller uses an LDAP directory server that supports the LDAP V3 protocol.

The following objects are represented by entries in the Domain Controller:

- Domain
- Identifiers
- Registries
- Associations
- Policies

7.2.1 EIM LDAP Directory Tree

Figure 7-3 on page 98 shows an example of a directory tree for the myJavaDomain domain.

The subtrees to the EIM domain entry contain:

- EIM identifiers - A source user ID points to an EIM identifier. Therefore, all existing source associations are represented at the EIM identifier level.
- registries - These are the representations of the IBM identifier local to a system of application. The lookup target associations are represented at the registry level.
- groups - These are EIM access control groups.

These directory entries can be built and maintained via the Java APIs. Users should not try to manage them directly using their own LDAP operations.

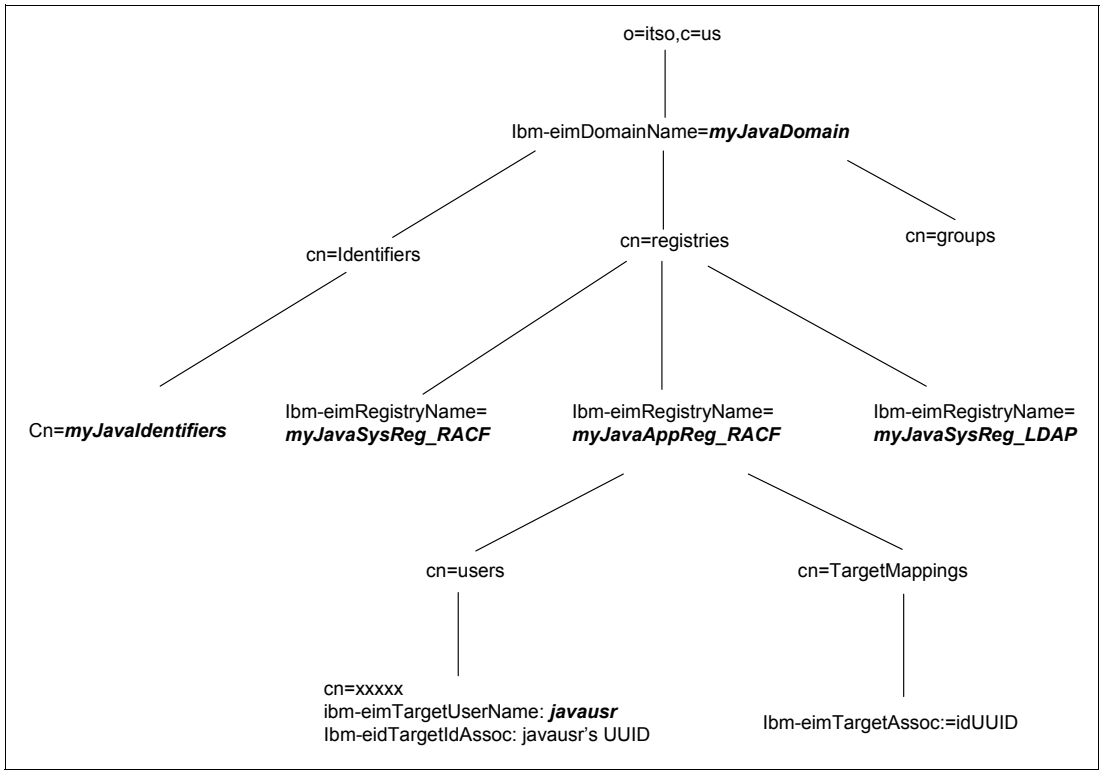


Figure 7-3 EIM Domain Controller Directory Information Tree

7.2.2 Access control to the EIM Domain Controller

The access control to the EIM Domain Controller and its contents is based on the accessing user's distinguished name and its proper authentication.

User authentication on z/OS LDAP

A user accessing an EIM Domain Controller running on z/OS can authenticate with:

- The LDAP simple bind - the bind distinguished name and the bind password flow over the TCP/IP session, in clear text or protected by SSL encryption.
- Alternatively, the simple bind can use a password protected by the CRAM-MD5 hashing.
- A Kerberos service ticket, with the Kerberos principal name of the client. The Kerberos protocol itself ensures proper authentication of the client.

- A client digital certificate, over an SSL connection, that contains the client distinguished name. The SSL protocol itself ensures proper authentication of the client.

Note: Much information, such as the LDAP bind and authentication data and the domain to be used can be entered, for security and information management purposes, in RACF profiles used when connecting to the EIM Domain Controller. For more information about this topic, refer to 7.4, “Writing EIM Java applications” on page 104.

Access control to EIM objects

The LDAP objects in the EIM Domain Controller are subject to access controls via LDAP ACLs that grant access to the authenticated user according to the EIM access control group that the user is a member of. The predefined access control groups are established at the EIM Domain level. At the time of writing, they are:

- EIM Administrator - This access control group allows the user to manage all of the EIM data within this EIM domain.
- EIM Registries administrator or registry x administrator - This access control group allows the user to manage all EIM registries, or only one specific registry, definitions, and target associations.
- EIM Identifier administrator - This access control group allows the user to add and change EIM identifiers and manage source and administrative associations.
- EIM Mapping Operations- This access control group allows the user to conduct EIM lookup operations. A user with this access control can perform the following functions:
  - Perform EIM lookup operations
  - Retrieve associations, EIM identifiers, and EIM registry definitions

The LDAP administrator has full access to any object in the directory, and only an LDAP administrator can create a new domain.

Users are assigned to access control groups by the addAdminAccessUser method.
A user invoking the EIM API is authorized to specific functions as shown in Table 7-1.

Table 7-1 EIM access controls

API class	EIM Admin	EIM Registries Admin *	EIM Identifier Admin	EIM Mapping Operations
Domain	Delete, change, list	None	None	None
Registry, Registry Alias	Add, remove, change, list	Change, list	List	List
Registry User	Change, list	Change, list	List	List
Identifier	Add, remove, change, list	List	Add, change, list	List
Association	Add, remove, list all types (admin, source, target)	(target) add, remove; (all) list	(admin, source) add, remove; (all) list	List all
Access	Add, list, remove, query	None	None	None
Mapping Lookups	All types	All types	All types	All types

API class	EIM Admin	EIM Registries Admin *	EIM Identifier Admin	EIM Mapping Operations
Policy Associations **	Add, list, remove	Add, list, remove	List	Add, list, remove
Policy Filters **	Add, list, remove	List	List	List

* There is also a registry admin group for each registry.

** Available with z/OS 1.6.

7.2.3 LDAP setup

The EIM Domain Controller can take advantage of the usual LDAP configurations aiming at enhancing availability or at simplifying the name space management. These are:

- LDAP server with replica(s) - The replica is another usable instance of the master directory. The master-replica configuration can be implemented in z/OS, beginning with z/OS V1R6 a peer-to-peer replication configuration can also be used.
- LDAP directory with referrals - A referral is a pointer in a directory towards another directory, so that the naming space can split across several LDAP servers. LDAP V3 referrals can be used with the z/OS LDAP server.
- To achieve very high availability, z/OS also allows you to configure a single directory to be served by multiple LDAP servers located in members of a sysplex.

Note: LDAP servers candidates to host an EIM domain controller should support the following attributes:

- ibm-entryUUID
- aclEntry
- aclPropagate
- aclSource
- entryOwner
- entryPropagate
- entrySource

EIM schemas

In z/OS, the EIM LDAP object classes and attributes definitions are delivered in the schema.IBM.ldif file in /usr/lpp/ldap/etc/. At the time of writing, they are:

Objectclasses

- ibm-eimDomain
- ibm-eimIdentifier
- ibm-eimRegistry
- ibm-eimSystemRegistry
- ibm-eimApplicationRegistry
- ibm-eimRegistryUser
- ibm-eimSourceRelationship
- ibm-eimTargetRelationship
- ibm-eimDefaultPolicy
- ibm-eimDomainName
- ibm-eimFilterPolicy
- ibm-eimPolicyListAux

Attributes

- ibm-eimDomainName
- ibm-eimAdditionalInformation
- ibm-eimAdminUserAssoc
- ibm-eimDomainVersion
- ibm-eimRegistryAliases
- ibm-eimRegistryEntryName
- ibm-eimRegistryName
- ibm-eimRegistryType
- ibm-eimSourceUserAssoc
- ibm-eimTargetIdAssoc
- ibm-eimTargetUserName
- ibm-eimUserAssoc
- ibm-eimFilterType
- ibm-eimFilterValue
- ibm-eimPolicyStatus

7.2.4 RACF profiles to keep EIM default parameters for LDAP bind information

Several profiles can be used to store in RACF an LDAP URL, the bind distinguished name, and the bind password, which can then be retrieved by the EIM services to bind to the EIM Domain Controller. The choice of the profile is dictated by the intent to either establish a system default that is used for EIM only, or that can be used by other functions (such as the z/OS PKI Services), or to establish a default IRR.EIM.DEFAULTS profile in the LDAPBIND class or IRR.PROXY.DEFAULTS profile in the FACILITY class.

These profiles are searched in the following order, as shown in Figure 7-4 on page 102, by EIM services which have been invoked without specifying the Domain Controller or Domain in their parameter list:

1. Application user ID specific default

   The user-specific default is in a profile in the LDAPBIND class, with an arbitrary name that should also appear in the EIM segment of the USER profile.

   An example of definition of this profile is:

   ```
   RDEFINE LDAPBIND BUCKSDOMAIN +
   EIM(DOMAINDN('ibm-eimDomain=Bucks Domain.o=ibm.c=us')) +
   OPTIONS(ENABLE)) +
   PROXY(LDAPHOST(ldap://another.big.host) +
   BINDDN('cn=EIM Application Lookups') BINDPW('secret'))
   ```

   To establish this profile as a default for a the specific user SERVERID, the EIM segment in the USER profile can be updated as follows:

   ```
   ALTUSER SERVERID EIM(LDAPPROF(BUCKSDOMAIN))
   ```

2. System-wide default

   The IRR.EIM.DEFAULTS profile in the LDAPBIND class establishes a default. An example of definition of this profile is:

   ```
   RDEFINE LDAPBIND IRR.EIM.DEFAULTS +
   EIM(DOMAINDN('ibm-eimDomain=Joes Domain.o=ibm.c=us')) +
   OPTIONS(ENABLE)) +
   PROXY(LDAPHOST(ldap://some.big.host) +
   BINDDN('cn=EIM Lookup') BINDPW('secret'))T
   ```

3. System-wide default

   IRR.PROXY.DEFAULTS profile in the FACILITY class. An example of definition of this profile is:

```
RDEFINE FACILITY IRR.PROXY.DEFAULTS +
EIM(DOMAINDN('ibm-eimDomain=Joes Domain.o=ibm.c=us')) +
OPTIONS(ENABLE)) +
PROXY(LDAPHOST(ldap://some.big.host) +
BINDDN('cn=EIM Lookup') BINDPW('secret'))
```

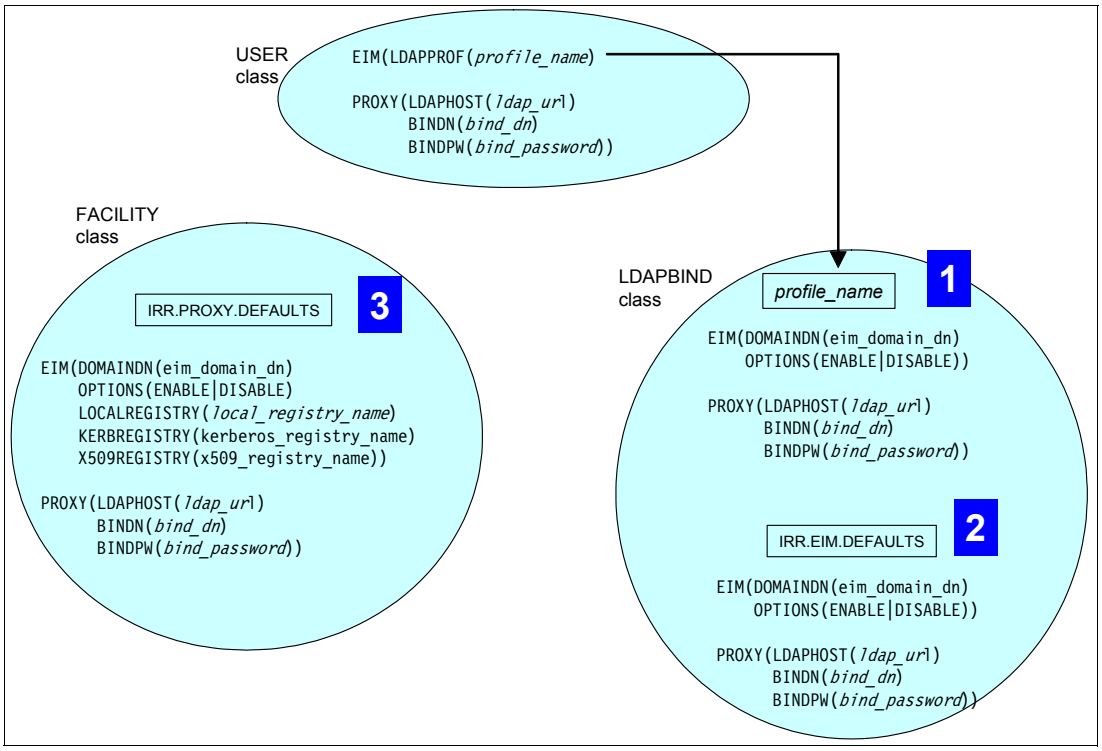


Figure 7-4 Keeping EIM LDAP and domain information in RACF profiles

The EIM and PROXY segment keywords and options combine to define the EIM domain, the LDAP host it resides on, and the bind information required by the EIM services to establish a connection with an EIM domain. The EIM services will attempt to retrieve this information when it is not explicitly supplied via invocation parameters.

7.2.5 RACF profiles to keep EIM default parameters for the local registry name

The assumption here is that the registry name is also specified in the EIM Domain controller, presumably by using the administrative API, or the eimadmin utility.

The local registry name can be set in the FACILITY IRR.PROXY.DEFAULTS profile by the RACF administrator for retrieval by the following methods, when the local registry is not indicated in the parameter list:

- findTarget
- findTargetFromSource
- getAssociations
- getAssociatedEids

The local registry name can be set with the following commands:

```
RALTER FACILITY IRR.PROXY.DEFAULTS EIM(LOCALREGISTRY('registry_name'))
```

A local registry name can be removed from storage with the following commands:

```
RALTER FACILITY IRR.PROXY.DEFAULTS EIM(NOLOCALREGISTRY)
```

7.3 z/OS EIM Java API

The z/OS EIM Java API actually wraps the EIM C/C++ API, as shown in Figure 7-5. It may be used by applications that require a lookup of the EIM mapping information, or application designed for administering this information.

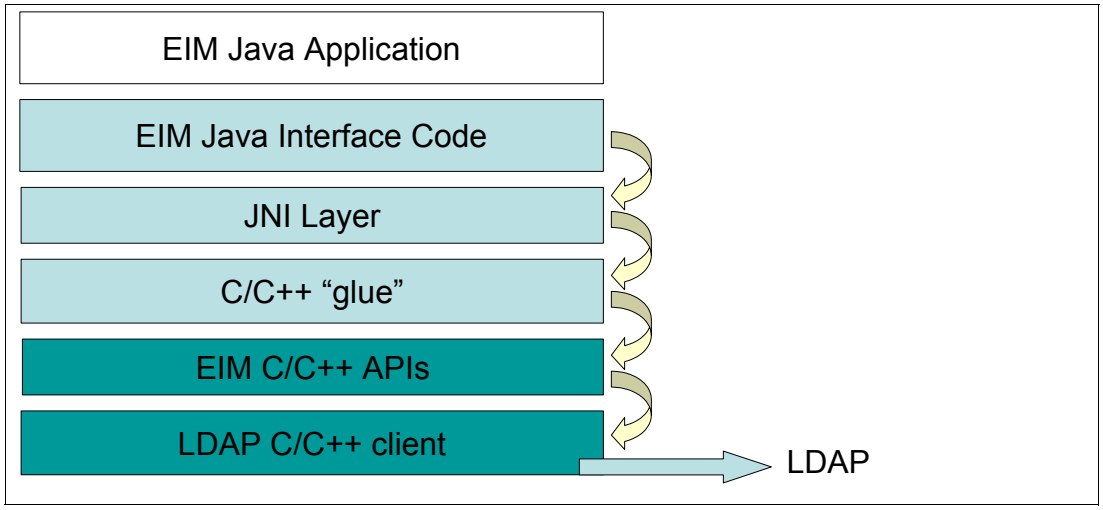


Figure 7-5 The z/OS EIM Java API

The documentation is available in the z/OS HFS at /usr/lpp/eim/lib/eimzOS_DOC.jar. Like any jar file, it contains ASCII data and should be transferred as a binary image to a workstation for unjarring and then browsing the HTML pages.

7.3.1 Configuring the EIM Java API

This section explains what setups are to be performed in z/OS to make the EIM Java API usable.

Java classpath

The following two jar files must be in the user classpath:

- /usr/lpp/eim/lib/eim.jar
- /usr/lpp/eim/lib/eimzOS.jar

RACF authorizations for using the EIM API

This section lists the RACF profiles that protect access to EIM functions.

Domain retrieval requirements

- READ access to the IRR.RGETINFO.EIM profile in the FACILITY class
- READ access to the IRR.RDCEKEY profile in the FACILITY class

Mapping lookup requirement

- READ access to the IRR.RGETINFO.EIM profile in the FACILITY class

EIM auditing requirement

- READ access to the IRR.RAUDITX profile in the FACILITY class

Configuration via the ConfigurationManager class requirements

- READ access to the IRR.RADMIN profile in the FACILITY class
- READ access to the IRR.RGETINFO.EIM profile in the FACILITY class

RACF setup for a Java application running with the identity TSOUSER

The FACILITY class of resources should be made active in RACF and RACLIST'ed with the following commands:

```
SETR CLASSACT(FACILITY)
SETR RACLIST(FACILITY)
```

The proper profiles have to be defined in the FACILITY class:

```
RDEFINE FACILITY IRR.RGETINFO.EIM UACC(NONE)
RDEFINE FACILITY IRR.RDCEKEY UACC(NONE)
RDEFINE FACILITY IRR.RAUDITX UACC(NONE)
RDEFINE FACILITY IRR.RADMIN UACC(NONE)
```

The user TSOUSER must have permission to the profiles:

```
PERMIT IRR.RGETINFO.EIM CLASS(FACILITY) ID(TSOUSER) ACCESS(READ)
PERMIT IRR.RDCEKEY CLASS(FACILITY) ID(TSOUSER) ACCESS(READ)
PERMIT IRR.RAUDITX CLASS(FACILITY) ID(TSOUSER) ACCESS(READ)
PERMIT IRR.RADMIN CLASS(FACILITY) ID(TSOUSER) ACCESS(READ)
```

Finally, the RACLIST'ed profiles should be refreshed to take into account the new definitions in the FACILITY class:

```
SETR RACLIST(FACILITY) REFRESH
```

7.4 Writing EIM Java applications

The EIM Java API is an extensive set of interfaces used to build applications in Java that perform EIM administration or request mapping information. The following sections build on each other to create EIM domains, registries, identifiers, and associations to perform an EIM lookup operation. Note that each sample application adds to the previous sample, and that the last example is a fictitious application intended to be a synthesis of all the elementary functions previously discussed.

7.4.1 Basic EIM administration

This section explains how to set up an EIM environment using the EIM Java administrative API.

Step 1: Specifying the EIM connection

The ConnectInfo class provides methods that specify how to connect to LDAP. A Java application can provide a bind distinguished name, bind password, SSL information and protection information to the constructors. The constructed object will later be passed as a parameter to the DomainManager to establish a connection to an EIM domain. Figure 7-6 shows how a ConectInfo object can be created.

```
// Define the LDAP distinguished name and LDAP password for the connection
String ldapDN = "cn=ldap_administrator";
String ldapPW = "secret";

// Specify the connection information
ConnectInfo connectInfo = new ConnectInfo(ldapDN, ldapPW);
```

Figure 7-6 Creating a ConnectInfo object

Step 2: Setting up the EIM Domain

An EIM Domain contains the mappings for an enterprise. The DomainManager class establishes a connection to a particular EIM domain in LDAP. The ConnectInfo object and an LDAP URL is required for the connection. The LDAP URL uses the following format

```
ldap://eimsystem:389/ibm-eimDomainName=myEimDomain,o=mycompany,c=us"
```

The URL contains the server address, EIM domain name, and the root for that domain. The port number defaults to 389 if not specified.

The user must get an instance of the DomainManager in order to work with EIM domains. The DomainManager can create new domains or retrieve existing domains. Figure 7-7 on page 106 shows how to create an EIM Domain.

```
// Specify the connection information
ConnectInfo connectInfo = new ConnectInfo("cn=ldap_administrator",
"secret");

// Get an instance of the domain manager
DomainManager domainMgr = DomainManager.getInstance();

// Build the ldap url
String ldapServer = "9.12.4.18";
String domainName = "myJavaDomain";
String baseDn = "o=itso,c=us";
String ldapUrl = "ldap://" + ldapServer + "/ibm-eimdomainname=" + domainName
+ "," + baseDn;

// Create the domain
Domain myJavaDomain = domainMgr.createDomain(ldapUrl, connectInfo,
"description");
```

Figure 7-7 Creating an EIM Domain

Existing domains may be retrieved to perform lookups or administration. The user can retrieve a single domain by specifying the domain name, or can retrieve several domains by specifying only the server address and root. Figure 7-8 shows how to retrieve a single EIM domain.

```
// Specify the connection information
ConnectInfo connectInfo = new ConnectInfo("cn=ldap_administrator",
"secret");

// Get an instance of the domain manager
DomainManager domainMgr = DomainManager.getInstance();

// Retrieve a single domain
String ldapUrl =
"ldap://9.12.4.18/ibm-eimdomainname=myJavaDomain,o=itso,c=us";
Domain myJavaDomain = domainMgr.getDomain(ldapUrl, connectInfo);

// Display the domain
System.out.println("domain: " + myJavaDomain.getName());
```

Figure 7-8 Retrieving a single Domain

Figure 7-9 on page 107 shows how to retrieve multiple domains.

```
// Specify the connection information
ConnectInfo connectInfo = new ConnectInfo("cn=ldap_administrator",
"secret");

// Get an instance of the domain manager
DomainManager domainMgr = DomainManager.getInstance();

// Retrieve all domains (notice that no domain is specified in the
ldap url)
String ldapUrl = "ldap://9.12.4.18,o=itso,c=us";
Set myJavaDomains = domainMgr.getDomains(ldapUrl, connectInfo);

// Display the domains
Iterator iter = myJavaDomains.iterator();
while(iter.hasNext())
System.out.println("domain: " + iter.next());
```

Figure 7-9 Retrieving multiple domains

Note again that the LDAP URL is optional for creating and retrieving the default domain. Instead, a null value can be passed, which results in the EIM Defaults being used. Refer to 7.4, “Writing EIM Java applications” on page 104 for more information about this topic.

Figure 7-10 shows the retrieval of a default domain. The relevant RACF profile definition is as follows:

```
RALTER FACILITY IRR.PROXY.DEFAULTS +
EIM(DOMAINDN('ibm-eimDomainname=myJavaDomain.o=itso.c=us')) + OPTIONS(ENABLE)) +
PROXY(LDAPHOST(ldap://some.big.host) + BINDDN('cn=ldap_administrator,o=itso,c=us')
BINDPW('secret'))
```

```
// Specify the connection information
ConnectInfo connectInfo = new ConnectInfo("cn=ldap_administrator", "secret");

// Get an instance of the domain manager
DomainManager domainMgr = DomainManager.getInstance();

// Retrieve the default domain (notice that null is specified in the ldap url)
Set myJavaDomains = domainMgr.getDomains(null, connectInfo);


// Display the domain
System.out.println("domain: " + myJavaDomain.getName());
```

Figure 7-10 Using the domain definitions in RACF profile

Although the DomainManager class can create and retrieve domains, the Domain class provides both administrative and lookup functionality.

Users can perform the following administrative functions:

- Add system and application registries, policy associations, identifiers
- List associations, identifiers, registries
- Remove policy associations
- Delete the domain
- Disconnect or reconnect the domain

Users can also perform the following lookup function:

- Get target users from source

Domain usage samples are provided in steps 3 to 7.

Step 3: Adding Registries

The Registry class defines behavior for system and application registries in an EIM domain. A system registry is a logical representation of a user registry that exists on one of the systems in a network. An application registry represents a user registry particular to an application on a system. As such, an application registry is associated with a system registry.

Examples of user registries are RACF (or equivalent), LDAP (which contains bind IDs and passwords), Lotus Notes, and so on. An EIM registry only contains user IDs and EIM information needed for mappings with EIM identifiers. It does not contain any of the other information, such as passwords or user attributes, that normally exist in a user registry.

Registries are created in the Domain class by calling either the addSystemRegistry or addApplicationRegistry method. A Registry object is returned that can use the following registry functionality.

- Add aliases and certificate policies
- Delete the registry
- List aliases, policies, users
- Remove aliases, users, policies

Registry objects must specify a registry type. The Registry class provides several constants that can be used for the registry type.

The provided registry types are:

- EIM_REGTYPE_AIX
- EIM_REGTYPE_KERBEROS_EX
- EIM_REGTYPE_KERBEROS_IG
- EIM_REGTYPE_LDAP
- EIM_REGTYPE_LINUX
- EIM_REGTYPE_NDS
- EIM_REGTYPE_OS400
- EIM_REGTYPE_POLICY_DIRECTOR
- EIM_REGTYPE_RACF
- EIM_REGTYPE_TIVOLI_ACCESS_MANAGER
- EIM_REGTYPE_WIN2K
- EIM_REGTYPE_X509

The Registry object that is returned on the call to addSystemRegistry or addApplicationRegistry can be cast to either a SystemRegistry or an ApplicationRegistry object. This is referred as the kind of registry which will always be either EIM_APPLICATION_REGISTRY or EIM_SYSTEM_REGISTRY.

Steps 4 and 5 shows how to add users to these registries by creating identifiers and associations. Figure 7-11 shows how to add registries.

```
// Add a system registry
Registry ldapSysReg = myJavaDomain.addSystemRegistry("myJavaSysReg_LDAP",
Registry.EIM_REGTYPE_LDAP, "description", "uri");

// Add an application registry
Registry racfSysReg = myJavaDomain.addSystemRegistry("myJavaSysReg_RACF",
Registry.EIM_REGTYPE_RACF, "description", "uri");
Registry racfAppReg =
myJavaDomain.addApplicationRegistry("myJavaAppReg_RACF",
Registry.EIM_REGTYPE_RACF, "description", racfSysReg.getName());

// List the registries
Set regs = myJavaDomain.getRegistries();

// Display the registries
Iterator iter = regs.iterator();
while(iter.hasNext()){
   Registry reg = ((Registry) iter.next());
   System.out.println("registry: " + reg.getName() + " kind: " +
reg.getKind());
}
```

Figure 7-11 Adding registries

Step 4: Adding identifiers

An *identifier* is the name that represents the unique name of an individual or entity within the enterprise. It can be a name or number or some combination of the two that represents a person or entity in your company. The EIM identifier is the anchor point for all mappings.

The Eid object represents an EIM identifier. Identifiers are created in the Domain class by calling the addEid method with the name and description. An Eid object is returned that can be used to associate a user with a registry in the domain. Additionally, an EID provides functionality to:

- Add aliases and associations
- Delete the identifier
- Find targets associated with the EID
- List aliases and associations
- Remove aliases and associations

Figure 7-12 on page 110 shows how to add an identifier.

```
// Add a user identifier
Eid identifier = myJavaDomain.addEid("myJavaIdentifier", "description");

// List the identifiers
Set identifiers = myJavaDomain.getEids();

// Display the identifiers
Iterator iter = identifiers.iterator();
while(iter.hasNext()){
   Eid eid = ((Eid) iter.next());
   System.out.println("identifier: " + eid.getName());
}
```

Figure 7-12 Adding an identifier

Step 5: Adding associations

An *association* is a relationship between an EIM identifier and an EIM Registry and EIM registry user. There are three kinds of associations: source associations, target associations, and administrative associations, as explained here.

- Source association - The registry user ID can be used as a source, that is, a starting point, for a lookup operation. It is assumed here that the source identity has been properly authenticated by the application.
- Target Association - The registry user ID can be returned as the target value for a lookup operation. It is expected that this user ID is intended to be used for access control by the application.

The EIM domain controller also provides fields for application-specific information that can be used to refine the lookup results.

- Administrative association - The user ID is kept in the EIM Domain Controller for administration purposes only. However, it is not recognized as a source or target userID in a mapping lookup. Also, it is not returned as a result of a mapping lookup.

User IDs can be defined with both source and target and, if desired, with administrative associations.

Associations must indicate an association type when added to an identifier.

The provided association constants include:

- EIM_ADMIN
- EIM_ALL_ASSOC
- EIM_ALL_POLICY_ASSOC
- EIM_CERT_FILTER_POLICY
- EIM_DEFAULT_DOMAIN_POLICY
- EIM_DEFAULT_REG_POLICY
- EIM_SOURCE
- EIM_SOURCE_AND_TARGET
- EIM_TARGET

Figure 7-13 on page 111 shows how to add associations.

```
// Associate the identifier with the users in various registries
identifier.addAssociation(Association.EIM_SOURCE, ldapSysReg.getName(),
"javauser@us.ibm.com");
identifier.addAssociation(Association.EIM_TARGET, racfAppReg.getName(),
"javausr");

// List associations
Set srcAssoc = identifier.getAssociations(Association.EIM_SOURCE);
Set tgtAssoc = identifier.getAssociations(Association.EIM_TARGET);

// Display associations
Iterator iter = srcAssoc.iterator();
while(iter.hasNext()){
   Association assoc = ((Association) iter.next());
   System.out.println("association: " + assoc.getUid());
}
iter = tgtAssoc.iterator();
while(iter.hasNext()){
   Association assoc = ((Association) iter.next());
   System.out.println("association: " + assoc.getUid());
}
```

Figure 7-13 Adding an association

Step 6: Listing registry users

An EIM registry user is defined as the target user of an EIM lookup operation. The RegistryUser class represents registry users. A RegistryUser cannot be instantiated, but is returned as a result of a lookup operation. Figure 7-14 shows how to list registry users.

```
   // List target users
   Set ldapUsers = ldapSysReg.getUsers();  // No target users
   Set racfUsers = racfAppReg.getUsers();  // One target user

   // Display target users
   Iterator iter = ldapUsers.iterator();
   while(iter.hasNext()){
      RegistryUser regUser = ((RegistryUser) iter.next());
      System.out.println("registry=" + regUser.getRegistryName() + " user=" +
   regUser.getTargetUserName());
   }
   iter = racfUsers.iterator();
   while(iter.hasNext()){
      RegistryUser regUser = ((RegistryUser) iter.next());
      System.out.println("registry=" + regUser.getRegistryName() + " user=" +
   regUser.getTargetUserName());
   }
```

Figure 7-14 Listing registry users

Deleting an EIM domain

To delete an EIM domain, components under the domain must be deleted first. Then the domain itself can be deleted. Figure 7-15 shows how to delete an EIM domain.

```
// Delete all registries
Set regs = myJavaDomain.getRegistries();
Iterator regIter = regs.iterator();
while(regIter.hasNext())
   ((Registry) (regIter.next())).delete();

// Delete all identifiers
Set eids = myJavaDomain.getEids();
Iterator eidIter = eids.iterator();
while(eidIter.hasNext())
    ((Eid) (eidIter.next())).delete();

// Delete domain after identifiers and registries are deleted
myJavaDomain.delete();
```

Figure 7-15 Deleting an EIM domain

7.4.2 EIM runtime lookups

After the EIM environment has been set up as previously described, the EIM-enabled applications can then perform lookup operations as shown in this section.

Step 7: Performing EIM lookup operations

A lookup operation uses identifier associations to map a user in one registry to another. It accepts a source user and source registry, along with a target registry, to identify the target user.

The Domain class provides interfaces for performing EIM lookups. The findTargetFromSource methods provide functionality to lookup users. Figure 7-16 shows examples of lookup operations.

```
// Use EIM to map the off-platform user id to a RACF user id
Set regUsers = myJavaDomain.findTargetFromSource("javauser@us.ibm.com",
"myJavaSysReg_LDAP", "myJavaAppReg_RACF");

// List the user
RegistryUser regUser = ((RegistryUser) regUsers.iterator().next());
String mappedUser = regUser.getTargetUserName();

// Display the user
System.out.println("mappedUser: " + mappedUser);
```

Figure 7-16 Looking up for user identity

Using the target registry is optional for retrieving the local registry. Instead, a null value can be passed in which results in the EIM Defaults be used. For more information about this topic,

refer to 7.2.5, “RACF profiles to keep EIM default parameters for the local registry name” on page 102.

If the defaults are defined using the following RACF commands:

```
RALTER FACILITY IRR.PROXY.DEFAULTS EIM(LOCALREGISTRY('myJavaSysReg_RACF'))

SETROPTS EIMREGISTRY
```

then the lookup operation can be invoked as shown in Figure 7-17.

```
// Use EIM to map the off-platform user id to a RACF user id
// Note that the target registry is null
Set regUsers = myJavaDomain.findTargetFromSource("javauser@us.ibm.com",
"myJavaSysReg_LDAP", null);

// List the user
RegistryUser regUser = ((RegistryUser) regUsers.iterator().next());
String mappedUser = regUser.getTargetUserName();

// Display the user
System.out.println("mappedUser: " + mappedUser);
```

Figure 7-17 Lookup operations using the default definitions

7.5 A demonstration of EIM and other z/OS Java APIs

This section provides an example of a fictitious application that demonstrates the use of several Java APIs described in this book:

- The EIM API
- The PassTicket API
- The JSec API

7.5.1 The eimjavademo application

This Java application performs a trusted mapping of a non-z/OS identity to a z/OS user ID that is eventually used to create a new user on z/OS.

The demonstration requires the following components:

- A Java application to perform the trust transfer
- A Java application to authenticate the initiating user
- An EIM domain to host the mapping information
- Passticket support for the mapped z/OS user using an LDAP application
- JSec support to create the new user

The following users have been registered in RACF, the EIM target registry, as:

- TSOUSR, the RACF user ID running the Java application on z/OS
- JAVAUSR, the RACF user ID authorized to create a new RACF user

The components of the eimjavademo application are shown In Figure 7-18, with an indication of which z/OS Java APIs are exploited.

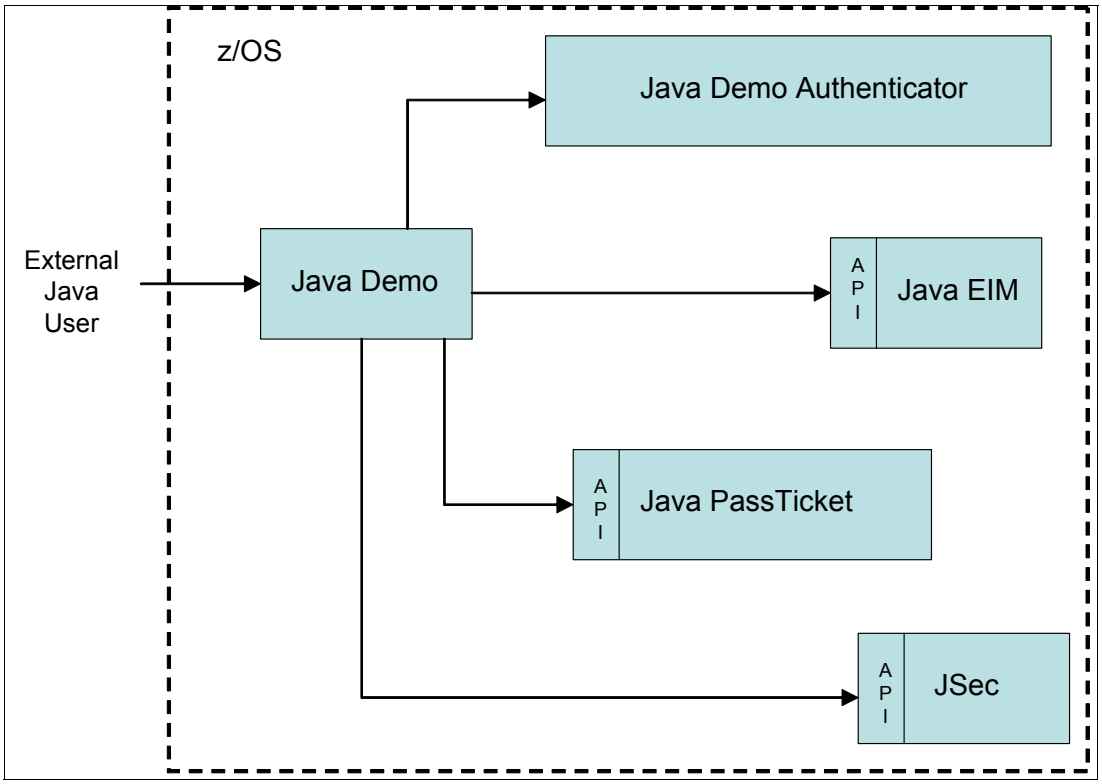


Figure 7-18 The eimjavademo application

The application control flow

The application control flow is illustrated in Figure 7-19 on page 115.

1. A Java user calls the demo application.
2. The demo application authenticates the Java user, using its own authentication mechanism.
3. The demo application maps the Java user to a z/OS user, but a password is needed.
4. The demo application generates a PassTicket for the z/OS user.
5. The demo application establishes a connection to z/OS JSec via LDAP by binding with the z/OS userID and passticket.
6. The demo application can now use JSec to create a new user.

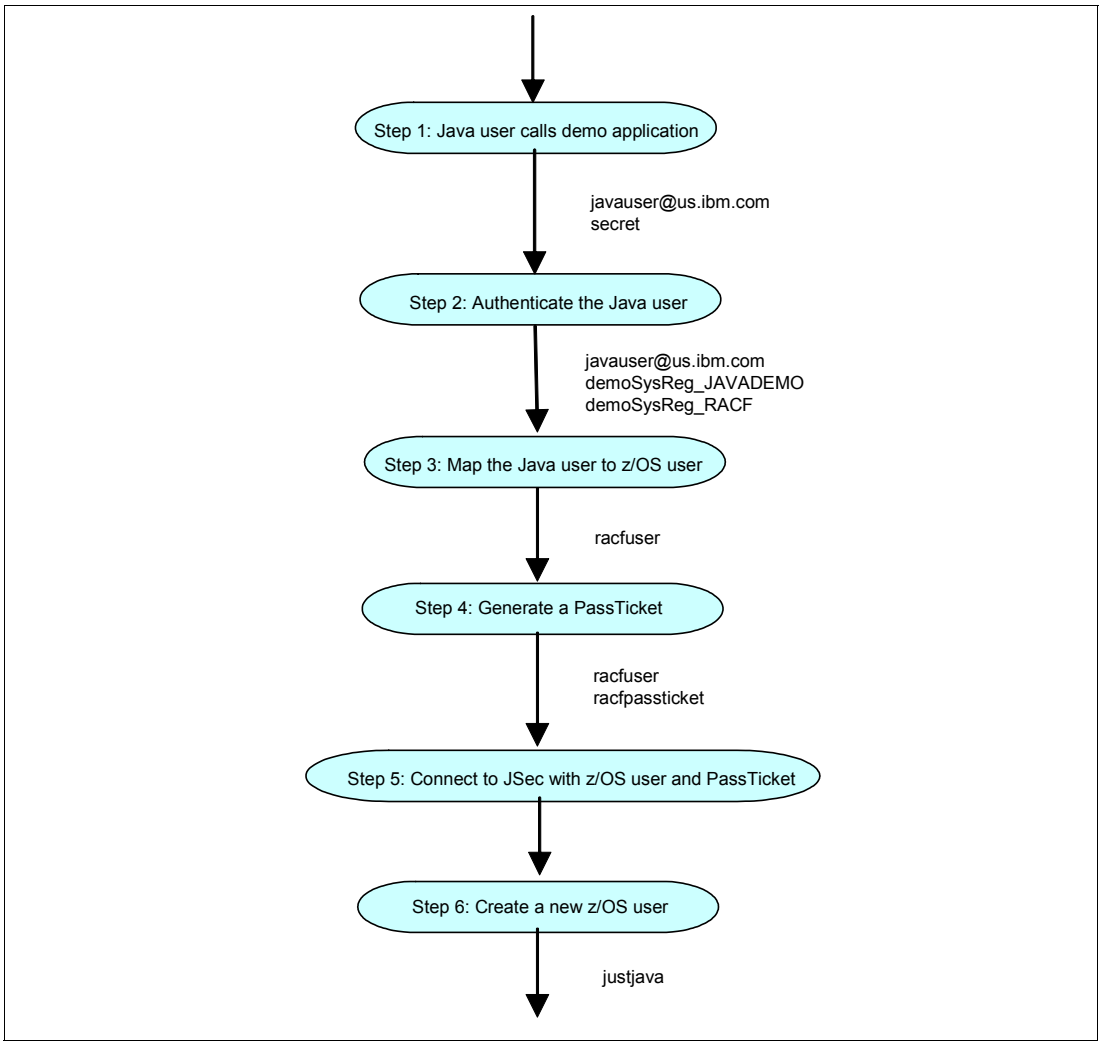


Figure 7-19 The eimjavademo application control flow

The eimjavademo directory tree

The demo application uses the EIM directory tree shown in Figure 7-20 on page 116.

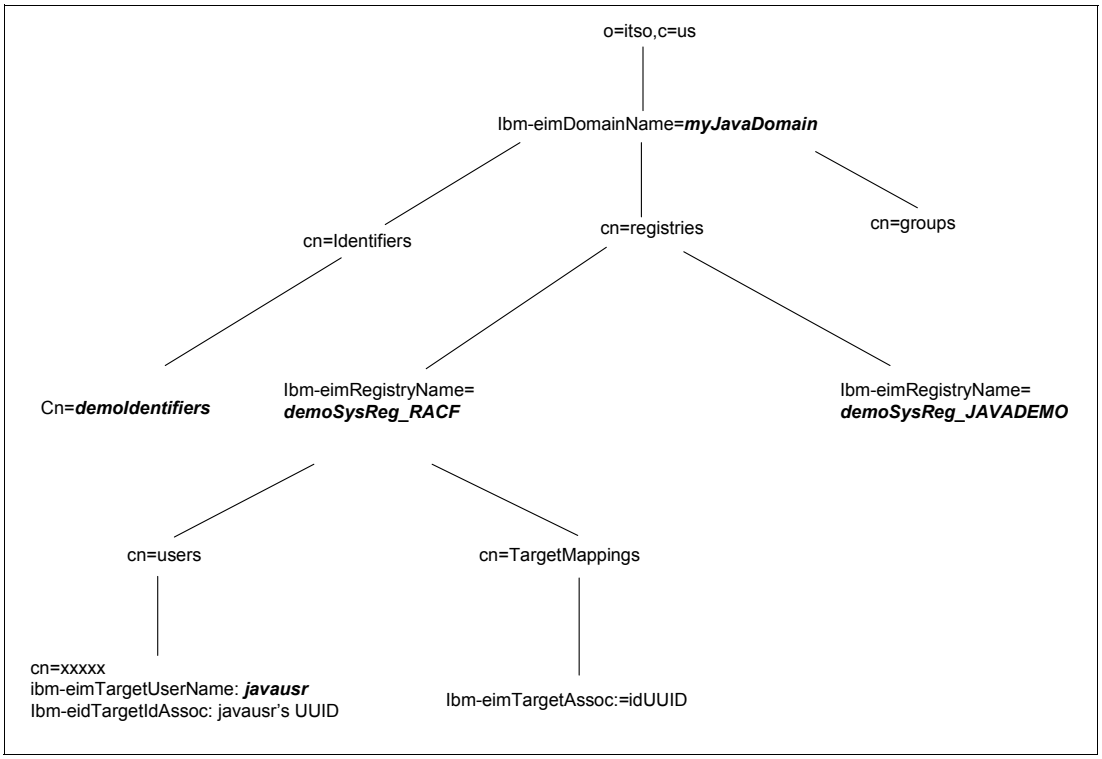


Figure 7-20 The eimjavademo directory tree

7.5.2 Setting up the demo environment

This section shows the specific setups needed to install and run the demo application.

Java setup

The following jar files must be in the java classpath in order to try the demo.

- For the EIM API
  - /usr/lpp/eim/lib/eim.jar
  - /usr/lpp/eim/lib/eimzOS.jar
- For the Passticket API
  - /usr/include/java_classes/IRRRacf.jar
- For the JSec API
  - /usr/include/java_classes/RACFuserregistry.jar
  - /usr/include/java_classes/userregistry.jar

EIM setup

EIM administration is performed via the EIM Java APIs. The relevant administrative functions calls are included in the demo code.

PassTicket setup

Refer to Chapter 6, "RACF PassTicket generation and evaluation by z/OS Java applications" on page 77 for basic setup information.

The PassTicket you obtain will be used to logon to the JSec API, that is, to perform an LDAP authenticated bind (actually verified by RACF through the SDBM back-end).

The application name to be used to qualify the PTKTDATA profile in RACF is, in that case, the job name of the LDAP server started task (ITDS1, in our case).

Furthermore, the z/OS user associated with the PassTicket is JAVAUSR, which is the mapped-to user in EIM. The RACF PTKTDATA profile should therefore be defined as follows:

```
RDEFINE PTKTDATA IRRPTAUTH.ITDS1.JAVAUSR UACC(NONE)
PERMIT IRRPTAUTH.ITDS1.JAVAUSR CLASS(PTKTDATA) ID(TSOUSER) ACCESS(UPDATE)
SETR RACLIST(PTKTDATA) REFRESH
```

LDAP setup for JSec

Refer to Chapter 5, “Java Security Administration” on page 61 for information about JSec Setup.

7.5.3 Running the demo

The sample code used is available in Appendix E, EIM API demo sample code. Three Java files are required to run the sample code:

- eimjavademo.EimJavaSetup.java - creates the EIM domain, registries, identifiers, and associations
- eimjavademo.EimJavaAuth.java - authenticates the java user (this is a stub in place of an actual authentication program)
- eimjavademo.EimJavaDemo.java: - performs the EIM lookup, creates the Passticket, and uses JSec to create a new RACF user

Part 3

z/OS Java cryptography

This part provides a review of cryptography concepts, and then describes and explains how Java stand-alone applications that are executing on z/OS can exploit z/OS and System z integrated hardware cryptography.

© Copyright IBM Corp. 2008. All rights reserved.

8

Introduction to z/OS cryptography and Java

With the advent of the information age and the exploding use of networks of every kind as media for exchanging data and conducting business transactions, enterprises are facing new challenges to IT security.

To help meet these challenges, System z holds to its tradition as being the centerpiece of enterprise IT security policy. It provides state of the art cryptographic services and hardware.

This brief chapter provides an introduction to z/OS cryptography and Java. In the following chapters, the book details how System z and z/OS cryptographic capabilities can be leveraged from applications written in Java.

© Copyright IBM Corp. 2008. All rights reserved.

8.1 Introduction

Enterprise security is crucial to the success of any organization. System z has a long-standing reputation for being an extremely secure and reliable system. It features a security-rich holistic design in which security-relevant controls are enforced on all levels of the computer including the processor hardware level, the operating system level, and the application level.

In this environment, cryptography is the most suitable technique for handling the many security concerns that enterprises face today, including impersonation or having sensitive data stolen or compromised.

It is assumed, in the following chapters, that readers are already familiar with cryptographic techniques and the use of cryptographic algorithms to implement functions such as:

- Encryption (symmetric or asymmetric encryption) Keys
- Digital signature
- Digests and hashing

Familiarity with their specific executions, such as padding, cipher chaining and so on, is also assumed.

For readers who are unfamiliar with these techniques and concepts, refer to a summary of these topics in Appendix F, “Basics of cryptography” on page 259.

Prior to leveraging System z cryptographic capabilities from Java applications, you must properly configure System z hardware and z/OS software. Appendix A, “z/OS integrated hardware cryptography setup details” on page 199, explains the steps required to configure the hardware cryptography infrastructure on System z. The appendix includes detailed information about setting up and using proper RACF protections for the hardware cryptography-related z/OS resources.

After System z, z/OS, and the Java JVM have been configured to operate the hardware cryptographic devices, Java applications can be written that transparently exploit System z hardware cryptography. However, application developers must thoroughly understand the standard application framework provided by the Java SDK. Java Cryptographic Extension (JCE) is a framework that Java applications can invoke to get cryptographic functions to be performed on data. Chapter 9, “Introduction to Java Cryptographic Extension Framework and API” on page 125, provides a detailed description of the JCE framework, the related provider concept, and how the JCE framework can be exploited to interact with ICSF and the cryptographic hardware devices CPACF and the Crypto Express2 Coprocessor.

Cryptographic keys are required for most cryptographic functions. They are installation assets which must be duly secured and managed. z/OS provides strongly secure cryptographic key management functions that can not only be exploited for cryptographic workloads running on the platform itself, but also make z/OS a central hub in an installation for managing cryptographic keys to be used by other platforms. This topic is addressed in Chapter 11, “Java and key management on z/OS” on page 157. Subsequent chapters explain how Java applications can interact with the z/OS key management facilities and use the secure keys mechanism implemented in ICSF and the Crypto Express2 Coprocessor.

Two use cases are used to provide examples of actual implementations of z/OS Java cryptography. Appendix G, “Case study: IBM Encryption Key Manager” on page 267 stresses how the Java-based EKM exploits the z/OS key management facilities to support the critical

function of providing cryptographic keys to be used to secure data that is backed-up on the IBM TS1120 encrypting tape drives.

The second use case, as discussed in Appendix H, “Performance case study: IBM Encryption Facility for z/OS OpenPGP support” on page 275, addresses the Java-based OpenPGP support for the IBM Encryption Facility for z/OS. Here the focus is on how System z-specific hardware components such as the zAAP specialty engine and the CPACF cryptographic device work together to significantly enhance Java cryptographic performance.

9

Introduction to Java Cryptographic Extension Framework and API

This chapter explains what the Java Cryptographic Extension (JCE) framework is, how it operates, and how it can be used to call cryptographic services from Java applications.

© Copyright IBM Corp. 2008. All rights reserved.

9.1 Java Cryptographic Extension overview

Cryptography support in Java is provided by the Java Cryptography Extension (JCE), which has been a part of the Java SDK since Version 1.4. The JCE is an enhancement of the earlier Java Cryptography Architecture (JCA) that appeared with Version 1.1 of the Java SDK. (Note that it is common to see the terms JCE and JCA used interchangeably in documents that are available on the Web.)

The JCE provides a set of cryptographic APIs, which are also referred to as *engine classes*, that Java applications can use. The basic design points of the JCE are algorithm implementations and algorithm independence. This means that Java applications can use these APIs to exploit cryptographic services without having to know the underlying implementation of these services. However, if a Java application wants to use a specific algorithm and implementation, this can also be accomplished with the same set of APIs.

The flexibility of the JCE is achieved using a framework and a provider-based architecture, as explained here:

- The JCE framework defines the set of available cryptographic APIs and provides the infrastructure needed by cryptographic service providers, also referred to simply as “providers”, to plug into the framework.
- The providers are packages that provide the actual implementation of the algorithm and the relevant JCE APIs. This provider-based architecture can provide Java applications access to multiple implementations of the JCE APIs.

  This capability can be used by Java applications to selectively exploit the best providers based on criteria such as whether or not cryptographic coprocessors are available in the system to provide hardware assistance to the algorithm to execute.

9.2 The JCE design point

As stated previously, the JCE design goals are to provide:

- Algorithm independence and extensibility
- Implementation independence

Algorithm independence is achieved via cryptographic engine classes. The cryptographic engine classes’ functionality is implemented by cryptographic service providers.

To use an engine class, a Java application will invoke the getInstance method of the engine class specifying a particular algorithm, and optionally specifying a provider that implements that service. For example, if a Java application wanted to exploit digital signatures, it can do so by getting an instance of a signature object.

```
Signature sign = Signature.getInstance("DSA");
```

This example will obtain a signature object for the Digital Signature Algorithm (DSA) from one of the installed providers. From the example, it is evident that an application can readily switch algorithms by just changing the getInstance method to specify another algorithm.

Algorithm extensibility is easily accomplished by adding support for new algorithms in the engine classes.

Implementation independence is achieved through the provider-based architecture. A *provider* is a package or set of packages that implement one or more of the engine classes.

The JVM can be set up so that a list of providers will be searched in preference order, to find an implementation of a particular type of object (for example, a Signature object using the DSA algorithm.) The provider search order can also be dynamically altered by a Java application, or the provider can be explicitly specified as a parameter on some of the engine classes.

The following example will obtain a signature object for the DSA algorithm from the IBMJCECCA provider.

```
Signature sign = Signature.getInstance("DSA","IBMJCECCA");
```

9.3 The JCE framework

The JCE framework can be thought of as an abstract layer that Java applications can use for security services. The JCE framework defines a consistent and flexible set of security-related Java APIs that emphasize algorithm and implementation independence. The JCE framework also provides the infrastructure that cryptographic service providers use in order to be pluggable into the JCE's provider-based architecture.

The description of the JCE framework requires defining a few terms first:

- Cryptographic service - This is a service that provides a cryptographic function, such as Encryption (also called “Cipher”) or Signature.
- Engine class - This term defines a cryptographic service without providing an implementation; that is, it is an abstract class.

  These are the JCE engine classes:

  - AlgorithmParameterGenerator
  - AlgorithmParameters
  - CertificateFactory
  - Cipher
  - ExemptionMechanism
  - KeyAgreement
  - KeyFactory
  - KeyGenerator
  - KeyPairGenerator
  - KeyStore
  - Mac
  - MessageDigest
  - SecretKeyFactory
  - SecureRandom
  - Signature

An *engine class* defines the interface for a cryptographic service. This is the API that Java applications will use to access the cryptographic services. For each engine class, there is an abstract Service Provider Interface (SPI) class. This interface class defines the abstract methods that every provider must implement if it is to support that cryptographic service.

Therefore, if a provider supports a cryptographic service, it would have had to subclass the SPI for that engine class and provide a concrete implementation of all the abstract methods declared in the SPI.

When a Java application needs to use the cryptographic service, it must first invoke the getInstance method for that engine class. The getInstance class will create an instance of that engine class and encapsulate an instance of the corresponding SPI class from the provider that is supplying the concrete implementation of the SPI abstract methods. This

linkage ensures that the correct provider implementation for the cryptographic service is used whenever methods from that instance of the engine class are invoked.

9.4 Cryptographic service providers

As mentioned, a *cryptographic service provider* is a package or set of packages that implement one or more engine classes. A provider has the option to implement all the engine classes, or only a subset of them.

A provider is always a subclass of the java.security.Provider class. A provider object can be queried to list all the security services and algorithms that it supports.

A JVM can have access to more than one provider. If a provider is not explicitly specified on an API invocation, all the available providers are searched in preference order until the first provider that supplies an implementation of the needed service and appropriate algorithm type is found.

9.4.1 Installing and configuring providers

The JCE framework authenticates a provider and ensures that only trusted providers can be plugged into the framework. If users are writing their own providers, the jar files must be signed by a certificate that must be obtained from Sun, before it can be used by the JCE. Any provider made available by IBM is already properly signed.

You need to install a provider before trying to reference it, unless the provider comes as a standard component with the version in use of the Java JDK. A provider is usually packaged as a jar file. The provider jar file should be placed in the /lib/ext/ directory under the Java installation directory. For example, for Java 5.0, the directory where the provider jar files are located is:

```
/usr/lpp/java/J5.0/lib/ext/
```

Providers must also be configured, in addition to being installed. Providers can be configured in either of the following ways:

- Statically, by configuring the JVM
- Dynamically, while the Java application is running

Statically configuring a provider

The java.security file contains information about the installed providers and specifies a preference order among them. When a provider is not explicitly specified on a Java API call (that is, on Java API calls that support such a parameter), then the preference order in java.security is used to search for the first provider that implements the requested service and algorithm.

The java.security file is located in the /lib/security/ directory under the Java installation directory. For example, for Java 5.0, the java.security file is located under:

```
/usr/lpp/java/J5.0/lib/security/
```

Figure 9-1 on page 129 displays the section of the java.security file that is relevant to the cryptographic providers.

```
 security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
security.provider.2=com.ibm.crypto.provider.IBMJCE
security.provider.3=com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

Figure 9-1 A java.security providers list

To register a new provider, an entry must be added to specify the provider subclass name and the preference order for that provider. The entry has the following format:

```
security.provider.<n>=<className>
```

Note that `n` specifies the preference order. 1 is the most preferred provider. `className` specifies the name of the subclass that is implementing the Provider class.

Providers registered via the java.security file are instantiated when the JVM is initialized. The provider settings will be in effect for any JVM that is created using that Java installation configuration.

Careful planning may be necessary to ensure that the provider precedence behavior will be as expected. If the first provider in the list supplies every service and algorithm that applications will use, then this provider will always be the one selected, unless the Java application code explicitly specifies another particular provider in the list

Dynamically configuring a provider

A provider can be dynamically added or its preference order can be dynamically changed by a Java application. This can be accomplished by the addProvider, insertProviderAt, or removeProvider methods from the class java.security.Security. The changes are not persistent and the changes are only in effect for the Java application that dynamically alters the installed providers.

If the Java Security Manager is enabled, a Java application will need the appropriate permissions to modify the installed provider list (refer to 1.4.1, “The Java Virtual Machine Security framework components” on page 14 for a discussion on the Java Security Manager and permissions).

The addProvider method adds a provider to the bottom of the list of available providers. This implies that this provider will be *lowest* in preference order. Note that a provider will not be added if it is already installed. The following code example will dynamically add the IBMJCE provider:

```
Security.addProvider(new IBMJCE());
```

To add a provider at a particular preference order, you must use the insertProviderAt method. This method will add the provider at the preference order that is specified. If there is already a provider installed at that preference order, that provider and all the ones with lower precedence are moved down one position.

A provider will not be added if it is already installed. This means that the insertProviderAt method cannot be used to change the position of an installed provider. The following code example will dynamically add the IBMJCE provider at precedence order 2:

```
Security.insertProviderAt(new IBMJCE(), 2);
```

To change the precedence order of an installed provider, you must remove the provider and then reinsert it at the desired precedence order. The removeProvider method is used to remove a provider from the provider list.

When a provider is removed from the provider list, all providers lower in precedence to the removed provider will move up one position in precedence order. The following code example can be used to change the precedence order of the IBMJCE provider to position 2:

```
Security.removeProvider("IBMJCE");
Security.insertProviderAt(new IBMJCE(), 2);
```

Querying providers

Because all providers extend the java.security.Provider class, a Java application has the option of programmatically querying all installed providers and the services that they support. The Java application shown in Figure 9-2 will query all installed providers and display each of them, as well as the services they provide.

```
import java.security.Provider;
import java.security.Security;

public class Example3 {
     public static void main(String[] args) {
        Provider[] providers = Security.getProviders();
        for (int i = 0; i < providers.length; i++) {
            System.out.println(providers[i].getInfo());
        }
     }
}
```

Figure 9-2 Querying providers

9.4.2 Policy files

Due to export restrictions, the IBM JDKs ship with a set of restricted policy files that limit the size of the cryptographic keys that are supported. To overcome these restrictions, the user will need to obtain the “unrestricted policy files” and substitute them to files that are supplied with the JDK. The security files are located in the $JAVA_HOME/lib/security directory and are named:

- local_policy.jar
- US_export_policy.jar

The unrestricted policy files are the same for the IBM JDK 1.4.2, IBM JDK 5 and IBM JDK 6. These files can be downloaded from:

http://www.ibm.com/servers/eserver/zseries/software/java/j6jcecca.html

9.5 The IBM providers

The JCE provider-based architecture allows Java applications to be platform-neutral and still be able to exploit the strengths of each particular platform. The Java applications can invoke different providers, depending on the host system they execute in, and the provider implementations will exploit the strengths of each particular host system.

The implementation independence of the JCE architecture ensures that a wide range of providers is available. The providers can be implemented, for example, to exploit specialized hardware devices or programs available on host platforms. On z/OS, the exploitation of System z hardware cryptographic devices can provide benefits such as:

- Increased speed in performing cryptographic functions
- Offloading the cryptographic functions and preserving CPU MIPS for running the workloads' business logic
- Providing a physically more secure environment for algorithms and keys within the physical boundary of the coprocessors.

The IBM z/OS JDK comes with a large set of providers, some of which are listed here:

- IBMJCE (ibmjceprovider.jar) - Software implementation for the JCE.
- IBMJCECCA (ibmjcecca.jar) - Implementation for the JCE that supports hardware cryptographic devices on z/OS.
- IBMJCE4578 - Implementation for the JCE that supports hardware cryptographic devices on z/OS. The IBMJCECCA is a compatible replacement for the IBMJCE4578 provider.
- IBMJCEFIPS (ibmjcefips.jar) - JCE implementation that is compatible with the FIPS 140-2 standard.
- IBMJSSE - Java Secure Sockets Extension (SSL and TLS).
- IBMJSSEFIPS - JSSE implementation that is compatible with the FIPS 140-2 standard.
- IBMJSSE2 (ibmjsseprovider2.jar) - Newer version of the IBMJSSEI.
- IBMJAAS - Java authentication and authorization service.
- IBMJGSS (ibmjgssprovider.jar) - Generic security services (Kerberos and GSS-API support).
- IBMPKCS11Impl (ibmpkcs11impl.jar) - Public Key Cryptography standard #11 cryptographic API.
- IBMSASL (ibmsaslprovider.jar) - Simple authentication and security layer framework implementation.

Most of these providers are the same, regardless of platform. However, the IBMJCECCA provider is specific to the z/OS platform and will exploit the System z hardware cryptographic devices.

The IBMPKCS11Impl provider works the same as on other platforms, but because PKCS#11 hardware tokens are not supported on z/OS, a discussion of the conceptual differences in using PKCS11 on z/OS compared to using it on distributed platforms is provided in 9.7, "The IBM providers - IBMPKCS11Impl" on page 138. The other providers basically work the same on all the platforms, whether distributed or z/OS, and are only briefly described.

9.5.1 IBMJCE

The IBMJCE provider is a multiplatform software implementation of the JCE. This provider should be used if platform compatibility is a concern for the specific Java application to be run on z/OS.

The IBMJCE provider supports the following algorithms:

- Cipher
  - RSA encryption/decryption
  - Blowfish
  - DES
  - Triple DES
  - Mars
  - RC2
  - RC4
  - Seal
  - PBE with MD2 and DES
  - PBE with MD2 and Triple DES
  - PBE with MD2 and RC2
  - PBE with MD5 and DES
  - PBE with MD5 and Triple DES
  - PBE with MD5 and RC2
  - PBE with SHA1 and DES
  - PBE with SHA1 and TripleDES
  - PBE with SHA1 and RC2
  - PBE with SHA1 and 40-bit RC2
  - PBE with SHA1 and 128-bit RC2
  - PBE with SHA1 and 40-bit RC4
  - PBE with SHA1 and 128-bit RC4
  - PBE with SHA1 and 2-key Triple DES
  - PBE with SHA1 and 3-key Triple DES
- Signature
  - RSA/SHA1, RSA/MD5, RSA/MD2 signatures
  - DSA/SHA1 signature
- Message Digest
  - SHA1
  - MD5
  - MD2
  - Message Authentication code
  - Hmac/SHA1
  - Hmac/MD2
  - Hmac/MD5
- Key agreement algorithms
  - DiffieHellman
  - Random number generation
  - Secure Random

It also supports the following keystores:

- JCEKS
- JKS
- PKCS12KS

9.5.2 IBMJCE4578

The IBMJCE4578 provider is specific to z/OS and exploits cryptographic hardware. The IBMJCE4578 provider has been replaced with the compatible IBMJCECCA provider. For

more details about the IBMJCECCA provider, refer to 9.6, “The IBM providers - IBMJCECCA” on page 136.

9.5.3 IBMJSSE2 and IBMJSSE

The IBMJSSE2 (Java Secure Socket Extension) and IBMJSSE providers enable secure Internet communications. They implement a Java version of Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols, and include functions for data encryption, server authentication, message integrity, and client authentication.

IBMJSSE is the older version and has been superseded by IBMJSSE2.

The original IBMJSSE provider used its own internal cryptographic code. The IBMJSSE2 provider instead relies on the cryptographic services provided by the IBMJCE, IBMJCECCA, or IBMPKCS11Impl providers.

Because the IBMJSSE2 provider no longer contains cryptographic code, it is not required to be FIPS-certified. Instead, it leverages the cryptographic services provided by the IBMJCEFIPS provider. This means that instead of using the IBMJSSEFIPS provider, the preferred method is to use the IBMJSSE2 provider that has been configured to use the IBMJCEFIPS provider.

These providers are multiplatform-compatible and provide the same functionality on all the supported platforms. However, because of provider pluggability, the IBMJSSE2 can be configured to use any JCE provider. This means that on z/OS, the IBMJSSE2 provider can be configured to exploit hardware cryptographic devices via the IBMJCECCA provider.

The IBMJSSE2 reference guide is available at the following Web site:

http://www.ibm.com/developerworks/java/jdk/security/60/secguides/jsse2Docs/JSSE2RefGuide.html#plug

9.5.4 IBMJCEFIPS and IBMJSSEFIPS

The IBMJCEFIPS and IBMJSSEFIPS providers support only FIPS-compliant cryptographic operations and algorithms. These providers are compliant with the Federal Information Processing Standards (FIPS) 140-2 Level 1.

These providers are to be used by Java applications that need to be FIPS-compliant. These providers are multiplatform-compatible and provide the same functionality on all the supported platforms.

The IBMJCEFIPS provider does *not* contain a keystore. Key storing and management facilities are the user’s responsibility.

The following are the supported algorithms for IBMJCEFIPS provider v1.2:

- Ciphers
  - AES with modes ECB, CBC, OFB and CFB
  - DESede(TripleDES) with modes ECB, CBC, OFB and CFB
  - RSA with PKCS#1 padding (This is a non-approved algorithm. It can only be used to encrypt and decrypt keys for transport to stay within the boundaries of the approved mode of FIPS 140-2 Level 1.)
- Signature
  - SHA1withDSA

  - SHA1withRSA
- Key Agreement
  - DiffieHellman (This is a non-approved algorithm, but is allowed for use in exchanging keys.)
- Key(pair) generation
  - DSA
  - AES
  - TripleDES
  - HmacSHA1
  - RSA
  - DiffieHellman
- Message authentication code (MAC)
  - HmacSHA1
- Message digest
  - SHA-1
  - SHA-256
  - SHA-384
  - SHA-512
  - MD5 (This is a non-approved algorithm. It can only be used if the user is implementing the TLS protocol for Secure Sockets. Any other use will cause the application to be non-compliant to FIPS 140-2 Level 1.)
- Algorithm parameter generator
  - DiffieHellman
  - DSA
- Algorithm parameter
  - AES
  - DiffieHellman
  - TripleDES
  - DSA
- Key Factory
  - DiffieHellman
  - DSA
  - RSA
- Secret Key Factory
  - AES
  - TripleDES
- Certificates
  - X.509
- Random number generation
  - IBMSecureRandom which is also known as FIPSPRNG

The following are the supported cipher suites for the IBM JSSE FIPS provider. This provider supports only the TLS protocol.

SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_FIPS_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA

SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_AES_128_CBC_SHA
SSL_RSA_WITH_AES_256_CBC_SHA
SSL_DHE_RSA_WITH_AES_128_CBC_SHA
SSL_DHE_RSA_WITH_AES_256_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_WITH_AES_128_CBC_SHA
SSL_DHE_DSS_WITH_AES_256_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DH_anon_WITH_AES_128_CBC_SHA
SSL_DH_anon_WITH_AES_256_CBC_SHA
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA

9.5.5 IBMJAAS

The Java Authentication and Authorization Service (IBMJAAS) provider allows you to:

- Authenticate users - That is, reliably and securely validate the identity of the user who is executing the Java code
- Authorization - That is, ensure that an authenticated user has the permissions to do the action that is being requested

JAAS gives Java the capability to provide access controls based on the identity of an authenticated user. This capability complements the imbedded Java capability to provide access controls based on the executing code's code source and code signer. The IBMJAAS provider is multiplatform and provides the same functionality on all supported platforms.

The IBMJAAS reference guide is available at the following Web site:

http://www.ibm.com/developerworks/java/jdk/security/60/secguides/JaasDocs/api.html

9.5.6 IBMJGSS

The IBMJGSS provider is a Java-implemented Generic Security Service Application Programming Interface (GSSAPI) framework with Kerberos V5 as the underlying default security mechanism. GSSAPI is a standardized abstract interface under which can be plugged different security mechanisms based on security technologies that use either symmetric or asymmetric cryptographic algorithms.

GSSAPI shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. GSSAPI provides identity and message origin authentication, message integrity, and message confidentiality.

The IBMJGSS user's guide is available at the following Web site:

http://www.ibm.com/developerworks/java/jdk/security/60/secguides/jgssDocs/users_guide.jgss.ibm.html

The IBMJGSS application developer's guide is available at the following Web site:

http://www.ibm.com/developerworks/java/jdk/security/60/secguides/jgssDocs/developers_guide.jgss.ibm.html

9.5.7 IBMSASL

The IBMSASL provider provides an implementation of the Simple Authentication and Security Layer (SASL) framework that the Java SASL APIs can use. SASL is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications.

SASL defines how authentication data is to be exchanged, but does not specify the contents of that data. SASL is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

The IBMSASL provider guide is available at the following Web site:

http://www.ibm.com/developerworks/java/jdk/security/60/

9.6 The IBM providers - IBMJCECCA

The IBMJCECCA provider extends the JCE capability with the use of hardware cryptography via the IBM Common Cryptographic Architecture (CCA) API provided by ICSF on z/OS. The IBM CCA is a set of software elements that provide common application interfaces to IBM-provided secure, high speed cryptographic services on various platforms via hardware cryptographic devices. Refer to “Cryptographic coprocessors” on page 200, for information about the types of cryptographic devices supported on z/OS.

The IBMJCECCA provider takes advantage of hardware cryptography within the existing JCE architecture. It gives Java programmers the significant security and performance advantages of hardware cryptography with minimal changes to existing Java applications. In most cases, the only changes needed to a Java application are the method invocations that specify the IBMJCE provider to be changed to specify the IBMJCECCA provider.

The IBMJCECCA provider can also be specified in the java.security file. The precedence order of the IBMJCECCA provider should always be *higher* than the IBMJCE provider. If this is not the case, any request for Java cryptographic services that do not explicitly specify the provider will always be routed to the IBMJCE provider, because the IBMJCE provider supports a superset of the algorithms supported by the IBMJCECCA provider. In such a scenario, a Java program will not exploit the benefits of hardware cryptography as provided by the IBMJCECCA provider.

One of the prerequisites for using the IBMJCECCA provider is that the required hardware cryptographic devices be installed, configured and operational, as explained in Appendix A, “z/OS integrated hardware cryptography setup details” on page 199.

Attention: If the required hardware cryptographic devices are not available, some of the IBMJCECCA services can fail. In such cases, there is no failover capability.

In other words, if the hardware needed to service a request is not available, the service will fail. There will be no retry invoking a software implementation of the same service.

The IBMJCECCA provider interfaces with the hardware cryptographic devices through the Integrated Cryptographic Service Facility (ICSF) component of z/OS. For more information about ICSF, refer to Appendix A, “z/OS integrated hardware cryptography setup details” on page 199.

The IBMJCECCA provider uses a Java Native Interface (JNI) module to interface with the ICSF API. If the ICSF API returns a non-zero return code, the Java service will throw an exception that will contain the ICSF return and reason code.

The IBMJCECCA provider supports the following:

- Signature algorithms
  - SHA1withDSA*
  - SHA1withRSA
  - MD5withRSA
  - MD2withRSA
- Cipher algorithms
  - DES
  - TripleDES
  - AES
  - PBEWithMD2AndDES
  - PBEWithMD2AndTripleDES
  - PBEWithMD5AndDES
  - PBEWithMD5AndTripleDES
  - PBEWithSHA1AndDES
  - PBEWithSHA1AndTripleDES
  - PBEWithSHAAnd2KeyTripleDES
  - PBEWithSHAAnd3KeyTripleDES
  - RSA
- Message authentication code (MAC)
  - HmacSHA1
  - HmacMD2
  - HmacMD5
- Key (pair) generator
  - DSA (supported on machines prior to the IBM 2084 eServer zSeries 900 only)
  - DES
  - TripleDES
  - HmacMD2
  - HmacMD5
  - HmacSHA1
  - RSA
- Message digest
  - MD2
  - MD5
  - SHA-1
- Algorithm parameter generator
  - DSA (supported on machines prior to the IBM 2084 eServer zSeries 900 only)
- Algorithm parameter
  - DES
  - TripleDES
  - DSA (supported on machines prior to the IBM 2084 eServer zSeries 900 only)

- AES
- PBEwithMD5AndDES

- Key factory
  - DSA (Only supported on machines prior to the IBM 2084 eServer zSeries 900)
  - RSA

- Secret key factory
  - DES
  - TripleDES
  - AES
  - PKCS5Key
  - PBKDF1
  - PBKDF2(PKCS5DerivedKey)

- Certificate
  - X.509

- Secure random
  - IBMSecureRandom

The IBMJCECCA also supports the following keystores

- JCECCAKS
- JCECCARACFKS
- JCA4758KS

9.7 The IBM providers - IBMPKCS11Impl

RSA Laboratories' Public Key Cryptography Standards #11 (PKCS#11) was initially created to exploit smart cards or other plug-installable simple cryptographic devices accessed through a physical interface such as a smart card reader.

In PKCS#11 terminology, a “token” is a hardware device (such as the smart card) that, in addition to providing accelerated cryptographic operations via hardware engines, can also securely store information of the following types:

- Application specific data
- Certificate
- Key

PKCS#11 tokens are usually initialized before first use. For example, a banking customer may be given a token that already contains a certificate and key that can be eventually used to properly identify the owner of that token. The token is also assigned a secret PIN code that must be provided in order to retrieve or use the stored information.

Note that because PKCS#11 tokens are expected to be real physical devices, the standard does not provide any mechanism to delete them.

When administering tokens, there are two roles defined. One role is the standard user (USER) who will be using the actual token via an application that invokes the PKCS#11 API. The other role is the Security Officer (SO) who is responsible for initializing the token whenever it is required as a preamble to the use by the standard user.

As a storage device, the token gives a conceptual view of the stored data as:

- Data objects to store application specific data

- Certificate objects to store certificates
- Key objects to store keys (which can be public, private, or secret keys)

A token can contain multiple data, certificate, or key objects. The objects are classified according to their lifetime and visibility. Token objects are visible to all applications connected to the token (a *session* is defined as the connection that an application makes to a token), provided that the applications have sufficient permission and the objects are persistent on the token, even after the sessions are closed or the token is removed from the card reader.

The “session objects” are temporary in nature. When a session is closed, all session objects created by that session are automatically deleted. Session objects are only visible to the application that created them.

Token and session objects can also be assigned attributes. Attributes give specific properties to the objects, such as whether their contents are of a public or private nature.

9.7.1 Specific z/OS considerations for PKCS#11

The concept of users using their own physical token through a physical interface, such as a card reader, does not lend itself to a System z platform implementation. Likewise, the PKCS#11 access protection model does not directly translate into the multiple users and privileges granularity approach used by z/OS.

However, any rationale for using PKCS#11 applications still applies whether the application is running on a personal computer or on z/OS. Therefore, z/OS provides a PKCS#11 provider for Java JDK 6 that allows applications written for other platforms’ implementation of PKCS#11 to be used on z/OS as well, with just a simple recompilation.

To support PKCS#11 on z/OS, the conceptual view of the token is preserved. However, tokens are no longer physical devices but storage areas provided in an ICSF-managed dataset. The hardware cryptographic engines of a physical token are replaced by access provided to the System z hardware cryptographic devices to the PKCS#11 application.

The z/OS ICSF component provides support for the PKCS#11 API at z/OS V1R9. The z/OS PKCS#11 tokens are virtual tokens and are records in a VSAM dataset called the Token Key Data Set (TKDS). Because the tokens are virtual, they can be created at any time and can also be deleted. Instead of using a PIN to control token access in z/OS, profiles in the RACF CRYPTOZ class of resources are used to give users, or security officers, access to the tokens in the TKDS.

The following is a link to the z/OS IBMPKCS11Impl guide:

`http://www.ibm.com/servers/eserver/zseries/software/java/j6pkcs11implgd.html`

Configuring the PKCS#11 provider on z/OS

Unlike other providers, to be able to use the IBMPKCS11Impl provider, a preliminary configuration is necessary that addresses:

- Allocation of the ICSF Token Key Dataset (TKDS), creation of a z/OS PKCS#11 token, and setup of access controls for the token and ICSF services
- The configuration changes required to the Java environment

Only the configuration of the Java environment is discussed in this chapter. Configuration of ICSF and RACF are discussed in Appendix A, “z/OS integrated hardware cryptography setup details” on page 199. You can also refer to other relevant IBM documentation, including:

- *z/OS Cryptographic Services Integrated Cryptographic Service Facility Writing PKCS #11 Applications,* SA23-2231
- The IBM Redbooks publication *System z Cryptographic Services and z/OS PKI Services*, SG24-7470

Configuration changes to the Java environment for the IBMPKCS11Impl provider entail adding the IBMPKCS11Impl provider to the list of security providers in the java.security file and creating an IBMPKCS11Impl configuration file.

An IBMPKCS11Impl configuration file contains the following information:

- *name* - a string to append to IBMPKCS11Impl- to create the provider's instance name. For example, using the configuration file in the following example, a Java application will use the following String value to reference this provider:

  `IBMPKCS11Impl-ATMAPP`

- *library* - the HFS full path name of the PKCS#11 DLL library.
  - For 31-bit addressing mode, this will be /usr/lpp/pkcs11/lib/csnpcapi.so.
  - For 64-bit addressing mode, this will be /usr/lpp/pkcs11/lib/csnpca64.so.
- *description* - a string to describe the configuration.
- *tokenLabel* - the name of the token. Access to the token and the objects it contains is controlled via profiles in the RACF CRYPTOZ class of resources.

Example 9-1 shows a sample IBMPKCS11Impl configuration. It assumes that the name of the file is atmapp.cfg.

Example 9-1 IBMPKCS11Impl configuration file

```
name = ATMAPP
library= /usr/lpp/pkcs11/lib/csnpcapi.so
description=ATM Application
tokenLabel=CUSTOMER1
```

This IBMPKCS11Impl configuration file must reside in an HFS directory. Before using the provider, it must be initialized. The following methods can be used to initialize the provider:

- The provider can be specified in the java.security file. As part of specifying the IBMPKCS11Impl provider, the configuration file is also specified, as shown in Example 9-2.

Example 9-2 The java.security file with the IBMPKCS11Impl provider

```
security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
security.provider.2=com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
security.provider.3=com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl
/home/user1/atmapp.cfg
security.provider.4=com.ibm.crypto.provider.IBMJCE
security.provider.5=com.ibm.security.jgss.IBMJGSSProvider
security.provider.6=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

- The provider can be programmatically initialized with the Init method:

  `com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl.Init(`*`String`*`, char[])`

  The *String* parameter is the full directory path and name of the configuration file. The char[] parameter is the password and is ignored.

- The provider can be programmatically initialized when the provider object instance is created

  `com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl(`*`String`* `fullFileName)`

  The *String* parameter is the name of the IBMPKCS11Impl configuration file.

The recommendation is to initialize the provider programmatically via the Init method or when the provider object is created. Updating the java.security file sets the provider for all instances of the JVM. A PKCS#11 session is created when a JVM loads and initializes the IBMPKCS11Impl provider.

Note the following points in regard to the PKCS#11 support on z/OS:

- Only clear keys are supported in the ICSF TKDS. There is no Java PKCS#11 secure key support. If secure key support is needed, the user should consider using the IBMJCECCA provider instead.
- Creation of a new empty token is required before a Java IBMPKCS11Impl application can be run. Token creation can be done using the RACF RACDCERT command or the System SSL gskkyman UNIX program.

  Refer to *z/OS Security Server RACF Command Language Reference*, SA22-7687, and *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683, for information about the RACDCERT command.

  Refer to *z/OS Cryptographic Services System Secure Sockets Layer Programming,* SC24-5901, for information about gskkyman.

  In addition, a token can be created using ICSF ISPF panels for PKCS#11. Refer to z/OS Cryptographic Services Integrated Cryptographic Services Facility Administrator's Guide, SA22-7521, for more information about this topic.

- Support is not available in the IBMPKCS11Impl provider for PKCS#11 hardware exploitation by the IBMJSSE2 provider. If hardware cryptographic devices exploitation by the IBMJSSE2 provider is needed, the user should consider using the IBMJCECCA provider instead.
- If a com.ibm.pkcs11.PKCS11Exception is thrown that includes the string `Vendor defined error`, the code displayed is an ICSF return code and reason code. It is in the format `0x'pxxxyyyy'` where `xxx` is the ICSF return code and `yyyy` is the ICSF reason code (the `p` value can be ignored).

  To determine the meaning of these codes, refer to *z/OS Cryptographic Services ICSF Application Programmer's Guide*, SA22-7522, Appendix A.

- If a java.security.NoSuchAlgorithmException is thrown when running an IBMPKCS11Impl application and the algorithm is one that is supported, this may be due to the z/OS hardware cryptography not being in operation in the system.
- We strongly recommend that a call to com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl.removeSession() method be included at the end of every Java IBMPKCS11Impl application to remove and close the session that is currently associated with the provider. This avoids a build-up of a large

amount of session objects in the TDKS because all the session objects created by that session are destroyed when it is closed.

Calling the com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl.logout() method is not required because it has no effect on the PKCS#11 session state in the z/OS implementation. Logout and login are not needed (and actually will be null operations, if invoked) because access to the token and the objects it contains is controlled by RACF profiles only.

- When an IBMPKCS11Impl key is stored into the PKCS11IMPLKS key store using the KeyStore.setKeyEntry method, the key object is copied to the TKDS and the token attribute is set to true. The IBMPKCS11Impl key passed into setKeyEntry is no longer valid. A call to the KeyStore.getKey method must be done to retrieve the valid key object from the TKDS.
- It is strongly recommended that separate tokens be used if the Java PKCS#11 application runs in multiple JVMs or processes and adds and deletes objects in the TKDS.

  For example, suppose the Java PKCS#11 application on JVM#1 creates a new RSA public and private key pair using TOKEN1.

  It then does a KeyStore.load, followed by a Keystore.setKeyEntry for the new RSA private key, followed by a Keystore.deleteEntry of that RSA private key.

  On JVM #2, if the Java PKCS11 application does a Keystore.load also using TOKEN1 prior to the KeyStore.deleteEntry being done in JVM #1, unpredictable results may occur.
- When using the TokenLabel specification in the IBMPKCS11Impl configuration file, better performance is achieved for the IBMPKCS11Impl instance initialization when the CRYPTOZ profiles qualifiers match, or tend to be specific to, the TokenLabel value.

  For example, assume that TokenLabel specifies DAN1 and we define CRYTPOZ profiles defined as SO.DAN* and USER.DAN* (these are generic RACF profiles granting Security Officer and User privileges to tokens with a label name beginning with DAN) to give permission to the user DAN. The IBMPKCS11Impl instance initialization will complete faster in such a case than if we had defined the CRYPTOZ profiles as SO.* and USER.*.
- Tracing for the IBMPKCS11Impl provider can be enabled by adding the following line to the Java command:

  ```
  -Djava.security.auth.debug=all
  ```

  Details for enabling the z/OS PKCS#11 DLL are described in *z/OS Cryptographic Services ICSF Writing PKCS11#11 Applications*, SA23-2231. Tracing for the Java PKCS#11 JNI layer for z/OS can be enabled by issuing the command:

  ```
  export HJV_PKCS11_TRACE=ON
  ```

  Note that tracing should only be enabled when requested by an IBM service representative due to the volume of output that might be produced.

Supported algorithms and Implemented services

The IBMPKCS11Impl provider supports the following on z/OS:

- Signature
  - MD5withRSA
  - SHA1withRSA
  - SHA256withRSA
  - RSAforSSL
- Cipher
  - RSA/SSL/PKCS1Padding

  - RSA/SSL/NoPadding
  - RSAforSSL
  - DES/CBC/NoPadding
  - DES/CBC/Pad
  - DES/ECB/NoPadding
  - DESede/CBC/NoPadding
  - DESede/CBC/Pad
  - DESede/ECB/NoPadding
  - AES/CBC/NoPadding
  - AES/CBC/Pad
  - AES/ECB/NoPadding
- KeyPairGenerator
  - RSA
- KeyGenerator
  - DES
  - DESede
  - AES
- MessageDigest
  - MD5
  - SHA1
  - SHA-256
- KeyFactory
  - RSA
- SecretKeyFactory
  - DES
  - DESede
  - AES
- KeyStore
  - PKCS11IMPLKS

10

Simple examples of Java cryptography

As described in Chapter 9, “Introduction to Java Cryptographic Extension Framework and API” on page 125, the providers for the Java cryptographic services implement one or more of the engine classes required for JCE functionality.

This chapter provides examples of the basic usage of the following engine classes:

- SecureRandom
- MessageDigest
- Signature
- Cipher
- KeyGenerator
- KeyPairGenerator
- KeyFactory
- SecretKeyFactory
- KeyStore

For a more complete explanation of engine classes, refer to *Java Cryptography Architecture (JCA) Reference Guide* at:

`http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`

Note that these examples use keys that are dynamically generated, as opposed to keys kept in keystore. Keystore concept implementation and its uses are discussed in Chapter 11, “Java and key management on z/OS” on page 157.

© Copyright IBM Corp. 2008. All rights reserved.

10.1 Engine classes

All engine class objects are obtained through a call of the static factory method getInstance() on the engine class. The following statement instantiates a SecureRandom object that implements the IBMSecureRandom algorithm, if an IBM JCE cryptographic provider exists in the cryptographic service provider list that supports the algorithm.

```
SecureRandom random = SecureRandom.getInstance("IBMSecureRandom");
```

If several providers in the list support the algorithm, then the provider that comes first will be used.

If the user wants to use the algorithm implementation of a specific provider, this can be done by specifying the name of the provider as the second argument of **`getInstance()`**:

```
SecureRandom random = SecureRandom.getInstance("IBMSecureRandom", "IBMJCECCA");
```

This statement will ensure that the SecureRandom object returned is created using the IBMJCECCA provider.

Note the following points:

- In the case where no providers support the requested algorithm, a NoSuchAlgorithmException exception is thrown.
- In the case where an unknown provider is specified, then a NoSuchProviderException exception is thrown.

10.1.1 SecureRandom

The random number generation as provided in Java is, in fact, not fully random. Instead, pseudo random number generation techniques are used. Knowledge of what is the initialization value provided to a pseudo random number generator (the “seed”) eventually allows identical numbers to be generated.

Because one of the main purposes of random number generation is the creation of unpredictable secret keys values, it is of paramount importance to have access to a pseudo random number generation facility that introduces the proper level of entropy in the generation process. The SecureRandom engine class provided in Java is such a strong pseudo random number generator.

Note: The IBMJCECCA provider exploits the generation of true random numbers as implemented in the Crypto Express2 Coprocessor. This capability is provided to applications that invoke the SecureRandom class.

A SecureRandom object is created as follows:

```
SecureRandom random = SecureRandom.getInstance("IBMSecureRandom");
```

where the IBM implementation SecureRandom is specified. To exploit the hardware support for random number generation, you must specify:

```
SecureRandom random = SecureRandom.getInstance("IBMSecureRandom", "IBMJCECCA");
```

The generation of random numbers can the be invoked as follows:

```
byte[] randomBytes = new byte[16];
random.nextBytes(bytes);
```

Note that the SecureRandom object does not need to be provided with an initialization value before being invoked.

10.1.2 MessageDigest

MessageDigest, which is part of the one-way hash algorithms, is used for generating checksums (also called “fingerprints”) to verify message integrity. Hash algorithms are also commonly used in the generation of digital signatures (see “Digital signatures” on page 263, for an explanation of digital signatures).

As with the other engine classes, MessageDigest objects are obtained using the getInstance() method:

```
MessageDigest sha1Digest = MessageDigest.getInstance(“SHA-1”);
```

A MessageDigest does not need specific initialization data and comes ready to hash the input message. All parts of the message that need to be hashed are passed through the object using the update() method:

```
String messagePart1 = “Secret 1”;
String messagePart2 = “Secret 2”;
sha1Digest.update(messagePart1.getBytes());
sha1Digest.update(messagePart2.getBytes());
etc..
```

After all the data has been passed through the object, the final result can be obtained by a call to digest():

```
byte[] theDigest = messageDigest.digest();
```

As long as the message parts are passed through the object in the same order, it does not make any difference how long each part is on the update(); the resulting digest value would end up being the same.

Note: The update() method provided by the MessageDigest, Signature, and Cipher classes operates on byte arrays. In the preceding example, the strings messagePart1 and messagePart2 are converted to bytes using the getBytes() method. This triggers an implicit codepage conversion to the systems default character set.

Use care to ensure that the expected codepages are in use. If needed, it is still possible to specify a specific codepage on the getBytes() method like:

```
String message = “Some text”
byte[] messageUTF8bytes = message.getBytes(“UTF-8“) // Convert to UTF-8

String message2 = new String(messageUTF8bytes, “UTF-8”); // Converting back
```

10.1.3 Signature

The Signature engine class is used for generating digital signatures. See “Digital signatures” on page 263 for an explanation of digital signatures.

The Signature objects are generated using the getInstance() method as follows:

```
Signature signature = Signature.getInstance("MD5withRSA");
```

In this case the signature is based on the MD5 message digest combined with the RSA asymmetric algorithm. Before use, the signature object is initialized with the signer's private key:

```
signature.initSign(privateRsaKey);
```

After this, the complete message to sign is passed through the object:

```
signature.update(messagePart1.getBytes());
signature.update(messagePart2.getBytes());
etc.
```

After all the required data has been passed though the signature object, the actual signature is created using:

```
byte[] messageSignature = signature.sign();
```

If somebody wants to verify the signature, it can be done using the same signature object initialized for verification, using the signer's public key:

```
signature.initVerify(publicRsaKey);

signature.update(messagePart1.getBytes());
signature.update(messagePart2.getBytes());
etc.
```

After passing the complete signed message through the signature object, the verify() method is called, along with the original signature, to test if the signature is verified:

```
boolean validSignature = signature.verify(messageSignature);
```

10.1.4 Cipher

The Cipher engine class is the class that implements encryption and decryption of data.

The Cipher object can be instantiated by just specifying the encryption algorithm on the getInstance() call, as follows:

```
Cipher aesCipher = Cipher.getInstance("AES")
```

However, additional parameters are often needed for specifying additional properties of the encryption algorithm. The following statement is the instantiation of an AES cipher with cipher block-chaining (CBC) and the padding scheme PKCS5Padding:

```
Cipher aesCipher = Cipher.getInstance("AES/CBC/PKCS5Padding", "IBMJCECCA");
```

Note: Even though a provider, in essence, supports a specific encryption algorithm, it might not support all possible combinations of such additional options.

After a cipher has been instantiated, it needs to be initialized for one of the following four modes:

ENCRYPT_MODE	This is used for encryption.
DECRYPT_MODE	This is used for decryption.
WRAP_MODE	This is used when encrypting other keys (for example, session keys with hybrid encryption).
UNWRAP_MODE	This is used for decrypting encrypted keys.

Depending on the kind of algorithm chosen, several variations of the initialization method may be used.

As an example, the initialization of the AES cipher can be:

```
aesCipher.init(Cipher.ENCRYPT_MODE, aesKey, ivSpec);
```

where `aesKey` is a symmetric key and `ivSpec` is a random initialization vector (see “Encryption modes” on page 261 for an explanation of the initialization vector).

After initialization, the cipher can be used for encryption or decryption using the update() and doFinal() methods:

```
byte[] encryptedData1 = aesCipher.update(dataToEncypt1);
byte[] encryptedData2 = aesCipher.update(dataToEncypt2);
...
byte[] encryptedDataN = aesCipher.doFinal(dataToEncyptN);
```

where doFinal() is used with the last block of data to be processed. This ensures that proper padding is performed, if required.

10.1.5 KeyGenerator

KeyGenerator is used for generating new random symmetric keys. Like all other engine classes, a KeyGenerator object is instantiated through the getInstance() method:

```
KeyGenerator generator = KeyGenerator.getInstance("AES", "IBMJCECCA");
```

When initializing the generator, the key size, in bits, is specified. It is also possible to provide a SecureRandom generator that should be used when generating keys:

```
generator.init(256, random);
```

After initialization, a key is generated using the generateKey() method:

```
SecretKey aesKey = generator.generateKey();
```

10.1.6 KeyPairGenerator

KeyPairGenerator is similar to KeyGenerator, but is used for asymmetric algorithm keys and therefore produces key pairs. A KeyPairGenerator is instantiated as follows:

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA","IBMJCECCA");
```

It is initialized in a similar way to KeyGenerator:

```
generator.initialize(2048, random);
```

A key pair is generated with:

```
KeyPair rsaKeyPair = generator.generateKeyPair();
```

Each key can then be obtained as a private or public key:

```
PublicKey publicRsaKey = rsaKeyPair.getPublic();
PrivateKey privateRsaKey = rsaKeyPair.getPrivate();
```

10.2 Cryptographic services invocation examples

This section provides examples that illustrate the basic use of common Java cryptographic engine classes. These examples refer to the IBMJCECCA and IBMJCE providers, assuming that they are properly positioned in the cryptographic service providers list. Refer to 9.4,

"Cryptographic service providers" on page 128, for an explanation of providers and the providers list. There is nothing else needed, beside proper specification of the providers, in order to run these examples.

All encryption keys used in these examples are generated when needed by the program, instead of using a key management scheme that would involve using keystores. For more information about keystores, refer to Chapter 11, "Java and key management on z/OS" on page 157.

The examples provided here cover the execution of the following cryptographic functions:

- Random number generation (SecureRandom)
- Message Digest
- Signature
- Symmetric encryption
- Asymmetric encryption

10.2.1 Random number generation

Example 10-1 shows how to generate true random-numbers by invoking the Crypto Express2 coprocessor through the IBMJCECCA provider.

The example first generates three true random integers, and then seeds a pseudo random generator, which is then used for generating three pseudo random integers.

Example 10-1 Random number generation

```
import java.security.SecureRandom;
import java.util.Random;

public class IBMSecureRandom1 {
  public static void main(String[] args) {
    try {
      // get a real random generator
      SecureRandom reallyRandom = SecureRandom.getInstance("IBMSecureRandom",
                                                           "IBMJCECCA");

      System.out.print("Some really random numbers: ");
      for (int i = 0; i < 3; i++) {
        System.out.print(reallyRandom.nextInt() + "  ");
      }
      System.out.println();

      // make a pseudo random generator seeded by the real random generator
      Random pseudoRandom = new Random(reallyRandom.nextLong());
      System.out.print("Some pseudo random numbers: ");
      for (int i = 0; i < 3; i++) {
        System.out.print(pseudoRandom.nextInt() + "  ");
      }
      System.out.println();

    } catch (Exception e) {
      System.err.println("Something went wrong ...");
      e.printStackTrace();
    }
  }
```

```
}
```

Example 10-1 on page 150 yields the following output:

```
Some really random numbers: -1057432571  -302680133  97550332
Some pseudo random numbers: 826547108  74718224  1007937281
```

10.2.2 Message Digest

In Example 10-2, various message digests (MD2, MD5, SHA-1, SHA-256, SHA-384, and SHA-512) are computed on the same input message, without specifying a specific provider.

The name of the provider that actually performs the hash operation is printed in the output, along with the actual hash value in hexadecimal.

Example 10-2 Message Digest generation

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class MessageDigest1 {

  public static void doDigest(String digestName, String message) {
    try {
      MessageDigest messageDigest = MessageDigest.getInstance(digestName);
      messageDigest.update(message.getBytes());
      byte[] digest = messageDigest.digest();

      System.out.println("Name: " + digestName + "\t" +
                         "Provider: " + messageDigest.getProvider() + "\t" +
                         "Digest len: " + digest.length + " bytes\t" +
                         "\n  Digest (hex): " + toHex(digest));

    } catch (NoSuchAlgorithmException e) {
      System.out.println(digestName + " not supported by any providers");
      e.printStackTrace();
    }
  }

  public static String toHex(byte[] bytes) {
    final String hexDigits = "0123456789ABCDEF";
    char[] res = new char[bytes.length * 2];

    for (int i = 0; i < bytes.length; i++) {
      res[(i * 2)    ] = hexDigits.charAt((bytes[i] & 0xff) / 16);
      res[(i * 2) + 1] = hexDigits.charAt((bytes[i] & 0xff) % 16);
    }
    return new String(res);
  }

  public static void main(String[] args) {
    String message = "This is the string to digest ...";

    System.out.println("Digesting the message: <" + message + ">");
```

```
    doDigest("MD2", message);
    doDigest("MD5", message);
    doDigest("SHA-1", message);
    doDigest("SHA-256", message);
    doDigest("SHA-384", message);
    doDigest("SHA-512", message);
  }
}
```

When running this example with a provider list with IBMJCECCA in the first position and IBMJCE in the second one, the program produced the following output.

Because the IBMJCECCA provider does not implement, as of the time of writing, the SHA-256, SHA-384, and SHA-512 algorithms, these are provided by the IBMJCE provider via a pure software implementation.

```
Digesting the message: <This is the string to digest ...>
Name: MD2       Provider: IBMJCECCA version 1.2 Digest len: 16 bytes
  Digest (hex): BACAAA6DAA0BC8BE463EB53E22A4F6EA
Name: MD5       Provider: IBMJCECCA version 1.2 Digest len: 16 bytes
  Digest (hex): 26B2276F5EC4D5CAC6410E78C8878D00
Name: SHA-1     Provider: IBMJCECCA version 1.2 Digest len: 20 bytes
  Digest (hex): 0B71DA7625AA8F69D02C3F86E03606C7FA5C56AF
Name: SHA-256   Provider: IBMJCE version 1.2    Digest len: 32 bytes
  Digest (hex):
05DE3A8E1BBA281B6BA826B87D8F5938A4985EB389090ECAABE8EBB9E3D82B
33
Name: SHA-384   Provider: IBMJCE version 1.2    Digest len: 48 bytes
  Digest (hex):
2687CBDAB89C9BC2CD785700D278D3C4C6202C0A6014D6D1C618F2CAE0979E
13828850A66FCD7AD356F15BD02A9775C6
Name: SHA-512   Provider: IBMJCE version 1.2    Digest len: 64 bytes
  Digest (hex): F387E7A79684CD085D50B437D11121C7EFD6B6B092C2C50AFBBF9D27FCF02F
E997F7BEFAC753992B4CA2AD480CFECAF665E5C319998A6D46D5F4F4E62DB3E60E
```

10.2.3 Signature

Example 10-3 demonstrates how to create a digital signature for a message, using a private key, and then shows how to verify the created signature by using the corresponding public key. Lastly, it shows a verification that fails because of an altered message.

Example 10-3 Digital signature generation.

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Signature;

public class Signature1 {
```

```
public static byte[] makeSignature(String message,
                                   PrivateKey privateRsaKey) throws Exception {

  Signature signature = Signature.getInstance("MD5withRSA", "IBMJCECCA");

  signature.initSign(privateRsaKey);
  signature.update(message.getBytes());
  byte[] messageSignature = signature.sign();

  return messageSignature;
}

public static boolean verifySignature(byte[] messageSignature, String message,
                                      PublicKey publicRsaKey) throws Exception {

  Signature signature = Signature.getInstance("MD5withRSA", "IBMJCECCA");

  signature.initVerify(publicRsaKey);
  signature.update(message.getBytes());
  boolean validSignature = signature.verify(messageSignature);

  if (validSignature) {
    System.out.println("Signature is valid with message: " + message);
  } else {
    System.out.println("Signature is NOT valid with message: " + message);
  }

  return validSignature;
}

public static void main(String[] args) {
  try {
    SecureRandom rnd = SecureRandom.getInstance("IBMSecureRandom", "IBMJCECCA");
    KeyPairGenerator generator =KeyPairGenerator.getInstance("RSA","IBMJCECCA");

    // make random RSA keys with KeyPairGenerator
    generator.initialize(2048, rnd);
    KeyPair rsaKeyPair = generator.generateKeyPair();
    PublicKey publicRsaKey = rsaKeyPair.getPublic();
    PrivateKey privateRsaKey = rsaKeyPair.getPrivate();

    String message = "Please send $100";
    String fakeMessage = "Please send $500";

    System.out.println("Making signature on message: " + message);
    byte[] messageSignature = makeSignature(message, privateRsaKey);
    System.out.println("Signature created. Signature length: " +
                   messageSignature.length + " bytes");

    System.out.println("Verifying signatures on messages");
    verifySignature(messageSignature, message, publicRsaKey);
    verifySignature(messageSignature, fakeMessage, publicRsaKey);

  } catch (Exception e) {
```

```
      System.err.println("Something went wrong ...");
      e.printStackTrace();
    }
  }
}
```

Example 10-3 on page 152 yields the following output:

```
Making signature on message: Please send $100

Signature created. Signature length: 256 bytes

Verifying signatures on messages

Signature is valid with message: Please send $100

Signature is NOT valid with message: Please send $500
```

10.2.4 Symmetric encryption

Example 10-4 demonstrates symmetric encryption using the AES encryption algorithm as implemented in the IBMJCECCA provider. The AES encryption key is dynamically generated.

The example makes use of the cipher block-chaining mode (CBC) and the PKCS5Padding padding mode. Notice that, because the CBC mode is used, an initialization vector is needed to initialize the chain.

Example 10-4 Symmetric encryption

```
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;

public class Symmetric1 {

  public static void main(String[] args) {
    try {
      SecureRandom rnd = SecureRandom.getInstance("IBMSecureRandom", "IBMJCECCA");
      KeyGenerator generator = KeyGenerator.getInstance("AES", "IBMJCECCA");
      Cipher aesCipher = Cipher.getInstance("AES/CBC/PKCS5Padding", "IBMJCECCA");

      // make random AES key with KeyGenerator
      generator.init(256, rnd);
      SecretKey aesKey = generator.generateKey();

      // generate a random initialization vectore
      byte[] ivData = new byte[16];
      rnd.nextBytes(ivData);
      IvParameterSpec ivSpec = new IvParameterSpec(ivData);

      String secretMessage = "This is the secret";
```

```
      System.out.println("Clear text: " + secretMessage);

      // init cipher for encrypt & do the encrypt
      aesCipher.init(Cipher.ENCRYPT_MODE, aesKey, ivSpec);
      byte[] encryptedData = aesCipher.doFinal(secretMessage.getBytes());

      // print encrypted data
      System.out.print("Encrypted: ");
      for (int i = 0; i < encryptedData.length; i++) {
        System.out.print(encryptedData[i] + " ");
      }
      System.out.println();

      // init cipher for decrypt & do the decrypt
      aesCipher.init(Cipher.DECRYPT_MODE, aesKey, ivSpec);
      String decryptedMessage = new String(aesCipher.doFinal(encryptedData));

      System.out.println("Decrypted: " + decryptedMessage);

    } catch (Exception e) {
      System.err.println("Something went wrong ...");
      e.printStackTrace();
    }
  }
}
```

The program produces the output shown below. The 18-byte original clear text message ends up being 32 bytes of encrypted data, because padding was implicitly performed to meet the AES 256 block length requirement.

```
Clear text: This is the secret

Encrypted: 31 57 83 -19 -94 71 33 53 60 52 -69 -19 -100 13 38 -61 -92 -100 6 -27

 -72 43 -55 82 -115 78 -106 71 -52 44 -61 107

Decrypted: This is the secret
```

10.2.5 Asymmetric encryption

Example 10-5 demonstrates an example of asymmetric encryption using the RSA algorithm. The RSA implementation of the IBMJCECCA provider is used.

Note: For this example to work successfully, you must install the policy files mentioned in 9.4.2, “Policy files” on page 130. If not installed, the example fails the decryption with an invalid key error. This is due to the large key size used.

Example 10-5 Asymmetric encryption

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
```

```
import java.security.SecureRandom;

import javax.crypto.Cipher;

public class Asymmetric1 {
  public static void main(String[] args) {
    try {
      SecureRandom rnd = SecureRandom.getInstance("IBMSecureRandom", "IBMJCECCA");
      KeyPairGenerator generator =KeyPairGenerator.getInstance("RSA","IBMJCECCA");
      Cipher rsaCipher = Cipher.getInstance("RSA", "IBMJCECCA");

      // make random RSA keys with KeyPairGenerator
      generator.initialize(2048, rnd);
      KeyPair rsaKeyPair = generator.generateKeyPair();
      PublicKey publicRsaKey = rsaKeyPair.getPublic();
      PrivateKey privateRsaKey = rsaKeyPair.getPrivate();

      String secretMessage = "This is the secret";
      System.out.println("Clear text: " + secretMessage);

      // init cipher for encrypt & do the encrypt
      rsaCipher.init(Cipher.ENCRYPT_MODE, publicRsaKey);
      byte[] encryptedData = rsaCipher.doFinal(secretMessage.getBytes());

      System.out.println("Encrypted len: " + encryptedData.length + " bytes");

      // init cipher for decrypt & do the decrypt
      rsaCipher.init(Cipher.DECRYPT_MODE, privateRsaKey);
      String decryptedMessage = new String(rsaCipher.doFinal(encryptedData));

      System.out.println("Decrypted: " + decryptedMessage);

    } catch (Exception e) {
      System.err.println("Something went wrong ...");
      e.printStackTrace();
    }
  }
}
```

Example 10-5 on page 155 produces the following output. Note that even through the clear text is only 18 byte long, the encrypted data ends up being 256 bytes in length. This is due to the RSA key size of 2048 bits that we use and which requires a data block size of 256 bytes.

```
Clear text: This is the secret

Encrypted len: 256 bytes

Decrypted: This is the secret
```

11

Java and key management on z/OS

This chapter describes the z/OS-specific Java keystores and explains how to manipulate the key material they contain. It also shows program excerpts that deal with Java keystores on z/OS by applying the principles of operation that are explained here.

Chapter 12, “Usage examples - using Java keystores on z/OS” on page 179, provides usage examples of *all* keystores available with z/OS Java.

© Copyright IBM Corp. 2008. All rights reserved.

11.1 Introduction

As previously mentioned, cryptographic systems provide support to achieve the following security services in a very robust and reliable manner:

- Authentication - the capability of verifying the trustworthiness of a presented identity.
- Data confidentiality - making the meaning of data appear only to authorized users.
- Data integrity - providing the capability of detecting unwanted modification of data.

Cryptographic key material is essential for any cryptographic system intended to be used as a support to these services.

As it stands today, many cryptographic algorithms in use in the industry are publicly known; that is, the processes they trigger during their execution are freely documented. However, how these processes are actually executed and chained depends on the value of the key provided, along with the data, as an input to the algorithm.

As a result the “strength” of a cryptography-based modification of data, when facing attempts to retrieve original data or misuse the modification, is mainly quantified by the “strength” of the cryptographic key used when performing the modification. The metric commonly used to quantify the strength of a key and algorithm is the mean time for the key to be correctly guessed in a brute force attack. In this context, mathematical principles and the use of random numbers are the foundation for ensuring the generation of strong keys.

Note that there might be several inhibitors to the generation and use of very strong cryptographic keys, such as laws and regulations or unacceptable duration of the cryptographic process time. The execution of cryptographic algorithms by specialized hardware devices provides most of the time that the required acceleration to help solving the execution time concerns, if any.

Because cryptographic system strength is quantifiable via the properties of the cryptographic keys, the real burden of ensuring the effectiveness of such a system's ability to provide these services becomes how to securely manage the cryptographic keys intended to be used.

z/OS has a long tradition of providing facilities to securely manage cryptographic keys in an enterprise. This section discusses how Java and z/OS can make Java-based cryptographic services and their exploiting applications leverage the z/OS strong key management services.

Before moving forward, however, it is important to highlight how keys help provide the services described in the introduction, and to review key management concepts.

11.1.1 Key usage

This section restates a few essential points regarding cryptographic keys. Refer to Appendix F, “Basics of cryptography” on page 259, for a more detailed overview of cryptography and keys.

A *cryptographic key* is a sequence of bytes of pre-determined length that serves as input to cryptographic functions. The keys today can be classified in two categories, depending on the algorithms they are used with:

- Asymmetric keys are pairs of mathematically bound keys that work in conjunction to perform an associated cryptographic function. For instance, one key can encrypt and the other can decrypt.

  The most important point regarding asymmetric keys is that if a cryptographic function such as encryption is performed using one key in the pair, then the other key in the pair is the only sequence of bytes that can be used as a key to perform the associated function (decryption, in this example). Generally, one key of the pair is designated as the public key and can be freely distributed. The other key of the pair is designated as the private key and is securely kept secret by the entity that owns the pair.
- Symmetric key refers to a single secret key that is used to perform both encryption and decryption. Because different parties are expected to be involved in the encryption and decryption processes, the key must be exchanged securely between these parties and also securely kept secret by its users.

Digital certificates are used in the context of asymmetric cryptography as a means to:

- Exchange the public key material they contain.
- Cryptographically bind information such as the owning entity name (referred to as a subject's or principal's name) or what the key material should be used for, to the specific public key material contained in the certificate.

Note that this cryptographic binding is actually the foundation on which trust is established between parties exchanging digital certificates.

Authentication

Authentication uses asymmetric keys with a simple encryption/decryption process or with what is called a digital signature process.

The entity seeking authentication digitally signs or encrypts data using its private key. The verifying entity uses the public key of the entity seeking authentication to perform the authentication.

Analogous to a person signing a legal document, digital signatures establish a form of non-repudiation by cryptographically binding the owner of an asymmetric key pair to the data being signed. In essence, the owner of the key pair cannot repudiate possession and access to data. In return, the signer can be assured that any modification to the data after they have been signed will be detected. The act of signing data requires using the private key of the signer(s). The signature can then be verified by others using the corresponding public key of the signer(s).

Securing the private key of a key pair is therefore extremely important. Because the only instance of the private key is available exclusively to its owner, it is the basis for authenticating the owner's identity.

Data confidentiality

Asymmetric encryption requires more processing power than symmetric encryption. As a result, cryptographic systems that must be scalable and must meet transaction rate requirements typically do not perform their encryption using asymmetric algorithms. They rely instead on symmetric algorithms to achieve their performance objectives when encrypting or decrypting data.

It is as important to secure symmetric keys as it is asymmetric private keys.

Data integrity

Cryptographic systems can be used as the underlying mechanism for verifying the integrity of data.

Data integrity can be achieved using the so-called “secure hash” and “modification detection code” algorithms. Note that these algorithms do not use secret keys.

Digital signature is another method, although more sophisticated, of insuring data integrity. Digital signature not only establishes the integrity of data, but also provides an unforgeable proof of who originated the signed data (to the extent that the signer’s private key is not compromised).

Note that detection of unauthorized modification of data, unintentional or malevolent, as a result of data integrity checking is not a preventative measure. Proper access control to the data remains crucial to achieving data integrity.

11.2 Introduction to Java key management

In previous chapters, the Java Cryptographic Extension and Java security packages are introduced and discussed in detail. The following sections highlight the main key materials, key management interfaces, and abstract classes that are available for Java applications.

An *interface*, in Java terminology, allows application programmers to write applications without requiring implementation-specific details about the services they call. An interface is actually a construct for defining methods that will provide these services. Similarly, abstract classes define methods that any extending objects must implement. Unlike interfaces, abstract classes encapsulate implementation details.

Because interfaces do not contain any implementation details and abstract classes have only some implementation details, they cannot be instantiated directly. Instead, classes must implement interfaces or extend abstract classes to provide a complete class definition that can be instantiated by applications.

11.2.1 Interfaces and abstract classes

Interfaces and abstract classes allow you to choose between different implementations of a given service without having to modify the main application code.

Following are some essential interfaces and abstract classes for keys and key management in Java, along with their definitions and a URL where they can be downloaded.

- Certificate (abstract class)
  - This is an abstract class that represents a cryptographic certificate. Cryptographic certificates are used to bind an entity referred to as a principal to public key material.
  - Sun Javadoc URL:

    `http://java.sun.com/javase/6/docs/api/java/security/cert/Certificate.html`

- Key
  - This is the top level interface for all keys. Many of the JCE objects that perform cryptographic function (for instance, Cipher) require key implementations as parameters. All Java key objects have at least three characteristics:
    - Algorithm
    - Encoded Form
    - Format
  - Sun Javadoc URL

    http://java.sun.com/javase/6/docs/api/java/security/Key.html
- KeySpec
  - A key specification contains information about a key. For example, an implementation of this interface is used to describe key material managed by ICSF, such as keys stored in the PKDS.
  - Sun Javadoc URL:

    http://java.sun.com/javase/6/docs/api/java/security/spec/KeySpec.html
- KeyFactory
  - A key factory translates a KeySpec into a key implementation and vice versa. The key implementation can then be used in other JCE classes.
  - Sun Javadoc URL:

    http://java.sun.com/javase/6/docs/api/java/security/KeyFactory.html
- PublicKey
  - This is the interface used by all public key implementations.
  - Sun Javadoc URL:

    http://java.sun.com/javase/6/docs/api/java/security/PublicKey.html
- PrivateKey
  - This is the interface used by all private key implementations.
  - Sun Javadoc URL:

    http://java.sun.com/javase/6/docs/api/java/security/PrivateKey.html
- AlgorithmParameterSpec
  - This is an algorithm specification that contains information about a particular cryptographic algorithm. For example, an implementation of this interface is used to describe different parameters to generate and store an asymmetric key pair that will be stored in the ICSF PKDS.
  - Sun Javadoc URL:

    http://java.sun.com/javase/6/docs/api/java/security/spec/AlgorithmParameterSpec.html

11.2.2 KeyStore-related classes

The SDK and JCE provide a few classes that can be used directly to work with keys and perform key management.

- RACFInputStream
  - This is an InputStream implementation that allows Java applications to access RACF key rings.
  - The Javadoc comes in the JAR file that can be downloaded from:

    `ftp://ftp.software.ibm.com/s390/java/jce4758/java6/jceccaDocs.jar`
- RACFOutputStream
  - This is an OutputStream implementation that allows Java applications to write certificates and key material to RACF key rings (this also comprises storing related keys into the PKDS).
  - The Javadoc comes in the JAR file that can be downloaded from:

    `ftp://ftp.software.ibm.com/s390/java/jce4758/java6/jceccaDocs.jar`
- KeyStore
  - This is the main class that represents a key repository. This class, in conjunction with certain IBM JCE providers such as IBMJCE and IBMJCECCA, along with the RACFInputStream and RACFOutputStream classes, allow Java applications to add or access certificates and key material stored in a RACF key ring.

    This class also allows certificates and key material stored in ICSF PKDS, CKDS, and TKDS, and in JCEKS and JKS keystores (that is, located in z/OS UNIX files) to be accessed or added by Java applications, depending upon the IBM provider that is used.
  - Sun Javadoc URL:

    `http://java.sun.com/javase/6/docs/api/java/security/KeyStore.html`
- KeyGenerator
  - This class generates symmetric keys. Depending on the JCE provider, KeyStore instance, and the AlgorithmParameterSpec instance, applications can use z/OS hardware for random number generation. Further, applications can generate secret material in the ICSF CKDS, Java JCEKS, or JKS keystore file located in the z/OS UNIX file system.
  - Sun Javadoc URL:

    `http://java.sun.com/javase/6/docs/api/javax/crypto/KeyGenerator.html`
- KeyPairGenerator
  - This class generates key pairs. Depending on the JCE provider, KeyStore instance, and the AlgorithmParameterSpec instance, applications can use the System z cryptographic coprocessor for random number generation. Further, applications can generate key material in the ICSF PKDS, TKDS, RACF, RACF and PKDS, or Java JCEKS or JKS keystore file located in the USS file system.
  - Sun Javadoc URL:

    `http://java.sun.com/javase/6/docs/api/java/security/KeyPairGenerator.html`
- KeyPair
  - This is a basic class which serves as a container for single asymmetric key; that is, the PublicKey and PrivateKey instances.

– Sun javadoc URL:

  http://java.sun.com/javase/6/docs/api/java/security/KeyPair.html

11.3 z/OS keystore details and provider requirements

z/OS offers different types of data repositories that can be used to store key material. This section describes these keystores and Java-specific details relevant to the use of each z/OS-specific data repository that backs up a keystore.

A Java keystore can be seen as a database that stores a collection of symmetric and asymmetric keys. Because keystore key entries hold sensitive cryptographic key information, they are kept encrypted. Typically, a Java keystore is implemented in a UNIX file. Key entries contain symmetric secret keys or asymmetric private keys and the certificates chains for the corresponding public keys.

On z/OS, digital certificates and the keys related to them can also be created and stored in the RACF database (or an equivalent product that provides certificates and key repository functions, and is accessible through the z/OS System Authorization Facility interface).

Certificates and keys are then made accessible to applications by the RACF administrator using RACF “key rings” owned by the applications. RACF key rings will also appear as keystores to Java applications.

Certificates stored in a RACF key ring are always accompanied by a public key and optionally accompanied by the corresponding private key. There are no symmetric keys kept in the RACF key rings.

11.3.1 IBMJCE supported keystores

IBMJCE, when running on z/OS, provides support for the keystore types described here.

JKS

JKS is the Sun original keystore proprietary implementation. This is a z/OS UNIX flat file, Keys in the JKS can be managed using the **keytool** Java utility. For further information about the keytool utility, refer to:

http://www-128.ibm.com/developerworks/java/jdk/security/142/secguides/keytoolDocs/KeyToolUserGuide-142.html)

JCEKS

JCEKS is the IBM implementation of the Sun keystore concept; that is, the class java.security.KeyStore class. It is implemented in a z/OS UNIX file with the strong protection of the private keys it contains, using Triple-DES password-based encryption.

Keys in the JCEKS are managed by the keytool utility, as well.

Users can “upgrade” their keystore of type JKS to the type JCEKS by changing the password of the private key entry in the keystore. Note that after the keystore has been upgraded this way, then it can be used via the IBMJCE provider only.

JCERACFKS

JCERACFKS is backed up by key rings in RACF. RACF key rings are used to aggregate RSA or DSA key pairs and certificates, so that they can be used by an application that owns the

key rings. If needed, a RACF key ring can also be shared between several authorized applications.

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 key ring is under the control of RACF resource profiles set up by the RACF administrator or delegated administrator.

The RACF RACDCERT command is the administrative interface used to manage the RACF key rings and the keys and certificates they contain. The certificates and key management in RACF, as well as the RACDCERT command syntax, are discussed in *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683 and *z/OS Security Server RACF Command Language Reference*, SA22-7687.

Starting with z/OS V1R9, the RACF key rings can also be managed by applications using the API provided by the RACF callable service R_datalib. Refer to z/OS Security Server RACF Callable Services, SA22-7691, for a description of the R_datalib RACF callable service.

In SDK 6 SR1 and in SDK 5 SR8, the keytool utility has been updated to exploit this facility so that it can write certificates and keys to a RACF key ring.

11.3.2 IBMJCECCA supported keystores

The IBMJCECCA provider relies on the use of z/OS ICSF to exploit the hardware cryptographic coprocessors available in System z. It therefore requires keys to be kept in ICSF constructs called "key tokens" either in z/OS UNIX files or in the ICSF PKDS, depending on the key store type. This section describes the IBMJCECCA-specific keystore types.

Important: When the ICSF key material is kept in the ICSF managed data sets PKDS or CKDS, their access can be controlled by RACF profiles in the CSFKEYS class of resources, regardless of whether these keys are accessed by existing applications or "through" Java keystores.

JCECCAKS

This IBM implementation of a Java keystore is primarily intended to store RSA asymmetric keys that are to be used with the cryptographic hardware coprocessors only. It can also be used to access symmetric keys kept in the ICSF CKDS (that is DES, Triple-DES and AES keys).

Keys in the JCECCAKS are managed using the **`hwkeytool`** utility. For more information about hwkeytool, refer to:

`ftp://ftp.software.ibm.com/s390/java/jce4758/hwkeytool.html`

The RSA asymmetric keys kept in the JCECCAKS can actually exist in three different forms, depending on the options taken when they have been generated. They can be retained keys, PKDS keys, or clear keys, as explained here:

- Retained keys - The private key of the "retained" key pair is kept inside the cryptographic coprocessor itself and is never exposed outside of the coprocessor.

  Important: IBM does not recommend using retained keys with z/OS ICSF.

- PKDS keys - The RSA key pair is kept in the ICSF Public Key Data Set, with the private key encrypted with the cryptographic coprocessor's asymmetric Master Key.

- Clear keys - The RSA key pair is packaged as an ICSF key token, without encryption of the private key by the cryptographic coprocessor's asymmetric Master Key.

When using JCECCAKS, the clear keys reside in the z/OS UNIX Java keystore, with the additional protection of a password. Retained keys and PKDS keys are only pointed at by a label which itself is kept in a keystore. All key information in the JCECCAKS is pointed at by the application using a keystore entry alias.

Note that only clear keys are returned to applications in order to be used. PKDS resident keys are directly picked up and sent to the coprocessor by ICSF itself. Retained keys, by definition, already reside inside the coprocessor.

Important: Clear key pairs may be exported in a PKCS#12 file. It is not possible to export PKDS or retained key pairs.

JCECCARACFKS

JCECCARACFKS is a variation of JCERACFKS, where the keys still appear as aggregated in a RACF key ring. However, the key ring does not contain the key material itself. Instead, it contains a PKDS label and the actual key is kept, encrypted with the coprocessor Master Key, in the PKDS.

11.3.3 IBMPKCS11Impl

The implementation of PKCS#11 tokens and how they can be exploited by z/OS Java applications is explained in 9.7, "The IBM providers - IBMPKCS11Impl" on page 138.

This section discusses the fact that using the IBMPKCS11Impl provider entails adding the IBMPKCS11Impl provider to the list of security providers in the java.security file and creating a IBMPKCS11Impl configuration file.

The IBMPKCS11Impl configuration file contains the following information:

- *name* - a string to append to IBMPKCS11Impl- to create the provider's instance name. For example, using the following example configuration file, a Java application will use the following String value to reference this provider:

  `IBMPKCS11Impl-ATMAPP`

- *library* - the HFS full path name of the PKCS#11 DLL library.

  For 31-bit mode, this will be /usr/lpp/pkcs11/lib/csnpcapi.so.

  For 64-bit mode, this will be /usr/lpp/pkcs11/lib/csnpca64.so.

- *description* - a string to describe the configuration
- *tokenLabel* - the name of the token (access is controlled via profiles in the SAF.CRYPTOZ class)

Here is an example of a IBMPKCS11Impl configuration file (assume that the name of the file is atmapp.cfg):

```
name = ATMAPP
library= /usr/lpp/pkcs11/lib/csnpcapi.so
description=ATM Application
tokenLabel=CUSTOMER1
```

The IBMPKCS11Impl provider supports a keystore type of PKCS11IMPLKS, where the PKCS#11 objects are stored in z/OS PKCS#11 tokens. The z/OS PKCS#11 tokens are

actually records in the ICSF TKDS. Access to these keystore objects is done through java.security.KeyStore class.

Important: When using the IBMPKCS11Impl provider, certificate objects, private key objects, secret key objects, and so on are stored as z/OS PKCS#11 tokens in the ICSF managed TKDS data set. Access to the tokens is controlled by RACF profiles in the CRYPTOZ class of resources.

In addition, ICSF will perform access control checks on underlying PKCS#11 callable services if the CSFSERV class is active in RACF.

11.3.4 z/OS keystore repositories and JCE provider list requirements

Table 11-1 summarizes what the actual repositories for keystores can be in z/OS, along with information about the use of these keystores and JCE providers setup details.

Note there is a column in the table that describes the JCE provider list requirements to leverage a given repository. As you remember, there are two ways to set up the provider list as explained in 9.4.1, “Installing and configuring providers” on page 128. You can set it up statically by using the java.security properties file, or you can set it up programmatically by using the addProvider method.

Table 11-1 Java keystore repositories

z/OS-specific repository	First Providers in Provider List (order must be as listed)	Certificate and keys details	Other
CKDS	com.ibm.crypto.hdwrCCA . provider.IBMJCECCA	No certificates. Symmetric keys only. Information necessary to access the key token in the CKDS is stored in a keystore file in z/OS UNIX. To retrieve or add symmetric keys in CKDS, the KeyStore type set in the KeyStore.getInstance method must be JCECCAKS.	A com.ibm.crypto.hdwrCCA.provider.KeyLabelKeySpec instance with the CKDS label is passed to KeyGenerator to generate the Key instance that can be set in a JCECCAKS keystore. By setting this generated Key instance in a JCECCAKS keystore, a key is generated and set in the CKDS.

z/OS-specific repository	First Providers in Provider List (order must be as listed)	Certificate and keys details	Other
PKDS	com.ibm.crypto.hdwrCCA . provider.IBMJCECCA	X.509 certificates. **Note:** The PKDS itself only contains RSA public and private keys. It does not contain certificates. The certificate itself and the information necessary to access the PKDS key material are stored in a keystore file in z/OS UNIX. KeyStore type set in the KeyStore.getInstance method must be JCECCAKS.	The private key material is kept encrypted with the cryptographic coprocessor. Master Key, and its clear value cannot be retrieved. The PKDS cannot be used if the private key has to be exported out of the keystore.
RETAINED KEY (kept inside the cryptographic coprocessor hardware)	com.ibm.crypto.hdwrCCA . provider.IBMJCECCA	Only X.509 certificates. **Note:** The coprocessor only contains the private key material. Certificate and information necessary to access the RETAINED key material are stored in a keystore file in z/OS UNIX. KeyStore type set in the KeyStore.getInstance method must be JCECCAKS.	The use of RETAINED keys is not recommended on z/OS.
CLEAR (key material, stored as ICSF key tokens)	com.ibm.crypto.hdwrCCA . provider.IBMJCECCA	Only X.509 certificates. **Note:** ICSF handles the key material itself, not the certificate. Certificate and information necessary to access the CLEAR keys are stored in a keystore file in z/OS UNIX. KeyStore type set in the KeyStore.getInstance method must be JCECCAKS.	The private key material is kept in its clear value (that is, not encrypted with the coprocessor Master Key). This is useful if the key pair must be exported. The PKCS#12 format is then supported for the exportable file.

z/OS-specific repository	First Providers in Provider List (order must be as listed)	Certificate and keys details	Other
TKDS (PKCS#11 support)	com.ibm.crypto.pkcs11impl. provider.IBMPKCS11Impl	Only X.509 certificates. The certificates are stored in the ICSF TKDS PKCS#11 tokens. No z/OS UNIX keystore file to maintain KeyStore type set in the KeyStore.getInstance method must be PKCS11IMPLKS.	The IBMPKCS11Impl configuration file path is specified with the provider in the list.
RACF	com.ibm.crypto.provider.IBMJCE	Only X.509 Certificates The certificates and keys are stored in the RACF database and connected to RACF key rings. No z/OS UNIX keystore file maintained. KeyStore type set in the KeyStore.getInstance method must be JCERACFKS.	You must set the InputStream for the KeyStore.load method to a com.ibm.crypto.hdwrCCA.provider.RACFInputStream instance. You must set the OutputStream for the KeyStore.save method to a com.ibm.crypto.hdwrCCA.provider.RACFOutputStream instance. .
RACF + PKDS	com.ibm.crypto.hdwrCCA. provider.IBMJCECCA	Only X.509 Certificates The certificates are stored in the RACF database and connected to RACF key rings. The related keys are kept in the PKDS with the private key encrypted with the coprocessor Master Key. No z/OS UNIX keystore file maintained. KeyStore type set in the KeyStore.getInstance method must be JCECCARACFKS.	You must set the InputStream for the KeyStore.load method to a com.ibm.crypto.hdwrCCA.provider.RACFInputStream instance. You must set the OutputStream for the KeyStore.save method to a com.ibm.crypto.hdwrCCA.provider.RACFOutputStream instance.

11.4 Java programming and key management

At this point, basic key management concepts have been discussed and the Java constructs available for key management and key usage from Java applications have been introduced. This section details how to leverage z/OS key management characteristics. It also provides examples of Java code performing key generation and keystore access.

11.4.1 Access control to z/OS resources

Prior to executing any Java application that uses the JCE and Java security packages to leverage z/OS key management facilities, authorizations must be given to access the resources that are involved.

APF Authorization

The Authorized Program Facility of z/OS is discussed in 1.5.1, “Authorized programs” on page 18. This section explains further the APF authorization requirements for executing Java applications on z/OS. As indicated here, a z/OS Java library requires APF authorization for zAAP dispatching to operate.

The ICSF load library hlq.SCSFMOD0 data set must be APF authorized and reside in the LINKLIST.

ICSF services RACF protection

For a comprehensive list of the ICSF services that are candidates for invocation when using the IBMJCECCA provider, refer to the following site:

http://www-03.ibm.com/servers/eserver/zseries/software/java/j6jcecca.html#using

If the installation decides to protect access to these services by using RACF profiles in the CSFSERV class (which is recommended by IBM), then the RACF userID of the Java application invoking IBMJCECCA must have the appropriate permission to access these services.

For a list of the ICSF services invoked when using the IBMPKCS11Impl provider, refer to the following site. The RACF userID of the Java application also must be given permission to access these services.

http://www-03.ibm.com/servers/eserver/zseries/software/java/j6pkcs11implgd.html#j6config

11.4.2 UNIX System Services file system permissions

z/OS UNIX files permission bits can be used to control access to Java keystore files. As you recall, JCEKS, JKS, and JCECCAKS keystores require z/OS UNIX files to store key material and certificates. z/OS UNIX provides the standard UNIX file system permission bits; that is, three octets with the read, write and execute access control bits each, for:

1. The owner of the file
2. The owning group of the file
3. To all other users

For each octet, the first bit of the octet is set to 1 if read access is granted. Otherwise, it is set to zero (0). The second bit of the octet is set to 1 if write access is granted. Otherwise, it is set to (0). Finally, the last bit of the octet is set to 1 if execute access is granted,. Otherwise, it is set to (0).

When the execute bit is set to 1, the file is expected to contain an executable shell script or compiled program which can then be invoked. Keystore files should never have the execute bit set. Here is an example of allowing a read and write access to a keystore file for its owner and its owning group, and disallowing any access to all other users.

```
chmod 600 ibmkeystore.jce
```

11.4.3 Access control to RACF key rings and certificates

The access control for RACF key rings and certificates addresses both administrative accesses and application-required accesses.

Administrative accesses

The RACF RACDCERT command is the primary administrative interface used to manage the key rings. The command is documented in *z/OS Security Server RACF Command Language Reference*, SA22-7687, along with the permissions it requires. Refer to Chapter 12, “Usage examples - using Java keystores on z/OS” on page 179, for usage examples.

Accesses to the many functions available with the RACDCERT command are controlled with profiles in the FACILITY class of resources. The profiles names are of the form:

```
IRR.DIGTCERT.<function>
```

where <*function*> is the function keyword used with the RACDCERT command. Examples of such profiles are:

```
IRR.DIGTCERT.ADD
IRR.DIGTCERT.ALTER
IRR.DIGTCERT.DELETE
IRR.DIGTCERT.LIST
IRR.DIGTCERT.ADDRING
etc....
```

Programmatic access

Programmatic access to key rings and certificates is also protected by the IRR.DIGTCERT.*<function>* profiles in the FACILITY class, where *<function>* addresses the function invoked by the R_datalib RACF callable service, which is actually the API used for programmatic access to RACF key rings and certificates. Refer to *z/OS Security Server RACF Callable Services*, SA22-7691, for a description of the R_datalib RACF callable service.

Writable key rings and key rings granular access control

Programmatic access to RACF key rings and certificates was read-only previous to the availability of z/OS V1R9, meaning that they could not be programmatically managed by application. Only the RACDCERT RACF administration command could be used to manage the key and certificate material in the key rings.

z/OS V1R9 extends the functions provided by the R_datalib service to update functions to the key rings (“writable key rings”), which therefore become manageable by program. In SDK 6 SR1 and SDK 5 SR8, the keytool utility was updated so it can also write certificates and keys to a RACF key ring.

z/OSV1R9 also introduces more granularity to the control of key ring accesses by programs, with optional profiles in the RDATALIB class of RACF resources. At z/OS V1R9, the installation can choose using the RACF IRR.DIGTCERT.<function> profiles in the FACILITY class for global access control to the key rings. This is illustrated in the following setup, where <*KROWNER*> designates the application user ID that owns the key ring:

```
RDEFINE FACILITY IRR.DIGTCERT.LISTRING UACC(NONE)

RDEFINE FACILITY IRR.DIGTCERT.LIST UACC(NONE)

PERMIT IRR.DIGTCERT.LISTRING CLASS(FACILITY) ID(<KROWNER>) ACC(READ)
```

```
PERMIT IRR.DIGTCERT.LIST CLASS(FACILITY) ID(<KROWNER>) ACC(READ)
```

Alternatively, the installation can elect to use the more granular access control (“ring-specific access control”) with profiles in the RDATALIB class with a name of the form:

<KROWNER>.<KEYRING>.<function>

where *<KROWNER>* is the RACF user ID of the owner of the key ring, *<KEYRING>* is the name of the RACF key ring, and <function> is the type of access to the key ring.

The following setup example uses the RDATALIB profiles for giving read (function LST) and write (function UPD) access to key rings to the *<KRACCESSID>* userID.

```
RDEFINE RDATALIB <KRWONER>.<KEYRING>.UPD UACC(NONE)

RDEFINE RDATALIB <KRWONER>.<KEYRING>.LST UACC(NONE)

PERMIT <KRWONER>.<KEYRING>.LST CLASS(RDATALIB) ID(<KRACCESSID>) ACCESS(CONTROL)

PERMIT <KRWONER>.<KEYRING>.UPD CLASS(RDATALIB) ID(<KRACCESSID>) ACCESS(CONTROL)
```

11.5 A word about trust

Trust is another security-related property that needs to be managed, especially when working with digital certificates. When exchanging certificates, the digital signature that applies to the certificates allows recipients to determine whether the certificates can be trusted, and thus whether the key material they contain may be safely used by the cryptographic system.

One example of a cryptographic system that relies on trusted certificates is the SSL/TLS protocol. Generally speaking, SSL/TLS relies on X.509 V3 certificates and the digital signature they contain. A client connecting to a server using SSL or TLS receives the server's own X.509 certificate and verifies its signature using the corresponding trusted Certificate Authority certificate. If the signature is verified, then the trust property granted to the Certificate Authority (CA) certificate also applies to the server’s certificate, and the information it contains can then be trusted by the client to proceed with identification and authentication of the server.

Note that the server certificate’s Certificate Authority can be an intermediate authority and part of a certificate authorities chain, meaning that the server’s own certificate is signed by an intermediate CA, of which certificate is signed by an upper level CA, and so on until the top (or “root” CA). In that case, the SSL/TLS protocol makes it possible for the client to recursively acquire all the certificates that make up the certificate authorities chain, and verify each signature until the root Certificate Authority trusted certificate is received. In that case, trust applies transitively along the chain from the trusted root CA down to the server’s own certificate.

RACF provides ways of managing the level of trust granted to certificates. If a certificate that has not been signed by a known and trusted Certificate Authority is imported into the RACF database, it will be marked with NOTRUST by RACF and cannot be of any use to z/OS applications. It will require an intervention of the RACF administrator to switch the certificate from a NOTRUST status into TRUST status.

For Java applications using z/OS UNIX file-based keystores (JCEKS, JKS, and JCECCAKS), file system permissions can be used to protect keystore files. An approach for controlling trust could be to deem certain keystore files as trusted and use the permission bits accordingly to

strictly control updates to that keystore, so that the trustworthiness of their content is preserved.

The Java Secure Socket Extension (IBMJSSE2) providers use a trusted keystore concept to provide SSL/TLS services. Refer to the following site for more information about the way JSSE manages trust:

http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

11.6 Java program examples

Examples of exploitation of various z/OS keystores, along with examples of the required preliminary setup, are given in Chapter 12, “Usage examples - using Java keystores on z/OS” on page 179.

This section provides miscellaneous excerpts of Java code that address various key management-relevant operations.

11.6.1 Dynamic Provider List example

Example 11-1 contains code to dynamically set the providers list. This allows application developers to modify the providers list dynamically, without requiring an update to the Java installation file, java.security.

Example 11-1 Dynamic setting of providers list

```
try {
   String  provider = "com.ibm.crypto.hdwrCCA.provider.IBMJCECCA";
    Class providerClass = null;
    providerClass = Class.forName(provider);
    Provider p = (Provider)providerClass.getConstructor(new
Class[]{}).newInstance(new Object[]{});

    Security.removeProvider(provider); //ensure provider is moved to the top

    Security.insertProviderAt(p,1);
}catch (Throwable e){
    ....
}
```

11.6.2 Generate X.509 certificate

For storing asymmetric keys in a Java keystore, a certificate must be created. The public key is set in the certificate before adding to the keystore. The private key, if available, is stored with the certificate. Note that a private key is not required to accompany a certificate as long as the certificate is used with operations involving only the public key.

Example 11-2 is a program “snippet” where we assume that the private key is available and that the certificate is self-signed. Refer to 11.5, “A word about trust” on page 171 for more information about how digital signatures and certificates are used to establish trust. A self-signed certificate, as the name implies, is signed by the owning entity of the certificate itself. A self-signed certificate does not go with a certificate chain that could be used to establish trust,. Instead, it has to be trusted as is by the user.

Example 11-2 Generate an X.509 V3 certificate

```
X500Signer    issuer;
   X509CertImpl  cert;
   Date          lastDate;
   X500Name subjectDN,signerDN;

   //Key pair to encapsulate/self sign cert
   KeyPair pair = …;

   // Number of Days Valid
   long      validity = …;
   try {
      // SUBJECT of this certificate
     subjectDN = new X500Name("cn=self sign,ou=ITSO,o=IBM,c=US");

   //Certificate's SIGNER DN - since self signed, use subjectDN
      signerDN = subjectDN;

   //Issuer is the signer
      issuer = new
X500Signer(Signature.getInstance("SHA1withRSA","IBMJCECCA"),signerDN);

   //Establish validity period
      lastDate = new Date ();

//1 second = 1 000 milliseconds
      long todaysDate = lastDate.getTime() / 1000; //todays date in seconds
       // 1 day = 86400 seconds
      long validTime = (long)(validity * 86400) ; //valid time in seconds

      long totalTime = todaysDate + validTime; //today + days valid
            lastDate.setTime (totalTime * 1000); //Convert back to milliseconds

     CertificateValidity interval =
            new CertificateValidity(firstDate,lastDate);

     X509CertInfo info = new X509CertInfo();
            // Add all mandatory attributes
            // Note here that V1 = 0, V2 = 1, V3 = 2
     info.set(X509CertInfo.VERSION,
              new CertificateVersion(CertificateVersion.V3));

     info.set(X509CertInfo.SERIAL_NUMBER,
              new CertificateSerialNumber(
                     (int)(firstDate.getTime()/1000)));

     AlgorithmId algID = issuer.getAlgorithmId();
     info.set(X509CertInfo.ALGORITHM_ID,
              new CertificateAlgorithmId(algID));

     info.set(X509CertInfo.SUBJECT,
               new CertificateSubjectName(subjectDN));

     info.set(X509CertInfo.KEY, new CertificateX509Key(pair.getPublicKey()));
```

```
     info.set(X509CertInfo.VALIDITY, interval);

     info.set(X509CertInfo.ISSUER,
                     new CertificateIssuerName(issuer.getSigner()));


     cert = new X509CertImpl(info);

     cert.sign(pair.getPrivateKey(), "SHA1withRSA");
 } catch (Exception e) {
          ….
}
```

11.6.3 Reusing an existing RSA key in the ICSF PKDS

Example 11-3 demonstrates how you could generate a Java Object that represents an already existing RSA key that resides in the ICSF PKDS. The object appears to be stored in a keystore and can be retrieved using the keystore_alias value.

This Java Object can be used with the Cipher and Signature classes and the IBMJCECCA provider because it actually contains keys residing in the PKDS. Note that Example 11-3 refers to Example 11-2 on page 173 for the generation of an X.509 V3 certificate that is required to be stored in the keystore entry.

Example 11-3 Reusing an RSA key

```
// PKDS label of key to reuse
      String keyLabel = …;

     // keystore alias that can be reused to retrieve the key object
   String keystore_alias = …;

    //The password to the JCECCAKS keystore file where the key will be inserted.
    String keystore_password = …;

   KeyLabelKeySpec keyLabelKeySpec =
   new KeyLabelKeySpec(keyLabel);

      //KeyFactory used to generate Key objects
   KeyFactory kf = KeyFactory.getInstance("RSA","IBMJCECCA");

      //KeyPair object used to store in keystore
   KeyPair pair = new KeyPair(kf.generatePublic(keyLabelKeySpec),
                                kf.generatePrivate(keyLabelKeySpec));

   //Load the keystore and insert the key generated above into the keystore.
    KeyStore ks = KeyStore.getInstance(keystore_type,"IBMJCECCA");
    FileInputStream fis = null;
    try {
       fis = new FileInputStream(keystore_file);
     } catch ( java.io.FileNotFoundException e ) {
      //It is ok if the file does not exist. We will create the keystore.
     }
```

```
    ks.load(fis, keystore_password.toCharArray());

    // See "GENERATE X.509 CERTIFICATE" example for generating an X.509 cert
    Certificate cert = …;

    ks.setKeyEntry(keystore_alias, pair.getPrivateKey(),
                   keystore_password.toCharArray(), cert );

  //Save the keystore.
  ks.store(new FileOutputStream(keystore_file), keystore_password.toCharArray());
```

11.6.4 Generate an AES CKDS key

Example 11-4 demonstrates how you would generate an AES symmetric key to be stored in the CKDS. Note that a Java keystore file (type JCECCAKS) is created in the z/OS UNIX file system to properly interface with the Java application.

Example 11-4 Generate an AES CKDS key

```
// The keystore type.
    String keystore_type = new String("JCECCAKS");

    //The key type inside of the CKDS.
    //Valid values are DESede and AES.
    String algorithm = new String("AES");

    //The JCECCAKS keystore file name.
    String keystore_file = …;

    //The password to the JCECCAKS keystore file where the key will be inserted.
    String keystore_password = …;

    //The keystore alias that the key entry is inserted with in the JCECCAKS
keystore file.
    // It must be 21 characters
    // The first 3 characters must be alphabetic
    // The remaining characters must be hexadecimal.
    String keystore_alias = …;

    //The CKDS label to be associated with this key entry in the JCECCAKS keystore
file.
    //The label cannot be larger than 64 characters.
    String ckds_label = …;

   //Generate the secret key object corresponding to the label passed in.
   SecretKeyFactory myKeyFactory = SecretKeyFactory.getInstance(algorithm,
"IBMJCECCA");
   KeyLabelKeySpec spec = new KeyLabelKeySpec(ckds_label);
   SecretKey key = myKeyFactory.generateSecret(spec);


  //Load the keystore and insert the key generated above into the keystore.
  KeyStore ks = KeyStore.getInstance(keystore_type,"IBMJCECCA");
```

```
  FileInputStream fis = null;
  try {
    fis = new FileInputStream(keystore_file);
  } catch ( java.io.FileNotFoundException e ) {
      //It is ok if the file does not exist. We will create the keystore.
  }
  ks.load(fis, keystore_password.toCharArray());
  ks.setKeyEntry(keystore_alias, key, keystore_password.toCharArray(),      null);

  //Save the keystore.
  ks.store(new FileOutputStream(keystore_file), keystore_password.toCharArray());
```

11.6.5 Generate a RACF RSA key pair with the private key in the PKDS

Example 11-5 demonstrates how you would generate an RSA key pair with the certificate stored in the RACF database and the private key stored in the PKDS. This is equivalent to using the RACDCERT command with the ICSF keyword.

Note that the code refers to Example 11-2 on page 173 for generating an X.509 V3 certificate which, in this case, is kept in the RACF database. The RACFOutputStream and RACFInputStream used in this example are coming with the package com.ibm.crypto.hdwrCCA.provider.

When generating certificates and keys stored in RACF, only the RACFOutputStream and RACFInputStream classes from the com.ibm.crypto.provider package should be used.

Example 11-5 RACF and PKDS key generation

```
// RACF User ID to use to load/store keyring
       String racf_user_id = …;

    // The keystore type.
    String keystore_type = new String("JCERACFKS");


    //The RACF key ring name.
    String keystore_name = …;

    //The password to the RACF key ring (can be anything, but key password must
match keystore password.
    String keystore_password = …;

    //The keystore alias the key entry is inserted with in the JCECCAKS keystore
file.
    String keystore_alias = …;

   //Generate Hardware RSA key
   byte type = KeyHWAttributeValues.PKDS;

   byte usage = KeyHWAttributeValues.KEYMANAGEMENT;

   RSAKeyParameterSpec spec;

   spec = new RSAKeyParameterSpec(keySize,type,usage);
```

```
   KeyPairGenerator  generator = new RSAKeyPairGenerator();
   generator.initialize(spec);
   pair = generator.genKeyPair();

  //Load the keystore and insert the key generated above into the keystore.
  KeyStore ks = KeyStore.getInstance(keystore_type,"IBMJCECCA");
  RACFInputStream fis = null;
  fis = new
RACFInputStream(racf_user_id,keystore_file,keystore_password.toCharArray());
  ks.load(fis, keystore_password.toCharArray());


     // See "GENERATE X.509 CERTIFICATE" example for generating an X.509 cert
     Certificate cert = …;

  ks.setKeyEntry(keystore_alias, key, keystore_password.toCharArray(),      cert);

  RACFOutputStream fout = null;
  fout = new RACFOutputStream
(racf_user_id,keystore_file,keystore_password.toCharArray());

  //Save the keystore, i.e. set key ring in RACF.
  ks.store(fout, keystore_password.toCharArray());
```

12

Usage examples - using Java keystores on z/OS

This chapter provides Java examples that use the various keystores supported on z/OS. Each example includes the initial setup needed for running the sample programs.

Examples for the following keystores are provided:

- JCEKS
- JCECCAKS
- JCERACFKS
- JCECCARACFKS
- PKCS11IMPLKS

The chapter also provides examples that operate directly on keys residing in the ICSF CKDS and PKDS using their key labels.

© Copyright IBM Corp. 2008. All rights reserved.

12.1 JCEKS

Example 12-1 shows how to read keys out of a JCEKS keystore. The keystore were created with the keytool utility, as shown here. When using the JCEKS, both the certificate and the private key are stored in the file referred to by the **keystore** option.

Example 12-1 Generating an RSA key pair in a JCEKS with keytool

```
keytool -genkeypair \
        -alias AliceRSA \
        -dname "CN=IBM" \
        -keystore testkeys.jcks \
        -provider IBMJCE \
        -keyalg RSA \
        -keysize 2048 \
        -keypass passw0rd \
        -storepass passw0rd \
        -storetype JCEKS \
        -validity 999
```

Example 12-2 reads the public and private key generated in Example 12-1. Then the keys are used for a simple asymmetric encryption followed by decryption.

Example 12-2 RSA encryption and decryption using a JCEKS key pair

```
import java.io.FileInputStream;
import java.io.InputStream;
import java.security.KeyPair;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.cert.Certificate;

import javax.crypto.Cipher;

public class JceksExample1 {
  public KeyPair readKeys() throws Exception {
    try {
      KeyStore jceks = KeyStore.getInstance("JCEKS", "IBMJCE");
      InputStream is = new FileInputStream("/u/ebbemp/ks/testkeys.jcks");
      jceks.load(is, "passw0rd".toCharArray());

      Certificate aliceCertificate = jceks.getCertificate("AliceRSA");
      PublicKey alicePublicKey = aliceCertificate.getPublicKey();
      PrivateKey alicePrivateKey = (PrivateKey) jceks.getKey("AliceRSA",
                                                    "passw0rd".toCharArray());

      KeyPair keyPair = new KeyPair(alicePublicKey, alicePrivateKey);
      return keyPair;
    } catch (Exception e) {
      System.err.println("Could not read the keystore");
      throw new Exception(e);
    }
  }
```

```
  public byte[] encrypt(String clearText, PublicKey publicKey) {
    try {
      Cipher rsaCipher = Cipher.getInstance("RSA", "IBMJCE");
      rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);
      byte[] encryptedData = rsaCipher.doFinal(clearText.getBytes());
      return encryptedData;
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  public String decrypt(byte[] encryptedData, PrivateKey privateKey) {
    try {
      Cipher rsaCipher = Cipher.getInstance("RSA", "IBMJCE");
      rsaCipher.init(Cipher.DECRYPT_MODE, privateKey);
      byte[] decryptedMessageBytes = rsaCipher.doFinal(encryptedData);
      String decryptedMessage = new String(decryptedMessageBytes);
      return decryptedMessage;
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  public static void main(String[] args) {
    JceksExample1 example = new JceksExample1();
    String secretMessage = "Hi Alice, please don't tell anyone, but ..";

    try {
      System.out.println("Reading keys from Keystore");
      KeyPair keys = example.readKeys();
      System.out.println("Keys read");

      System.out.println("Encrypting: " + secretMessage);
      byte[] encrypted = example.encrypt(secretMessage, keys.getPublic());
      System.out.println("Message encrypted");

      System.out.println("Decrypting:");
      String decrypted = example.decrypt(encrypted, keys.getPrivate());
      System.out.println("Message decrypted: " + decrypted);
    } catch (Exception e) {
      System.err.println("Something went wrong ...");
      e.printStackTrace();
    }
  }
}
```

This program issues the messages shown in Figure 12-1 on page 182.

```
Reading keys from Keystore
Keys read
Encrypting: Hi Alice, please don't tell anyone, but ..
Message encrypted
Decrypting:
Message decrypted: Hi Alice, please don't tell anyone, but ..
```

Figure 12-1 Messages issued by the JCEKS use example

12.2 JCECCAKS

This example uses the **hwkeytool** to generate a JCECCAKS keystore with two RSA key pairs. The first key pair is created with the **hardwaretype** option CLEAR, as shown in Example 12-3. This means that the key pair will be stored in clear in an ICSF key token. However, the certificate itself is stored in the file referred to by the **keystore** option.

Example 12-3 Generating a clear key pair with the hwkeytool utility

```
hwkeytool -genkeypair \
          -alias AliceRSAclear \
          -dname "CN=IBM" \
          -keystore testkeys.cca \
          -provider IBMJCECCA \
          -keyalg RSA \
          -keysize 2048 \
          -storetype JCECCAKS \
          -keypass abc123 \
          -storepass abc123 \
          -hardwaretype CLEAR
```

The second RSA key pair is generated with the **hardwaretype PKDS** option, as shown in Example 12-4. The private key is encrypted with the hardware coprocessor asymmetric Master Key and is kept in the PKDS.

Note: hwkeytool allows the specification of a key label for the key kept in the PKDS. If this option is not used, **hwkeytool** generates a random value given as a key label to the key in the PKDS.

Example 12-4 Generating a PKDS RSA key pair with the hwkeytool utility

```
hwkeytool -genkeypair \
          -alias AliceRSApkds \
          -dname "CN=IBM" \
          -keystore testkeys.cca \
          -provider IBMJCECCA \
          -keyalg RSA \
          -keysize 2048 \
          -storetype JCECCAKS \
          -keypass abc123 \
          -storepass abc123 \
          -hardwaretype PKDS \
          -keylabel ALICE.RSA.TEST1
```

The program in Example 12-5 is run next, which performs the RSA encryption and decryption sequence twice, the first time using the RSA clear key pair and the second time using the PKDS key pair.

Example 12-5 RSA encryption and decryption using the JCECCAKS clear and PKDS key pairs

```
import java.io.FileInputStream;
import java.io.InputStream;
import java.security.KeyPair;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.cert.Certificate;

import javax.crypto.Cipher;

public class JceccaksExample1 {
  private KeyPair readKeys(String keyLabel) throws Exception {
    try {
      KeyStore jceccaks = KeyStore.getInstance("JCECCAKS", "IBMJCECCA");
      InputStream is = new FileInputStream("/u/ebbemp/ks/testkeys.cca");
      jceccaks.load(is, "abc123".toCharArray());

      Certificate aliceCertificate = jceccaks.getCertificate(keyLabel);
      PublicKey alicePublicKey = aliceCertificate.getPublicKey();
      PrivateKey alicePrivateKey = (PrivateKey) jceccaks.getKey(keyLabel,
                                                    "abc123".toCharArray());

      KeyPair keyPair = new KeyPair(alicePublicKey, alicePrivateKey);
      return keyPair;
    } catch (Exception e) {
      System.err.println("Could not read the keystore");
      throw new Exception(e);
    }
  }

  private byte[] encrypt(String clearText, PublicKey publicKey) {
    try {
      Cipher rsaCipher = Cipher.getInstance("RSA", "IBMJCECCA");
      rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);
      byte[] encryptedData = rsaCipher.doFinal(clearText.getBytes());
      return encryptedData;
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  private String decrypt(byte[] encryptedData, PrivateKey privateKey) {
    try {
      Cipher rsaCipher = Cipher.getInstance("RSA", "IBMJCECCA");
      rsaCipher.init(Cipher.DECRYPT_MODE, privateKey);
      byte[] decryptedMessageBytes = rsaCipher.doFinal(encryptedData);
      String decryptedMessage = new String(decryptedMessageBytes);
      return decryptedMessage;
    } catch (Exception e) {
```

```
      e.printStackTrace();
      return null;
    }
  }

  public void encryptDecryptTest(String keyLabel) {
    String secretMessage = "Hi Alice, please don't tell anyone, but ..";

    try {
      System.out.println("Reading " + keyLabel + " keys from Keystore");
      KeyPair keys = readKeys(keyLabel);
      System.out.println("Keys read");

      System.out.println("Encrypting: " + secretMessage);
      byte[] encrypted = encrypt(secretMessage, keys.getPublic());
      System.out.println("Message encrypted");

      System.out.println("Decrypting:");
      String decrypted = decrypt(encrypted, keys.getPrivate());
      System.out.println("Message decrypted: " + decrypted);
    } catch (Exception e) {
      System.err.println("Something went wrong ...");
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    JceccaksExample1 example = new JceccaksExample1();
    System.out.println("Clear key test");
    example.encryptDecryptTest("AliceRSAclear");

    System.out.println();
    System.out.println("PKDS key test");
    example.encryptDecryptTest("AliceRSApkds");
  }
}
```

When executing correctly, the program issues the messages shown in Figure 12-2 on page 185.

```
Clear key test
Reading AliceRSAclear keys from Keystore
Keys read
Encrypting: Hi Alice, please don't tell anyone, but ..
Message encrypted
Decrypting:
Message decrypted: Hi Alice, please don't tell anyone, but ..

PKDS key test
Reading AliceRSApkds keys from Keystore
Keys read
Encrypting: Hi Alice, please don't tell anyone, but ..
Message encrypted
Decrypting:
Message decrypted: Hi Alice, please don't tell anyone, but ..
```

Figure 12-2 Messages issued by the JCECCAKS use example

12.3 JCERACFKS

This example uses a JCERACFKS keystore. The content of the keystore is generated using the RACDCERT RACF command. The example generates an RSA key pair and a self-signed certificate that are connected to a new key ring in RACF with the USAGE(PERSONAL) option of the RACDCERT CONNECT command. Using this option allows you to also keep the private key that corresponds to the certificate in the RACF database.

Figure 12-3 shows the RACF RACDCERT commands issued to perform these tasks:

- Create the key pair and the self-signed certificate with a label AliceRSA1; this is the RACDCERT GENCERT command.
- Create a key ring with a label ALICE_JCERACFKS, owned by the RACF userID EBBEMP; this is the RACDCERT ADDRING command.
- Connect the certificate and the key pair to the key ring with the USAGE(PERSONAL) option; this is the RACDCERT CONNECT command.

```
RACDCERT ID(EBBEMP) GENCERT
SUBJECTSDN(CN('Alice Self Signed Cert JCERACFKS'))
SIZE(1024) WITHLABEL('AliceRSA1')

RACDCERT ID(EBBEMP) ADDRING(ALICE_JCERACFKS)

RACDCERT ID(EBBEMP) CONNECT(ID(EBBEMP) LABEL('AliceRSA1')
RING(ALICE_JCERACFKS) USAGE(PERSONAL))
```

Figure 12-3 Setup of the JCERACFKS example

To be able to access the key pair and certificate created, users need to have at least READ access to the IRR.DIGTCERT.LIST RACF resource in the RACF FACILITY class of resources, as shown in Figure 12-4 on page 186.

```
PE IRR.DIGTCERT.LIST CL(FACILITY) ID(EBBEMP) ACC(READ)
SETR RACLIST(FACILITY) REF
```

Figure 12-4 Granting RACF access to the key pair and certificate

The example program is shown in Example 12-6. Note that this program uses com.ibm.crypto.provider.RACFInputStream. This is an InputStream implementation that allows Java applications to access RACF key rings.

Example 12-6 RSA encryption and decryption using a JCERACFKS key pair

```
import java.io.InputStream;
import java.security.KeyPair;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.cert.Certificate;

import javax.crypto.Cipher;

import com.ibm.crypto.provider.RACFInputStream;

public class JceracfksExample1 {
  public KeyPair readKeys() throws Exception {
    try {
      KeyStore jceracfks = KeyStore.getInstance("JCERACFKS", "IBMJCE");
      InputStream is = new RACFInputStream("EBBEMP", "ALICE_JCERACFKS", null);
      jceracfks.load(is, null); // no password

      Certificate aliceCertificate = jceracfks.getCertificate("AliceRSA1");
      PublicKey alicePublicKey = aliceCertificate.getPublicKey();

      PrivateKey alicePrivateKey =(PrivateKey) jceracfks.getKey("AliceRSA1",null);

      KeyPair keyPair = new KeyPair(alicePublicKey, alicePrivateKey);
      return keyPair;
    } catch (Exception e) {
      System.err.println("Could not read the keystore");
      throw new Exception(e);
    }
  }

  public byte[] encrypt(String clearText, PublicKey publicKey) {
    try {
      Cipher rsaCipher = Cipher.getInstance("RSA");
      rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);
      byte[] encryptedData = rsaCipher.doFinal(clearText.getBytes());
      return encryptedData;
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  public String decrypt(byte[] encryptedData, PrivateKey privateKey) {
```

```
    try {
      Cipher rsaCipher = Cipher.getInstance("RSA");
      rsaCipher.init(Cipher.DECRYPT_MODE, privateKey);
      byte[] decryptedMessageBytes = rsaCipher.doFinal(encryptedData);
      String decryptedMessage = new String(decryptedMessageBytes);
      return decryptedMessage;
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  public static void main(String[] args) {
    JceracfksExample1 example = new JceracfksExample1();
    String secretMessage = "Hi Alice, please don't tell anyone, but ..";

    try {
      System.out.println("Reading keys from Keystore");
      KeyPair keys = example.readKeys();
      System.out.println("Keys read");

      System.out.println("Encrypting: " + secretMessage);
      byte[] encrypted = example.encrypt(secretMessage, keys.getPublic());
      System.out.println("Message encrypted");

      System.out.println("Decrypting:");
      String decrypted = example.decrypt(encrypted, keys.getPrivate());
      System.out.println("Message decrypted: " + decrypted);
    } catch (Exception e) {
      System.err.println("Something went wrong ...");
      e.printStackTrace();
    }
  }
}
```

12.4 JCECCARACFKS

This example uses the JCECCARACFKS keystore. The setup for this example is similar to the one used for the JCERACFKS example, with the exception that the PCICC option of the RACDCERT command is used when generating the key pair and certificate. This option results in the key pair being generated by the hardware cryptographic coprocessor and the private key to be stored in the PKDS encrypted with the coprocessor asymmetric Master Key. Figure 12-5 on page 188 shows the sequence of RACDCERT commands used to:

- Create an RSA key pair and a self-signed certificate with a label AliceRSA2. The private key is kept in the PKDS with a label ALICE.RSA.TEST2.
- Create a RACF key ring with a label ALICE_JCECCARACFKS.
- Connect the key pair and certificate to the key ring owned by EBBEMP.

```
RACDCERT ID(EBBEMP) GENCERT
SUBJECTSDN(CN('Alice Self Signed Cert JCECCARACFKS'))
SIZE(1024) WITHLABEL('AliceRSA2') PCICC(ALICE.RSA.TEST2)

RACDCERT ID(EBBEMP) ADDRING(ALICE_JCECCARACFKS)

RACDCERT ID(EBBEMP) CONNECT(ID(EBBEMP) LABEL('AliceRSA2')
RING(ALICE_JCECCARACFKS) USAGE(PERSONAL))
```

Figure 12-5 Setup of the JCECCARACFKS example

The example program is shown in Figure 12-6 on page 190.

com.ibm.crypto.hdwrCCA.provider.RACFInputStream is used again, which allows the program to access RACF key rings and get the private key PKDS label.

Example 12-7 RSA encryption and decryption using a JCECCARACFKS key pair

```
import java.io.InputStream;
import java.security.KeyPair;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.cert.Certificate;

import javax.crypto.Cipher;

import com.ibm.crypto.hdwrCCA.provider.RACFInputStream;

public class JceccaracfksExample1 {
   public KeyPair readKeys() throws Exception {
      try {
         KeyStore jceracfks = KeyStore.getInstance("JCECCARACFKS");
         InputStream is = new RACFInputStream("EBBEMP",
                              "ALICE_JCECCARACFCCAKS", null);
         jceracfks.load(is, null); // no password

         Certificate aliceCertificate = jceracfks.getCertificate("AliceRSA2");
         PublicKey alicePublicKey = aliceCertificate.getPublicKey();

         PrivateKey alicePrivateKey = (PrivateKey) jceracfks.getKey("AliceRSA2",
                                                                    null);

         KeyPair keyPair = new KeyPair(alicePublicKey, alicePrivateKey);
         return keyPair;
      } catch (Exception e) {
         System.err.println("Could not read the keystore");
         throw new Exception(e);
      }
   }

   public byte[] encrypt(String clearText, PublicKey publicKey) {
      try {
         Cipher rsaCipher = Cipher.getInstance("RSA");
         rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);
```

```
            byte[] encryptedData = rsaCipher.doFinal(clearText.getBytes());
            return encryptedData;
         } catch (Exception e) {
            e.printStackTrace();
            return null;
         }
      }

      public String decrypt(byte[] encryptedData, PrivateKey privateKey) {
         try {
            Cipher rsaCipher = Cipher.getInstance("RSA");
            rsaCipher.init(Cipher.DECRYPT_MODE, privateKey);
            byte[] decryptedMessageBytes = rsaCipher.doFinal(encryptedData);
            String decryptedMessage = new String(decryptedMessageBytes);
            return decryptedMessage;
         } catch (Exception e) {
            e.printStackTrace();
            return null;
         }
      }

      public static void main(String[] args) {
         JceccaracfksExample1 example = new JceccaracfksExample1();
         String secretMessage = "Hi Alice, please don't tell anyone, but ..";

         try {
            System.out.println("Reading keys from Keystore");
            KeyPair keys = example.readKeys();
            System.out.println("Keys read");

            System.out.println("Encrypting: " + secretMessage);
            byte[] encrypted = example.encrypt(secretMessage, keys.getPublic());
            System.out.println("Message encrypted");

            System.out.println("Decrypting:");
            String decrypted = example.decrypt(encrypted, keys.getPrivate());
            System.out.println("Message decrypted: " + decrypted);
         } catch (Exception e) {
            System.err.println("Something went wrong ...");
            e.printStackTrace();
         }
      }
   }
```

12.5 PKCS11IMPLKS

This example shows a program that reads RSA keys from a PKCS#11 token, and then uses them for a simple encryption followed by decryption.

Before you can use z/OS PKCS#11 tokens in a Java application, you must complete several preliminary setup steps, as explained in 9.7, “The IBM providers - IBMPKCS11Impl” on page 138. The steps needed before you run the example are summarized here:

- Setting up a token with a signed certificate in it
- Creating a configuration file for the PKCS11Impl provider
- Properly setting up a local security property file

Setting up a z/OS PKCS#11 token with a signed certificate

In our case, we used the RACDCERT command as shown in Figure 12-6 to perform the following steps:

1. Add a z/OS PKCS#11 token into the TKDS, with the RACDCERT ADDTOKEN command. The created token has the label EBBE.TEST1.
2. For our own needs, we created a Certificate Authority key pair in RACF. This is the RACDCERT GENCERT command with a key pair and certificate label of EBBE_pkcs11_certauth. This CA key pair is then used to sign a certificate at the same time we create it with a label EBBE_pkcs11_cert.
3. Finally, both certificates and their keys are copied into the EBBE.TEST1 PKCS#11 token in the TKDS, using the RACDCERT BIND command. Note that when a certificate is bound to a token, RACF creates the following objects in the token:
   - A certificate object
   - A public key object
   - A private key object, if the certificate has an associated private key and the BIND USAGE is PERSONAL

```
RACDCERT ADDTOKEN('EBBE.TEST1')

RACDCERT CERTAUTH GENCERT SUBJECTSDN(CN('EBBE_pkcs11_certauth') OU('IGA')
O('IBM') L('Alleroed') SP('') C('DK')) WITHLABEL('EBBE_pkcs11_certauth')

RACDCERT GENCERT SUBJECTSDN(CN('EBBE_pkcs11_cert') OU('IGA') O('IBM')
L('Alleroed') SP('') C('DK')) WITHLABEL('EBBE_pkcs11_cert')
SIGNWITH(CERTAUTH LABEL('EBBE_pkcs11_certauth'))

RACDCERT BIND(CERTAUTH LABEL('EBBE_pkcs11_certauth') TOKEN(EBBE.TEST1))
RACDCERT BIND(LABEL('EBBE_pkcs11_cert') TOKEN(EBBE.TEST1))
```

Figure 12-6 Setup of the PKCS11Impl example

Note, however, that the following permissions should be granted in RACF for these commands to operate:

- The class of resources CRYPTOZ should be activated and RACLIST’ed with the following RACF commands:
  ```
  SETROPTS CLASSACT(CRYPTOZ) GENERIC(CRYPTOZ) RACLIST(CRYPTOZ)
  SETROPTS RACLIST(CRYPTOZ) REFRESH
  ```
- The Security Officer and User authority should be given to the user who is issuing the RACDCERT commands. Our approach is to prevent everyone except the user EBBEMP from having this authority:
  ```
  RDEF CRYPTOZ SO.EBBE.* UACC(NONE)
  RDEF CRYPTOZ USER.EBBE.* UACC(NONE)
  ```

```
PERMIT SO.EBBE.* CLASS(CRYPTOZ) ID(EBBEMP) ACC(CONTROL)
SETROPTS RACLIST(CRYPTOZ) REFRESH
```

Notice that we elected here to give these permissions for all z/OS PKCS#11 tokens with a label name beginning with the qualifier EBBE.

- Authority should also be granted for executing the RACDCERT BIND command. We use the same approach here of denying this authority to everyone except the user EBBEMP:

```
RDEF FACILITY IRR.DIGTCERT.BIND UACC(NONE)
PERMIT IRR.DIGTCERT.BIND CLASS(FACILITY) ID(EBBEMP) ACC(UPDATE)
RDEF FACILITY IRR.DIGTCERT.LIST UACC(NONE)
PERMIT IRR.DIGTCERT.LIST CLASS(FACILITY) ID(EBBEMP) ACC(CONTROL)
```

Note: We illustrate here the RACF commands that we used in our specific setup. However, many possible variations of these commands can be used, depending on the intended granularity of permissions that are to be granted.

Refer to *z/OS Security Server RACF Command Language Reference*, SA22-7687, to determine the commands that best fit your use case.

After creating the z/OS PKCS#11 token in the TKDS, its contents can be listed with the RACF command:

```
RACDCERT LISTTOKEN(EBBE.TEST1)
```

The token contents, that is, the two certificates and their keys, is displayed as shown in Figure 12-7. Notice that the token contains the private key for the certificate EBBE_pkcs11_cert because it was bound, by default, as USAGE(PERSONAL).

```
Token: EBBE.TEST1

 Seq Num  Attributes                    Labels
 -------- ----------------------------  ------------------------------------
 00000001 Default: NO    Priv Key: NO   TKDS: EBBE_pkcs11_certauth
          Usage: CERTAUTH Pub Key: YES  RACF: EBBE_pkcs11_certauth
          Owner: CERTAUTH

 00000003 Default: NO    Priv Key: YES  TKDS: EBBE_pkcs11_cert
          Usage: PERSONAL Pub Key: YES  RACF: EBBE_pkcs11_cert
          Owner: ID(EBBEMP)
```

Figure 12-7 EBBE.TEST1 z/OS PKCS#11 token contents

IBMPKCS11Impl configuration file

As part of the required setup we created a file named pkcs11.cfg and established three properties for the provider:

- name - The name we will use to refer to this provider
- library - The full path of the PKCS#11 DLLs library
- tokenlabel - The label of the token we want to work with

In our case, the IBMPKCS11Impl configuration file looked as shown in Figure 12-8 on page 192.

```
EBBEMP @ SC60:/u/ebbemp/security>cat pkcs11.cfg
   name=ebbe
   library=csnpca3x.so
   tokenlabel=EBBE.TEST1
```

Figure 12-8 PKCS11Impl configuration file

Local security file

Finally, we updated the providers list in the local security file javaPKCS11.security by inserting the IBMPKCS11Impl provider in the list along with the full path to the PKCS11Impl configuration file created in Example 12-8 on page 193.

The configuration file name must be specified after the name of the provider, separated by a space. After inserting the provider in the list, you must readjust the sequence numbers of the rest of the providers so that they are still in sequence. The final local security file we used is shown in Figure 12-9.

```
EBBEMP @ SC60:/u/ebbemp/security>oedit javaPKCS11.security

EDIT       /u/ebbemp/security/javaPKCS11.security          Columns 00001 00072
Command ===>                                                  Scroll ===> CSR
000046 # insertProviderAt method in the Security class.
000047
000048 #
000049 # List of providers and their preference orders (see above):
000050 #
000051 security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
000052 security.provider.2=com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl
/u/ebbemp/security/pkcs11.cfg
000053 security.provider.3=com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
000054 security.provider.4=com.ibm.crypto.provider.IBMJCE
000055 security.provider.5=com.ibm.security.jgss.IBMJGSSProvider
000056 security.provider.6=com.ibm.security.cert.IBMCertPath
000057 security.provider.7=com.ibm.security.sasl.IBMSASL
000058 security.provider.8=com.ibm.xml.crypto.IBMXMLCryptoProvider
000059 security.provider.9=com.ibm.xml.enc.IBMXMLEncProvider
000060 security.provider.10=org.apache.harmony.security.provider.PolicyProvider
000061 security.provider.11=com.ibm.security.jgss.mech.spnego.IBMSPNEGO
000062
000063 #
```

Figure 12-9 PKCS#11 example java.security file

Access to z/OS PKCS#11 token objects

The example program needs user authority to the token contents, because it will use the RSA keys already installed in the token. Because the program is running with the user ID EBBEMP, this authority is granted when by issuing the RACF command:

```
PERMIT USER.EBBE.* CLASS(CRYPTOZ) ID(EBBEMP) ACC(READ)
```

Our test program

Our test program is shown in Example 12-8. Note that the provider it refers to is IBMPKCS11Impl-ebbe, and not IBMPKCS11Impl, as specified with the name property in the IBMPKCS11Impl configuration file.

The z/OS PKCS#11 token is read like an ordinary keystore would be read. However, a password is not needed because access authority to the z/OS PKCS#11 token is granted through the USER.EBBE.* profile in the RACF CRYPTOZ class.

To get the program properly executed, the user must specify that the local security file is to be used instead of the JVM java.security file. This is achieved by invoking the Pkcs11Exemple1 program with the following command:

```
java -Djava.security.properties==/u/ebbemp/security/javaPKCS11.security
Pkcs11Example1
```

Example 12-8 RSA encryption and decryption using a key pair in a z/OS PKCS#11 token

```
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.cert.Certificate;

import javax.crypto.Cipher;

public class Pkcs11Example1 {
  public static void main(String[] args) {
    try {
      KeyStore pkcs11ks = KeyStore.getInstance("PKCS11IMPLKS",
                                          "IBMPKCS11Impl-ebbe");
      pkcs11ks.load(null, null); // no stream or password needed

      Certificate cert = pkcs11ks.getCertificate("EBBE_pkcs11_cert");
      PublicKey  publicKey = cert.getPublicKey();
      PrivateKey privateKey =
            (PrivateKey) pkcs11ks.getKey("EBBE_pkcs11_cert", null);

      String secretMessage = "A very secret message ...";

      System.out.println("Encrypting: " + secretMessage);
      Cipher rsaCipher = Cipher.getInstance("RSA", "IBMPKCS11Impl-ebbe");
      rsaCipher.init(Cipher.ENCRYPT_MODE, publicKey);
      byte[] encryptedData = rsaCipher.doFinal(secretMessage.getBytes());

      System.out.println("Decrypting");
      rsaCipher.init(Cipher.DECRYPT_MODE, privateKey);
      byte[] decryptedDataBytes = rsaCipher.doFinal(encryptedData);
      String decryptedMessage = new String(decryptedDataBytes);
      System.out.println("Message decrypted: " + decryptedMessage);
    } catch (Exception e) {
      System.err.println("Something went wrong ..");
      e.printStackTrace();
    }
  }
}
```

12.6 Retrieving keys by their ICSF CKDS or PKDS label

In this example we perform symmetric encryption and decryption using two Triple-DES keys that reside in the ICSF CKDS. This is achieved via the IBMJCECCA-specific KeyLabelKeySpec class, which allows users to specify a key by passing its CKDS or PKDS label. Because the KeyLabelKeySpec is not a standard provider feature, access to the ibmjcecca.jar file is needed when compiling the example.

In our example, these are the labels of the two keys:

- SG24.CLEAR.TDES - This a clear Triple-DES key in the CKDS.
- SG24.PROT.TDES - This is a protected (that is, encrypted with the symmetric Master Key) Triple-DES key also in the CKDS.

Our test program is shown in Example 12-9.

Example 12-9 Triple-DES encryption and decryption calling CKDS keys with their label

```
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.IvParameterSpec;

import com.ibm.crypto.hdwrCCA.provider.KeyLabelKeySpec;

public class CkdsExample1 {
  public void encryptDecryptTest(String ckdsLabel) {
    try {
      System.out.println("Encrypt/decrypt test with label: " + ckdsLabel);

      SecureRandom rnd = SecureRandom.getInstance("IBMSecureRandom", "IBMJCECCA");
      byte[] ivData = new byte[8];
      rnd.nextBytes(ivData);
      IvParameterSpec ivSpec = new IvParameterSpec(ivData);

      SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DESEde",
                                                          "IBMJCECCA");

      // get a key object from the CKDS label
      KeyLabelKeySpec spec = new KeyLabelKeySpec(ckdsLabel);

      SecretKey tdesKey = keyFactory.generateSecret(spec);

      // Get a DESede cipher for encryption (DESede = Triple DES)
      Cipher cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding", "IBMJCECCA");

      String secretMessage = "CC No: 371234567890";

      System.out.println("Encrypting: " + secretMessage);
      cipher.init(Cipher.ENCRYPT_MODE, tdesKey, ivSpec);
      byte[] encryptedData = cipher.doFinal(secretMessage.getBytes());

      System.out.println("Decrypting");
      cipher.init(Cipher.DECRYPT_MODE, tdesKey, ivSpec);
```

```
      byte[] decryptedBytes = cipher.doFinal(encryptedData);
      String decryptedMessage = new String(decryptedBytes);
      System.out.println("Decrypted data: " + decryptedMessage);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    CkdsExample1 example = new CkdsExample1();
    example.encryptDecryptTest("SG24.CLEAR.TDES");
    System.out.println();
    example.encryptDecryptTest("SG24.PROT.TDES");
  }
}
```

When properly executed, the test program issues the messages shown in Figure 12-10.

```
Encrypt/decrypt test with label: SG24.CLEAR.TDES
Encrypting: CC No: 371234567890
Decrypting
Decrypted data: CC No: 371234567890

Encrypt/decrypt test with label: SG24.PROT.TDES
Encrypting: CC No: 371234567890
Decrypting
Decrypted data: CC No: 371234567890
```

Figure 12-10 Output of test program

Part 4

Appendixes

© Copyright IBM Corp. 2008. All rights reserved.

A

z/OS integrated hardware cryptography setup details

z/OS Java does not require a specific physical configuration to be performed for the zSeries or System z hardware cryptography. Instead, it exploits the same configuration as done for the earlier applications. This appendix explains and summarizes this configuration for readers who are new to this setup. Experienced readers may also benefit from reviewing the security considerations that are addressed along with the configuration steps.

This appendix discusses the following topics:

- Background information, including hardware components of the System z cryptography infrastructure
- The installation and setup steps required to use hardware cryptography with z/OS Java applications. Note that this information also applies to System z hardware cryptography use by programming models on z/OS other than Java.

© Copyright IBM Corp. 2008. All rights reserved.

Hardware components of the System z cryptography infrastructure

This section describes the zSeries and System z cryptographic coprocessors, the Trusted Key Entry (TKE) workstation, and the zAAP specialty engine.

Cryptographic coprocessors

The hardware coprocessor technologies available on the IBM zSeries and System z models differ, depending on the models they have been implemented on. Also note that some of the features referred to here are no longer available. Table A-2 on page 206 and Table A-3 on page 206 list the technologies discussed in this section.

The details provided in this appendix mainly relate to the coprocessor technology used in IBM Server z10.

Evolution of the cryptographic processors

Figure A-1 displays a graphical view of the evolution of the cryptographic hardware coprocessors in the zSeries z900/z800 family, in the z990/z890 family, and in the z9 and z10 servers.

The z10 supports two kinds of cryptographic hardware devices

- The Central Processor Assist for Cryptographic Functions (CPACF)
- The Crypto Express2 Coprocessor (CEX2C)

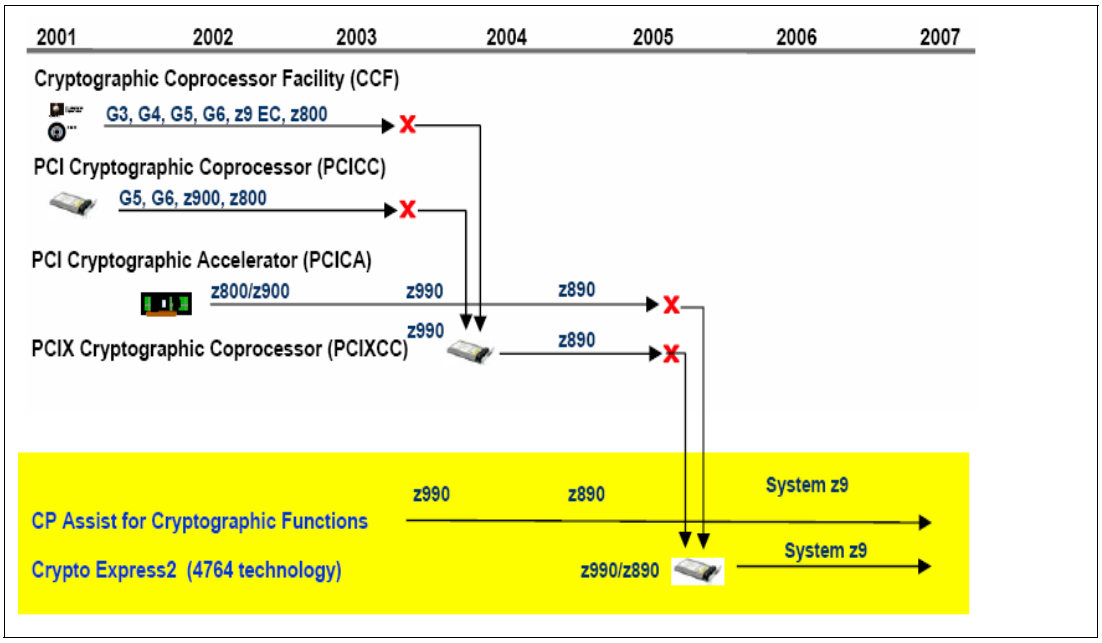


Figure A-1 Evolution of zSeries and System z cryptographic coprocessors technologies

The following section briefly explains the different coprocessor models that are referred to in the figure.

Cryptographic Coprocessor Feature

The Cryptographic Coprocessor Feature (CCF) is a standard orderable feature for the zSeries z900 and z800 family of systems. It provides hardware support for symmetric and

asymmetric algorithms, along with miscellaneous cryptographic services. The CCF supports secure key operations and has been certified to FIPS 140-1 Level 4. There are up to two CCF in a zSeries server, which requires a system power-on Reset (POR) to be activated. The CCF can be shared by up to 16 logical partitions.

PCI Cryptographic Coprocessor

The PCI Cryptographic Coprocessor (PCICC) is an optional priced feature for the z900/z800 family of systems (FC 0860). It interfaces with the host system using a Peripheral Component Interconnect (PCI) physical and logical connection.

On zSeries, the PCI interface is actually emulated by the system's own hardware pluggable interface. This hardware coprocessor specializes in asymmetric (RSA) computations and other miscellaneous functions that are not covered by the CCF.

The PCICC supports secure key operations and has been certified FIPS 140-1 Level 4. The PCICC also supports the capability of installing customers' own algorithms as additional coprocessor microcode (the User Defined Extensions, UDX, function).

The PCICC coprocessor can be shared by up to 16 logical partitions.

PCI Cryptographic Accelerator

The PCI Cryptographic Accelerator (PCICA) is an optional priced feature for the z900/z800 and z990/z890 families of systems (FC 0862). It is specialized in providing high speed hardware assist for the SSL/TLS handshake using the RSA algorithm. The PCICA operates with clear keys only.

The PCICA coprocessor can be shared by up to 16 logical partitions.

PCI eXtended Cryptographic Coprocessor

The PCI eXtended Cryptographic Coprocessor (PCIXCC) is an optional priced feature for the z990/z890 family of systems (FC 0868). It provides most of the services available in the CCF, PCICC, and PCICA devices.

At its core, the PCIXCC uses the so-called "4764" technology, so that PCIXCC and 4764 are often used interchangeably.

The PCIXCC has been certified at FIPS 140-2 Level 4.

The PCIXCC coprocessor can be shared by up to 16 logical partitions.

Crypto EXpress2 Coprocessor

The Crypto EXpress2 Coprocessor (CEX2C) is an optional priced feature for the z990/z890, z9, and z10 family of systems (FC 0863 or FC 0870). The CEX2C actually contains PCIXCC/4764 coprocessors (one or two, depending on the feature).

It supports the manual reconfiguration of coprocessors into "accelerators" (CEX2A). In accelerator mode, a CEX2A coprocessor is limited to providing the same functions as the previously available PCICA feature. That is, the functions required for the hardware support of the SSL/TLS handshake, but at a even higher speed than the PCICA feature.

When operating in coprocessor mode, the CEX2C coprocessor supports secure keys and the 4764 FIPS 140-2 Level 4 certification applies.

The CEX2C coprocessor can be shared by up to 16 logical partitions.

CP Assist for Cryptographic Functions

CP Assist for Cryptographic Functions (CPACF) is a standard orderable feature for the z990/z890, z9 and z10 families of systems. It is not an actual coprocessor per se but can be thought of as a set of specialized circuits imbedded in the system's processing units. It offers a limited set of functions dealing with very high speed symmetric and hash operations, and uses clear keys exclusively.

There is no concept of logical partition sharing for the CPACF because it is available to any logical partition dispatched on the PU.

Note: CPACF does not provide any support for asymmetric algorithms. It is therefore required to have at least one Crypto Express 2 feature installed in the system if RSA hardware assist is required.

A word on clear keys and secure keys

Figure A-2 illustrates the concepts of clear keys and secure keys. On the right, the CPACF or the Crypto Express2 in accelerator mode use clear keys only. The secret key is passed by the application requesting the service, and can then be seen in the system's memory.

Conversely, the Crypto Express2 implementation supports the concept of secure key, where the applications' secret keys are kept in the system's memory or file devices encrypted by a higher level key called the Master Key. The Master Key resides within the coprocessor physical boundary. It is installed and maintained by a set of security officers (at least two) and cannot be viewed from outside the coprocessor. The applications' secret keys are provided to the coprocessor encrypted with the Master Key and are decrypted, before being used, inside the coprocessor.

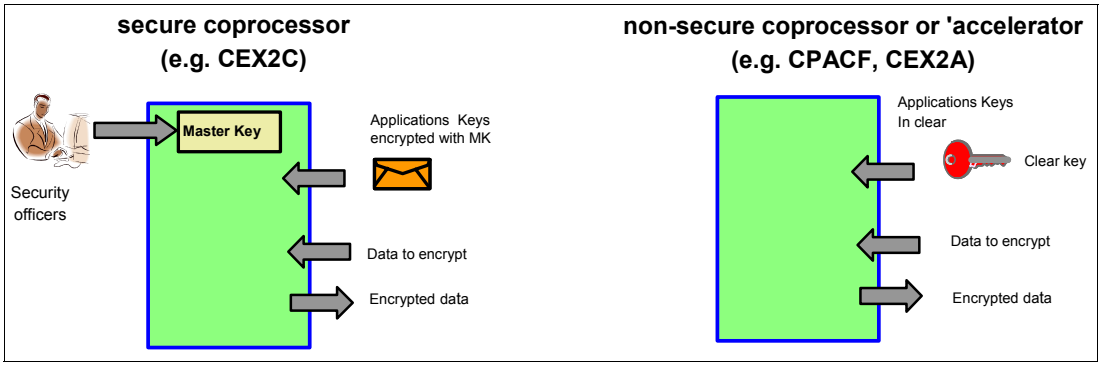


Figure A-2 Secure and non-secure coprocessors

The notion of "cryptographic domains"

The set of physical resources within a coprocessor that are made available to one logical partition is called a "domain". Currently a coprocessor supports up to 16 domains. That is, it hosts circuits to keep up to 16 sets of Master keys, one set per logical partition that have access to the coprocessor, and circuits to drive the input and output data flows with each one of the logical partitions.

The assignment of the domain to a logical partition is specified in the partition's image profile, which is defined at the system Hardware Management Console (HMC). The logical partition is assigned the domain in the coprocessor(s) it has access to, during its activation. This makes the domain in the given coprocessor unavailable for any other partition.

Note that the logical partition's image profile can have more than one domain specified for the partition. In that case, the single domain to be actually assigned to the partition is then specified in the Options Dataset of the ICSF instance executing in the partition.

Trusted Key Entry workstation

The Trusted Key Entry (TKE) workstation is an optional priced feature dedicated to the secure administration of the coprocessors Master Keys. It is a highly secure alternative to the ICSF ISPF panels that are used, by default, for coprocessor administration.

This feature is an IBM xSeries® with an integrated 4764 cryptographic hardware coprocessor and a specific application running under a special purpose operating system. The TKE is connected via TCP/IP to a z/OS instance that hosts ICSF and has access to the coprocessors to be administered. The communication can occur over a non-secure network because it is secured using encryption of data and digital signature of messages exchanged between the coprocessors and the security officers operating at the TKE.

System z Application Assist Processor

The System z Application Assist Processor (zAAP) specialty engine is a priced processor that can provide a financially attractive solution for executing Java applications on z/OS. The zAAP is designed to operate asynchronously with the general purpose CPs of the system and is dedicated to executing the Java instructions flows under control of the IBM Java Virtual Machine (JVM).

This can help reduce the demands and capacity requirements on general purpose CP processors (and its possible consequences on the software charges incurred by the execution of licensed programs) when it comes to execute Java applications.

This addresses the hardware cryptographic functions as well, because the zAAP has its own CPACF circuits, and any request for cryptographic services by a Java application will translate into additional zAAP cycles as opposed to general CP cycles.

The JVM processing cycles can be executed on the configured zAAPs with no anticipated modifications to the Java applications. Execution of the JVM processing cycles on a zAAP is a function of the IBM Software Developer's Kit (SDK) for z/OS, Java 2 Technology Edition V1.4 with z/OS 1.6.

The z/OS integrated hardware cryptography infrastructure

This section describes the components that require the z/OS infrastructure to use System z hardware cryptography, as shown in Figure A-3 on page 204.

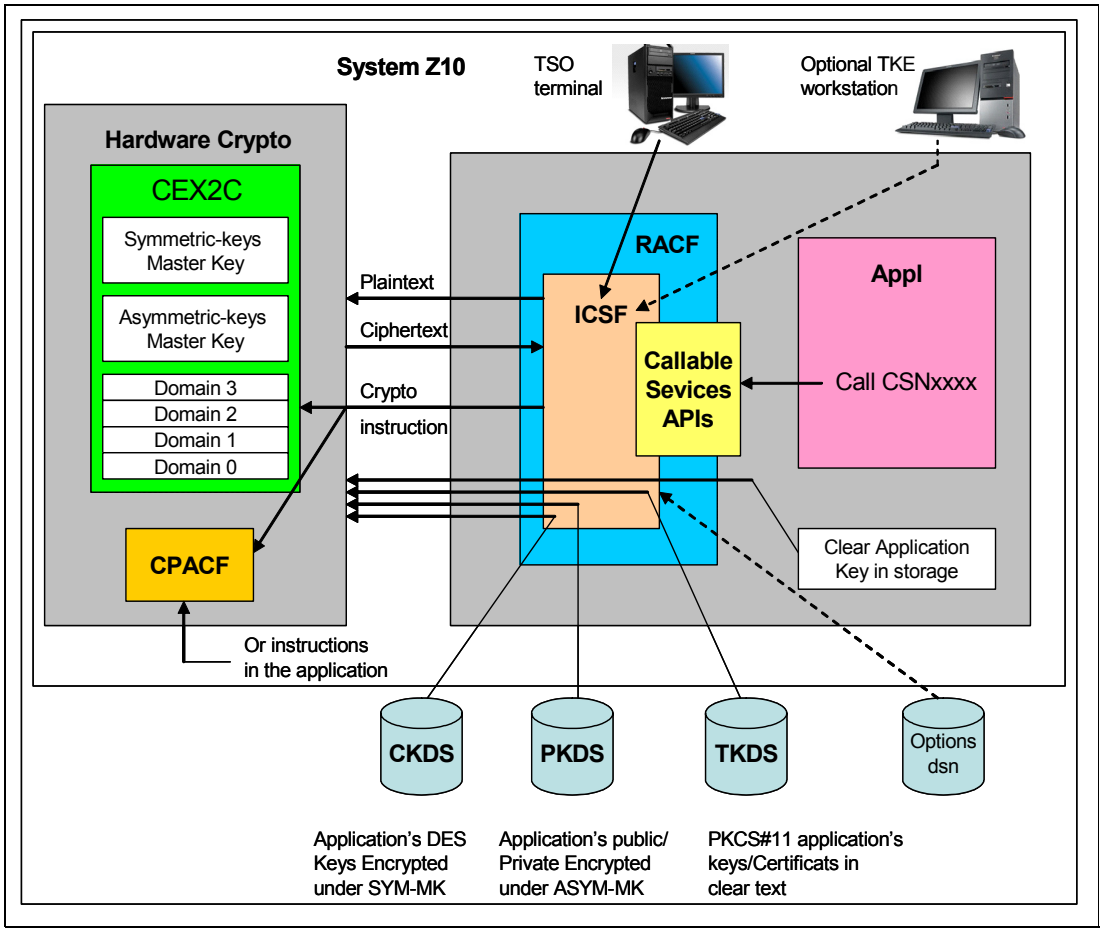


Figure A-3 z/OS integrated hardware cryptography infrastructure

Integrated Cryptographic Service Facility

Integrated Cryptographic Service Facility (ICSF) is the z/OS software component that drives the cryptographic hardware coprocessors. It provides the low level API for applications to interact with the coprocessors. It is the default interface for the administration of the coprocessors with these functions:

- Deactivation or activation of coprocessors
- Master keys and key datasets administration
- Monitoring of the installation options for ICSF
- Administrative functions of cryptographic services
- Utility functions invocations
- Interactive interface for the Key Generation Utility Program (KGUP)

ICSF invokes RACF for the protection, if any, of cryptographic keys (using profiles in the CSFKEYS class of resources), of cryptographic objects used through the PKCS#11 API (with the CRYPTOZ class of resources) and ICSF services (with the CSFSERV class of resources).

ICSF manages the following VSAM data sets.

Cryptographic Key Data Set

Cryptographic Key Data Set (CKDS) is used to store applications DES (symmetric) keys encrypted under the symmetric Master Key or clear AES keys. The keys stored in the CKDS are designated by their key label.

It is mandatory that a CKDS be allocated to ICSF and initialized with the symmetric Master Key.

ICSF maintains, for performance reason, an in-storage copy of the CKDS, with XCF-based sysplex data sharing support.

Public Key Data Set

Public Key Data Set (PKDS) is used to store applications' RSA asymmetric keys encrypted under the asymmetric Master Key or in clear form. There is an optional in-storage buffering of the PKDS.The keys stored in the PKDS are designated by their key label.

It is mandatory that a PKDS be allocated to ICSF and initialized with the asymmetric Master Key.

Token Key Data Set

Token Key Data Set (TKDS) is an optional dataset that is used and required only if applications are using the PKCS#11 functions. The TKDS records contain the z/OS PKCS#11 token objects.

ICSF maintains an in-storage copy of the TKDS objects, with XCF-based sysplex data sharing support.

Java keystores and the ICSF VSAM datasets

Java keystores on z/OS are backed up by the ICSF datasets shown in Table A-1.

Table A-1 Java keystores and ICSF datasets

Java key store	CKDS	PKDS	TKDS
JCECCAKS	X	X	
JCECCARACFKS		X	
PKCS11IMPLKS			X

Hardware setup

It is required to set up and enable the hardware cryptography facilities in zSeries or System z. This section describes the operations to be performed to make the hardware devices available for use. The steps in these operations may differ, depending on which system family the hardware devices belong to.

Cryptographic devices and system families

Table A-2 summarizes in which system families you can find the different cryptographic hardware devices described in “Cryptographic coprocessors” on page 200.

Table A-2 Cryptographic coprocessors by system family and model

Model	CCF	PCICC	PCICA	PCIXCC	CEX2C	CPACF	ZAAP (with CPACF)
z800 z900	Yes	Yes	Yes	No	No	No	No
z890 z990	No	No	Yes	Yes	Yes	Yes	Yes
z9	No	No	No	No	Yes	Yes	Yes
z10	No	No	No	No	Yes	Yes	Yes

Table A-3 indicates the maximum number of installed features by system family and model. Note that some features contain more than one coprocessor (for example, the CEX2C feature contains two coprocessors).

Table A-3 Number of features by system family and models

Type of Coprocessor (coprocessor number per feature)	z800 z900	z890 z990	z9 BC	z9 EC	z10 EC™	z10 BC
PCICC (2)	8	-	-	-	-	
PCICA (2)	6	2 [a] / 6	-	-	-	
PCIXCC (1)	-	4	-	-	-	
CEX2C-1P (1) [b]	-	-	4 (R07) 8 (R07)	-	-	8
CEX2C (2)	-	8	4 (R07) 8 (S07)	8	8	8
ZAAP (1)	-	16	3	27	32	5

a. For a z890, there is a maximum of two PCICA features.
b. The CEX2C-1P can be installed in z9 BC and z10 BC servers only.

Hardware installation

CCF and CPACF come as a standard orderable feature and are not required to be installed in the system. However, they are required to be enabled because they are delivered in a non-operational state.

CCF is enabled by installing the IBM provided enablement diskette, followed by a system power-on Reset (POR). Enabling CCF is a prerequisite for enabling the other coprocessor types that coexist in a system with the CCF.

CPACF is enabled by installing the LICC FC 3863. Enabling CPACF does not disrupt system operations. Enabling CPACF is a prerequisite for enabling the other coprocessor types that coexist in a system with the CPACF.

PCICC requires an enablement diskette of its own, but enabling PCICC does not disrupt system operations.

The other coprocessor types do not require a specific enablement process to be run as long as the CCF or CPACF in the system has been enabled.

Cryptographic hardware definitions on system z10

This section describes the hardware definitions that must be performed on a System z10 so that the cryptographic coprocessors can be accessed and used by logical partitions. Refer to the proper system documentation for defining the hardware on systems not in the System z10 family.

The definitions are entered into the system using the Hardware Management Console (HMC) or directly on the system's Support Element (SE). For more detailed information about this topic, refer to *System z10 Enterprise Class Processor Resource/Systems Manager Planning Guide*, SB10-7153.

Verifying hardware cryptography enablement

To verify that hardware cryptography is enabled on the system, follow these steps:

1. Logon to the system HMC with the user SYSPROG, or a user with similar system management privileges.
2. If multiple CPCs are connected to the HMC, select the one to connect to from the Servers list and invoke the Single Object Operations task, as shown in Figure A-4.

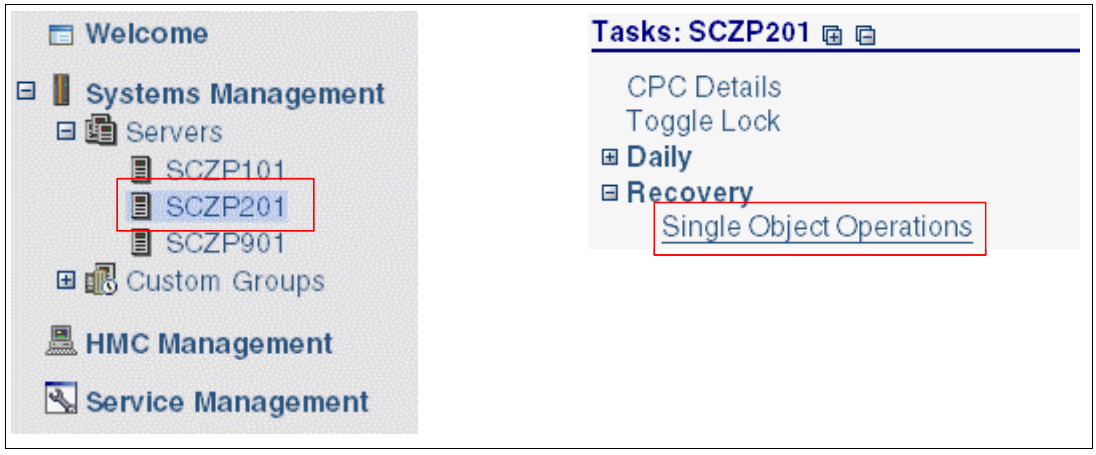


Figure A-4 Connecting to the target Support Element

3. From the Systems Management section on the left side of the screen, locate and click the target CPC and select the CPC detail, as shown in Figure A-5 on page 208.

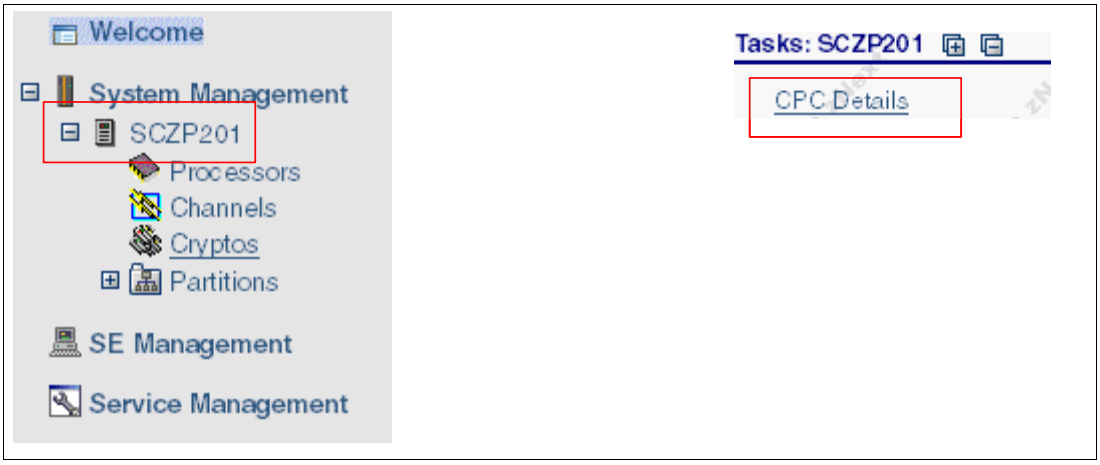


Figure A-5 Selecting CPS' Details

This CPC details panel shown in Figure A-6 opens.

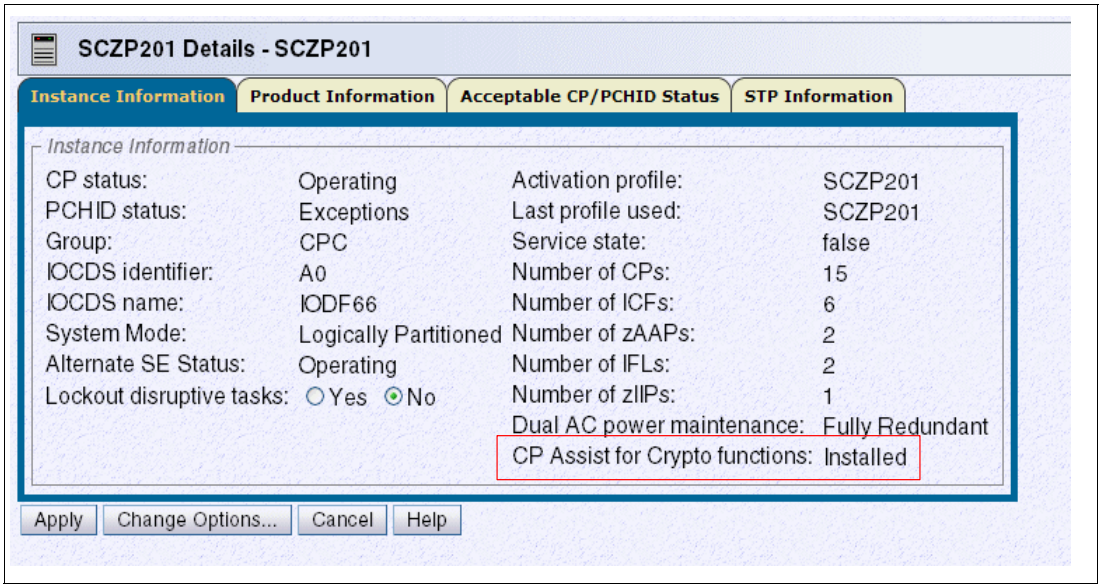


Figure A-6 Details for the selected CPC

Notice the information `CP Assist for Cryptographic Functions: Installed`, which indicates that FC 3863 is installed in the system.

Logical partition definitions

After performing this verification, you can now define cryptographic resources for your LPARs. Follow these steps:

1. As shown in Figure A-7 on page 209, select and click the target CPC from the Servers list in the HMC main menu. In the Operational Customization list, select **Customize/delete Activation Profiles**. In the list of profiles, select the target logical partition image profile, then click **Customize Selected Profile**.

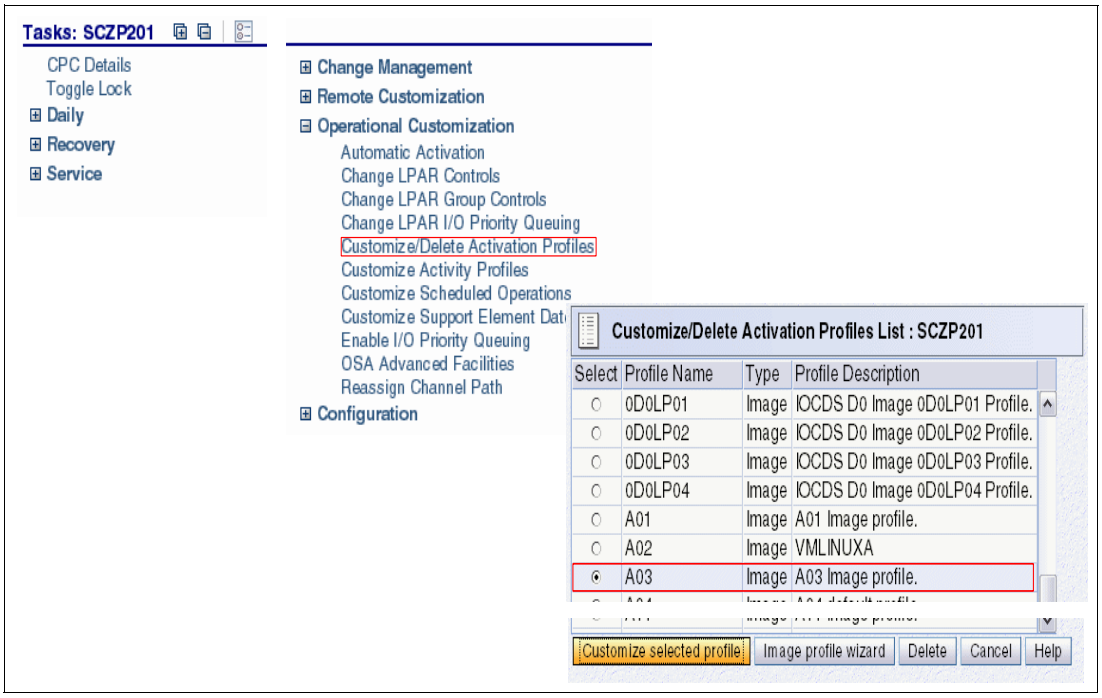


Figure A-7 Selecting the image profile to customize

2. Select the Crypto tab in the image profile. The list box for assignment of cryptographic coprocessors is displayed, as shown in Figure A-8.

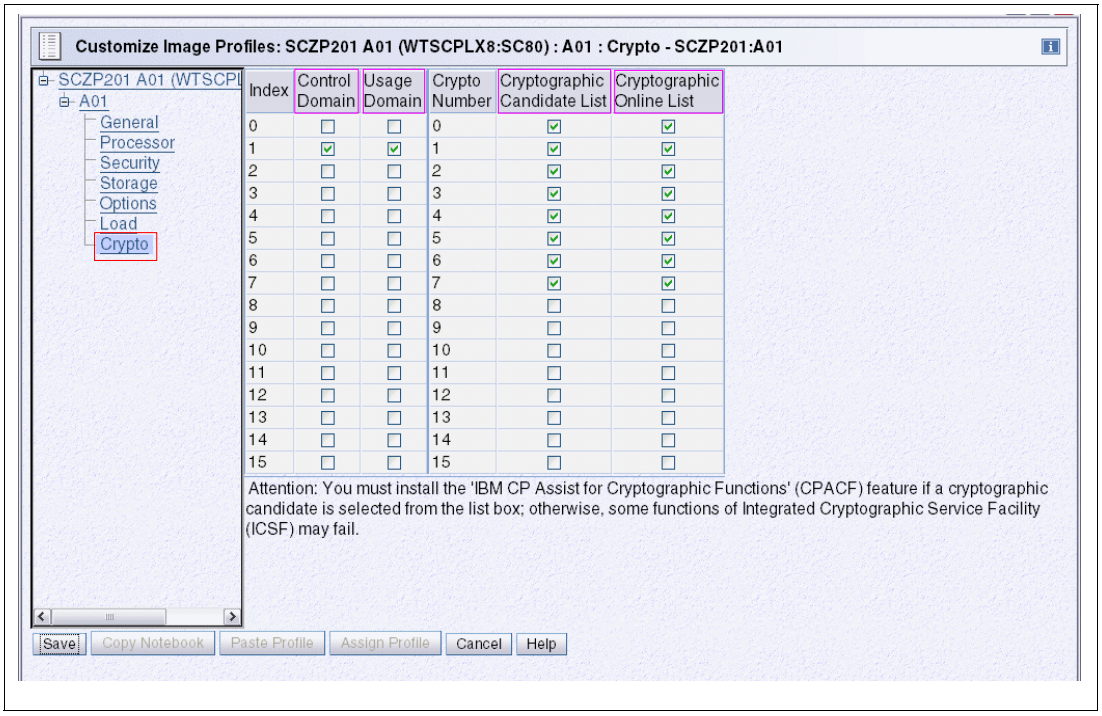


Figure A-8 Customizing an image profile for access to cryptographic coprocessors

The selections to be made address the following:

- **Usage Domain Index** - This is the selection of the cryptographic coprocessor domain(s) assigned to the partition in all cryptographic coprocessors that the partition has access to.

  If more than one usage domain is specified for the partition, only one of these domains should be used after the partition has been activated. This is the domain indicated in the Options Dataset of ICSF.

  The same usage domain index can be used by multiple partitions regardless of which LCSS they are defined to. However, the combination cryptographic coprocessor number and usage domain index number must be unique across all partitions planned to be active at the same time.

  Although it is possible to define duplicate combinations of cryptographic coprocessor number(s) and usage domain index(es) for different partitions image profiles, only one of these logical partitions can be active at any time. This is a valid definition, however, for backup configurations.

- **Control Domain Index** - This is the selection of the cryptographic coprocessors domain index(es) that can be administered from this logical partition being set up as the TCP/IP host for the TKE workstation. The usage domain index specified for this partition should also be part of the selected control domains.

- **PCI Cryptographic Coprocessor Candidate List** - This is the selection of the cryptographic coprocessors that are eligible to be accessed by this logical partition. From the scrollable list, select the coprocessor number(s), from 0 to 15, that identifies the coprocessor(s) to be accessed by this partition. Unless the coprocessor is also specified in the Online List, you must configure the coprocessor “on” in the Crypto Service Operations task list to make it accessible to the partition.

- **PCI Cryptographic Coprocessor Online List** - This is the selection of the cryptographic coprocessors that are automatically brought online to the partition during the logical partition activation. The coprocessor numbers selected in the Online List must also be part of the Candidate List.

  If the cryptographic coprocessor number and usage domain index combination for the coprocessor selected in the partition Online List is already in use by another active logical partition, the activation of the logical partition will fail.

zAAP definition

As for the cryptographic coprocessors, zAAPs are selected for access by logical partition. Select the target CPC in the HMC Servers list. Then, in the Operational Customization list, select the **Customize/Delete Activation Profiles** task. Select the target image profile and click the **Customize selected Profile** action. Select **Processor** in the image profile tab, as shown in Figure A-9 on page 211.

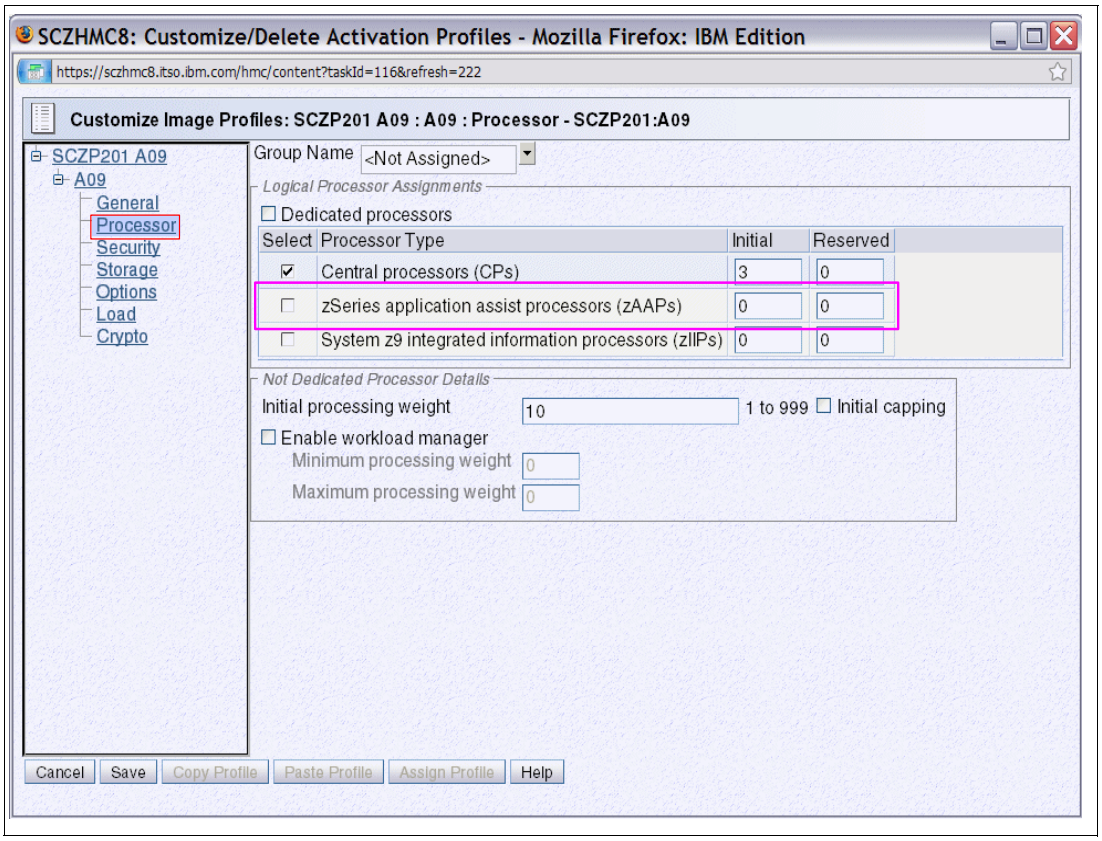


Figure A-9 Assign zAAP for the image

Select the appropriate number of zAAPs and save the configuration. Repeat this operation for each logical partition that needs to be customized for access to a zAAP specialty engine.

Software setup

This section explains how to set up the ICSF component of z/OS and how to start it. The steps to follow are presented here in their typical sequence order when performed at most installations.

Customizing SYSx.PARMLIB

- The load library for ICSF (hlq.SCSFMOD0) must be in the LINKLIST and APF lists of libraries. This can be achieved by dynamically changing these lists using the SETPROG APF and SETPROG LNKLST commands. The two members PROGxx and LNKSLSTxx of the PARMLIB also must be updated for a persistent change.
- The modules CSFDAUTH and CSFDPKDS must be added to the sections AUTHPGM and AUTHTSF of the IKJTSOxx member, as shown in Figure A-10 on page 212.

```
Menu  Utilities  Compilers  Help

------------------------------------------------------------------------------
BROWSE    SYS1.PARMLIB(IKJTSOxx) - 01.15            Line 00000000 Col 001 080
 Command ===>
********************* Top of Data ***********************************************
AUTHPGM NAMES (                          /* AUTHORIZED PROGRAMS */   +
       ….
       CSFDAUTH                                  /* ICSF */   +
       CSFDPKDS                                  /* ICSF */   +
       …….
 AUTHTSF NAMES (                          /* PROGRAMS TO BE AUTHORIZED */   +
       ….
       CSFDAUTH                                 /* ICSF */   +
       CSFDPKDS                                 /* ICSF */   +
       …….
********************** Bottom of Data ***************************************
```

Figure A-10 View of the IKJTSOxx PARMLIB's member

- For ease of use and maintenance, we recommend that you create a system symbol for each logical partition in the system &LPNUMn. These symbols are added in the IEASYMxx member as shown in Figure A-11.

```
Menu  Utilities  Compilers  Help

------------------------------------------------------------------------------
BROWSE    SYS1.PARMLIB(IEASYMnn) - 01.05          Line 00000000 Col 001 080
 Command ===>
********************* Top of Data ***********************************************
…….,
SYSDEF SYMDEF(&LPNUMA='x'),
SYSDEF SYMDEF(&LPNUMB='x'),
SYSDEF SYMDEF(&LPNUMC='x'),
….
********************** Bottom of Data ***************************************
```

Figure A-11 View of the IEASYMxx PARMLIB member

- ICSF requires run-time parameters to be specified in the Options Dataset. It is common practice to use the CSFPRMxx member of the PARMLIB as the Options Dataset to contain such parameters as the domain to use in the coprocessors and the dataset name of the ICSF CKDS, PKDS and TKDS. Figure A-12 on page 213 illustrates the contents of the Options Dataset.

```
Menu  Utilities  Compilers  Help
------------------------------------------------------------------------------
BROWSE    SYS1.PARMLIB(CSFPRMx0) - 01.06         Line 00000000 Col 001 080
 Command ===>
************************ Top of Data ***********************************
CKDSN(SYS1.&SYSNAME..SCSFCKDS)
PKDSN(SYS1.&SYSNAME..SCSFPKDS)
TKDSN(SYS1.&SYSNAME..SCSFTKDS)
PKDSCACHE(64)
COMPAT(NO)
SSM(YES)
DOMAIN(&LPNUMx)
KEYAUTH(YES)
CHECKAUTH(YES)
SYSPLEXTKDS(YES/NO)
SYSPLEXTKDS(YES/NO)
TRACEENTRY(1000)
USERPARM(USERPARM)
COMPENC(DES)
REASONCODES(ICSF)
************************** Bottom of Data **********************************
```

Figure A-12 View of the CSFPRMx0 PARMLIB member

ISPF panels

The ISPF options must be customized to make the ICSF ISPF panels selectable and operating. Figure A-13 shows a modified primary ISPF panel to offer the ICSF option.

```
Menu  Utilities  Compilers  Help

------------------------------------------------------------------------------
BROWSE  hlq.PANELS(ISR@PRIM) - 01.27             Line 00000000 Col 001 080
 Command ===>
************************ Top of Data ***********************************
)PANEL KEYLIST(ISRSAB,ISR)
)ATTR DEFAULT(...) FORMAT(MIX)
.....
)BODY  CMD(ZCMD)
.....
ICSF             For ICSF panels selection
.....
)PROC
.....
ICSF,'CMD(%$ICSF)'
.....
)END
************************ Bottom of Data ********************************
```

Figure A-13 Modified ISPF primary menu

Add the datasets shown in Figure A-14 in the TSO logon procedure. If an alternate TSO procedure is developed for the ICSF users, make sure that these users have proper access to the new procedure.

```
Menu  Utilities  Compilers  Help

------------------------------------------------------------------------------
BROWSE    SYS1.PROCLIB(<tso_proc>) - 01.06         Line 00000000 Col 001 080
 Command ===>
********************* Top of Data ********************************************
//tso_proc       EXEC    PGM=IKJTSO01, …..
//SYSPROC   DD          DSN=hlq.SCSFCLIO,DISP=SHR
// DD           DSN=……
//ISPPLIB       DD          DSN=hlq.SCSFPNLO,DISP=SHR
//                      DD          DSN=……
//ISPMLIB      DD          DSN=hlq.SCSFMSGO,DISP=SHR
//                      DD          DSN=……
//ISPSLIB       DD          DSN=hlq.SCSFSKLO,DISP=SHR
//                      DD          DSN=……
//ISPTLIB       DD          DSN=hlq.SCSFTLIB,DISP=SHR
********************** Bottom of Data ****************************************
```

Figure A-14 ICSF libraries in the TSO logon procedure

Allocating CKDS, PKDS, and TKDS datasets

Allocating these VSAM datasets is done by using the IDCAMS utility program. An example is provided for each of them in the CSFKDS member of the SYS1.SAMPLIB.

They must be allocated on system-resident DASD. Ensure that all the resident volumes are *not* enabled for dataset migration. Also, protect access to the datasets with the appropriate RACF profiles. Figure A-15 on page 215 shows the JCL provided in the SAMPLIB for allocating a CKDS.

```
Menu  Utilities  Compilers  Help

------------------------------------------------------------------------------
BROWSE    SYS1.SAMPLIB(CSFCKDS) - 01.00           Line 00000000 Col 001 080
 Command ===>
******************** Top of Data ********************************************
//CSFCKDS JOB <job card parameter>
//DEFINEEXEC PGM=IDCAMS,REGION=4M
//SYSPRINTDD SYSOUT=*
//SYSINDD *
  DEFINE CLUSTER (NAME(hlq.CFSCKDS)        -
VOLUMES(xxxxxx)              -
     RECORDS(100 50) -
     RECORDSIZE(252,252)      -
     KEYS(72 0)                          -
     FREESPACE(10,10)            -
     SHAREOPTIONS(2))           -
                 DATA   (NAME(hlq.CFSCKDS.DATA)     -
     BUFFERSPACE(100000)           -
     ERASE                                    -
     WRITECHECK                             -
                 INDEX    (NAME(hlq.CFSCKDS.INDEX))
/*
********************* Bottom of Data ***************************************
```

Figure A-15 JCL to allocate the CKDS

Startup procedure for ICSF

ICSF is started with an STC procedure like that shown in Figure A-16.

```
Menu  Utilities  Compilers  Help

------------------------------------------------------------------------------
BROWSE    SYS1.PROCLIB(<CSF>) - 01.04                 Line 00000000 Col 001
080
 Command ===>
********************** Top of Data ***************************************
//CSF            PROC   MEMBER=CSFPRMx0
//                      EXEC    PGM=CSFMMAIN,REGION=0M,TIME=NOLIMIT
//STEPLIB     DD       DSN=hlq.SCSFMOD0,DISP=SHR
//CSFLIST      DD         SYSOUT=*,DCB=(LRECL=132,BLKSIZE=132),HOLD=YES
//CSFPARM   DD         DSN=SYS1.PARMLIB(&MEMBER.),DISP=SHR
********************** Bottom of Data ************************************
```

Figure A-16 ICSF STC

We recommend that you define the started task name in the RACF STARTED class of resources, associated to a protected userID. A protected userID is a userID without a password, so that the userID cannot be misused to logon to the system.

Initializing the Master Key

Most of the functions provided by the coprocessors require that you have a Master Key set in the coprocessor domain to be used by the logical partition. There are actually two Master Keys per cryptographic domain:

- The asymmetric (or DES) Master Key to encrypt the applications' DES or Triple-DES keys
- The asymmetric (or PKA) Master Key to encrypt the applications' RSA keys

Therefore a reference to Master Key management, unless specified otherwise, generally involves the two Master Keys.

There are three different ways of setting up the Master Keys in the coprocessors domain:

- By using the Passphrase Initialization function
  - The value of the Master Keys is derived from an input character string that is 16 to 64 characters in length (that is, the "passphrase").

    This method, however, works only on uninitialized domains. That is, Passphrase Initialization cannot be used to change an already initialized Master Key.
- By using the change Master Key ISPF panel of ICSF

  This is the "production" method of administering the Master Keys, although using the ISPF panels is definitely less secure than using the TKE workstation. The ISPF panels offer options to initialize or modify the Master Keys, making it mandatory to have at least two security officers (the maximum number of officers is fixed by the installation) to enter each one in their own part of the Master Key. This is the implementation of the "split secret" principle.
- By using the TKE workstation

  Globally, the TKE workstation provides the same Master Key administrative functions as the ISPF panels. However, it implements a highly secure environment in terms of authentication and authorization of the security officers, as well as for the confidentiality and integrity of data exchanged with the coprocessors.

  The split secret principle is still observed at the TKE, with the capability, among other things, of requiring critical operations to be confirmed by a second security officer.

Pass Phrase Initialization

The Passphrase Initialization method is recommended for a quick setup to allow testing of the coprocessor setup. The Master Keys generated with the passphrase can then be replaced using the ISPF panels, or the TKE workstation, when switching to production mode.

This section describes how to initialize an uninitialized domain with Master Keys generated using the Passphrase Initialization method. For further information about Master Key management using ISPF panels, refer to *z/OS Cryptographic Services Integrated Cryptographic Service Facility Administrator's Guide*, SA22-7521. For more information about about using the TKE workstation, refer to *z/OS Cryptographic Services ICSF Trusted Key Entry PCIX Workstation User's Guide*, SA23-2211.

As a first step, prior to initializing the domain's Master Keys, you may want to look at the coprocessor status by using the ICSF ISPF panels. In the ICSF main panel, select option **1 COPROCESSOR MGMT**, as shown in Figure A-17 on page 217.

```
HCR7750  -------------- Integrated Cryptographic Service Facility--------
OPTION ===>
Enter the number of the desired option.

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

Figure A-17 ICSF main ISPF panel

The installed coprocessors are displayed in the panel shown in Figure A-18.

```
------------------------ ICSF Coprocessor Management -------- Row 1 to 4 of 4
COMMAND ===>                                                  SCROLL ===> PAGE

 Select the coprocessors to be processed and press ENTER.
 Action characters are: A, D, E, K, R and S. See the help panel for details.

    COPROCESSOR      SERIAL NUMBER   STATUS
    -----------      -------------   ------
s   E01              94000264        ONLINE
s   E02              95001434        ONLINE
s   E03              95001437        ONLINE
.    F00                                 ACTIVE
****************************** Bottom of data ********************************
```

Figure A-18 Coprocessor Management panel

To select a coprocessor, enter s. ICSF displays the status of some internal registers in the domain the logical partition has access to, as shown in Figure A-19 on page 218.

The status displayed in Figure A-19 on page 218 shows that both the symmetric and asymmetric Master Keys in the coprocessor domain are empty and therefore need to be initialized.

```
---------------------- ICSF - Coprocessor Hardware Status --------------------

        CRYPTO DOMAIN: 1

REGISTER STATUS                  COPROCESSOR E01
                                                                        More:
+
 Crypto Serial Number          : 94000264
 Status                        : ONLINE
 Symmetric-Keys Master Key
   New Master Key register     : EMPTY
     Verification pattern      :
     Hash pattern              :
                               :
   Old Master Key register     : EMPTY
     Verification pattern      :
     Hash pattern              :
                               :
   Current Master Key register : EMPTY
     Verification pattern      :
     Hash pattern              :
                               :
 Asymmetric-Keys Master Key
   New Master Key register     : EMPTY
     Hash pattern              :
                               :
   Old Master Key register     : EMPTY
     Hash pattern              :
                               :
                               :
   Current Master Key register : EMPTY
     Hash pattern              :
```

Figure A-19 Coprocessor hardware status panel showing empty Master Keys

To initialize the Master Keys with the Passphrase Initialization method, select option **6 PPINIT** in the ICSF main panel, as shown in Figure A-20 on page 219.

```
HCR7750  -------------- Integrated Cryptographic Service Facility--------
OPTION ===>
Enter the number of the desired option.

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

Figure A-20 ICSF main panel

The Passphrase Initialization panel is displayed and the passphrase can be entered as shown in Figure A-21, along with the CKDS and PKDS dataset names so that these data sets can also be initialized with the Master Keys.

```
------------------- ICSF - Pass Phrase MK/KDS Initialization ---------------
COMMAND ===>

Enter your pass phrase and the names of the CKDS and PKDS:

   Pass Phrase (16 to 64 characters)
   ===> Enter the pass phrase here
   CKDS
   ===> ‘hlq.csfckds'
   PKDS
   ===> ‘hlq.csfpkds'
   Initialize the CKDS and PKDS? (Y/N) ===> Y
   Initialize new online coprocessors only ? (Y/N) ===> N


Press ENTER to process.
Press END   to exit to the previous menu.
```

Figure A-21 Passphrase Initialization panel

After the successful passphrase initialization, the status of the coprocessor domain is changed as shown in Figure A-22 on page 220.

```
---------------------- ICSF - Coprocessor Hardware Status --------------------
         CRYPTO DOMAIN: 1

REGISTER STATUS                  COPROCESSOR E01
                                                                        More:
+
 Crypto Serial Number          : 94000264
 Status                        : ACTIVE
 Symmetric-Keys Master Key
   New Master Key register     : EMPTY
     Verification pattern      :
     Hash pattern              :
                               :
   Old Master Key register     : EMPTY
     Verification pattern      :
     Hash pattern              :
                               :
   Current Master Key register : VALID
     Verification pattern      : 5B8EAE2289D07CF7
     Hash pattern              : EB74EC9A1FDBE39A
                               : D0518360AB317815
 Asymmetric-Keys Master Key
   New Master Key register     : EMPTY
     Hash pattern              :
                               :
   Old Master Key register     : EMPTY
     Hash pattern              :
                               :
                               :
   Current Master Key register : VALID
     Hash pattern              : 82F0E5C8849F2ECA
                               : 4C13B323C769691F
```

Figure A-22 Coprocessor hardware status panel showing valid Master keys initialized

B

SAF sample code

This appendix contains a complete code example for the SAF interface in Java.

The example illustrates the following tasks:

- Checking to see if the Security Server or a specific security server class is active
- Extracting the user ID in effect for the current running thread
- Checking the user ID in effect for access rights to a resource
- Authenticating a user ID and password
- Checking if a user ID is a member of a group
- Changing a user's password

© Copyright IBM Corp. 2008. All rights reserved.

Sample code

Example: B-1 SAF sample

```
import com.ibm.os390.security.*;

public class SAFClassesTest {

   public static void main(String[] args) {

      // check if Security Server is active
      if (PlatformSecurityServer.isActive())
         System.out.println("Security Server is active.");
      else {
         System.out.println("Error: Security Server is not active.");
         return;
      }

      // check if resource type 'FACILITY' is active
      if (PlatformSecurityServer.resourceTypeIsActive("FACILITY"))
         System.out.println("FACILITY is active");
      else {
         System.out.println("Error: FACILITY is not active");
         return;
      }

      // find out current user
      String currentUser = PlatformThread.getUserName();
      System.out.println("CurrentUser is:"+currentUser);

      // find out if current user has has READ access to the resource named
      // BPX.SERVER of resource type FACILITY
      PlatformReturned pr =
         PlatformAccessControl.checkPermission("FACILITY","BPX.SERVER",
         PlatformAccessLevel.READ );
      if (pr == null)
         System.out.println("User has READ access to the resource named BPX.SERVER
            of resource type FACILITY");
      else {
         System.out.println("An error ocurred ...");
         System.out.println( "success: " + pr.success +
               "\nerrno: " + pr.errno +
               "\nerrno2: " + pr.errno2 +
               "\nerrnoMsg: " + pr.errnoMsg );
      }

      pr = PlatformUser.authenticate(currentUser,"secret");
      if (pr == null)
      {
         System.out.println("Password check was successful");
      }
      else
      {
         System.out.println("An error ocurred ...");
         System.out.println( "success: " + pr.success +
```

```
                "\nerrno: " + pr.errno +
                "\nerrno2: " + pr.errno2 +
                "\nerrnoMsg: " + pr.errnoMsg );
        }

        boolean isMember = PlatformUser.isUserInGroup(currentUser,"SYS1");
        if (isMember)
        {
           System.out.println(currentUser + " is a member of SYS1");
        }
        else
        {
           System.out.println(currentUser + " is not a member of SYS1");
        }

        pr = PlatformUser.changePassword(currentUser,"secret","passwd");
        if (pr == null)
           System.out.println("Password change successful.\n");
        else
        {
           System.out.println("An error ocurred ...");
           System.out.println( "success: " + pr.success +
                 "\nerrno: " + pr.errno +
                 "\nerrno2: " + pr.errno2 +
                 "\nerrnoMsg: " + pr.errnoMsg );
        }
     }
}
```

C

JSec sample code

This appendix contains code examples for Java Security Administration API (JSec). The samples can also be downloaded using the following link:

ftp://ftp.software.ibm.com/eserver/zseries/zos/racf/jsec/JSec_Webpages.zip

The examples illustrate the following tasks:

- Creating a TSO user ID
- Creating a protected user ID
- Deleting a user ID
- Showing attributes for a user, group, and membership in HTML format
- Searching users and groups within the security repository

© Copyright IBM Corp. 2008. All rights reserved.

Creating a TSO user ID

Example C-1 illustrates how to create a TSO user CAT with password meow. It refers to the TSO RACF command ADDUSER.

Example: C-1 CreateTSOUserid

```
import com.ibm.eserver.zos.racf.userregistry.*;
import com.ibm.security.userregistry.*;
import javax.naming.directory.*;

public class CreateTSOUserid {

   public static void main(String[] args)
   {
      SecAdmin racfAdmin = null;
      User catuser = null;

      // Instantiate RACF_remote object with connection data:
      RACF_remote remote = new RACF_remote("ldap://wtsc60.itso.ibm.com:389",
         "simple",
         "IBMUSER",
         "SECRET",       // password during testing
         "o=itsoracf");

      // Create a new RACF_SecAdmin object.  This will create connection to RACF
      // database with authority of userid provided in RACF_remote object.
      try {
         racfAdmin = new RACF_SecAdmin(remote);
      } catch (SecAdminException e) {
         System.out.println("Unable to connect to specified RACF database.
         "+e.getMessage());
         return;
      }

      // Define the user attributes and create the user
      try {
         BasicAttributes ba = new BasicAttributes();
         BasicAttribute pwd = new BasicAttribute("base_password");
         pwd.add("meow");       // cat simply has to enter ‘meow’ to log on
         pwd.add("noexpired");
         ba.put(pwd);
         ba.put(new BasicAttribute("TSO"));
         catuser = (User)racfAdmin.createUser("cat", ba);
         System.out.println("You have successfully created TSO user cat.");
      } catch (SecAdminException e) {
         System.out.println("Unable to create user 'cat'. "+e.getMessage());
         return;
      }

      // Get the user attributes of the recently created user and display the
      // BASE_PASSWORD attribute
      try {
         BasicAttributes u_at = catuser.getAttributes();
         System.out.println(u_at.get("BASE_PASSWORD"));
```

```
      } catch (SecAdminException e) {
         System.out.println("Error retrieving attributes "+e.getMessage());
         return;
      }
   }
}
```

Creating a protected user ID

Example C-2 illustrates s how to create a protected user. It refers to the TSO RACF command ADDUSER.

Example: C-2 CreateProtectedUserid

```
import com.ibm.eserver.zos.racf.userregistry.*;
import com.ibm.security.userregistry.*;
import javax.naming.directory.*;

public class CreateProtectedUserid {

   public static void main(String[] args)
   {
      SecAdmin racfAdmin = null;
      User protect = null;

      //  Instantiate RACF_remote object with connection data:
      RACF_remote remote = new RACF_remote("ldap://wtsc60.itso.ibm.com:389",
         "simple",
         "IBMUSER",
         "SECRET",       // password during testing
         "o=itsoracf");

      // Create a new RACF_SecAdmin object.  This will create connection to RACF
      // database with authority of userid provided in RACF_remote object.
      try {
         racfAdmin = new RACF_SecAdmin(remote);
      } catch (SecAdminException e) {
         System.out.println("Unable to connect to specified RACF database.
         "+e.getMessage());
         return;
      }
      // Define the user attributes and create the user
      try {
         BasicAttributes ba = new BasicAttributes();
         BasicAttribute pwd = new BasicAttribute("base_password");
         pwd.add("nopassword");
         ba.put(pwd);
         protect = racfAdmin.createUser("protect", ba);
         System.out.println("Successfully created userid 'protect'.");
      } catch (SecAdminException e) {
         System.out.println("Unable to create user 'protect'. "+e.getMessage());
         return;
      }
```

```
      // Get the user attributes of the recently created user and display the
      // BASE_PASSWORD attribute
      try {
         BasicAttributes prot_at = protect.getAttributes();
         System.out.println(prot_at.get("BASE_PASSWORD"));
      } catch (SecAdminException e) {
         System.out.println("Error retrieving attributes "+e.getMessage());
         return;
      }
   }
}
```

Deleting a user ID

Example C-3 illustrates how to delete a user from the security repository. It refers to the TSO RACF command REMOVE. To match with Example C-1 on page 226 and Example C-2 on page 227, users `cat` and `protect` are deleted here.

Example: C-3 DeleteUserid

```
import com.ibm.eserver.zos.racf.userregistry.*;
import com.ibm.security.userregistry.*;
import javax.naming.*;
import javax.naming.directory.*;

public class DeleteUserid {

   public static void main(String[] args)
   {
      SecAdmin racfAdmin = null;
      User protect = null;

      //  Instantiate RACF_remote object with connection data:
      RACF_remote remote = new RACF_remote("ldap://wtsc60.itso.ibm.com:389",
         "simple",
         "IBMUSER",
         "SECRET",       // password during testing
         "o=itsoracf");

      // Create a new RACF_SecAdmin object.  This will create connection to RACF
      // database with authority of userid provided in RACF_remote object.
      try {
         racfAdmin = new RACF_SecAdmin(remote);
      } catch (SecAdminException e) {
         System.out.println("Unable to connect to specified RACF database.
         "+e.getMessage());
         return;
      }
      // Now delete the userids we just created, so the testcase can be run
      // repeatedly.
      try {
         racfAdmin.deleteUser("protect");
         System.out.println("Successfully deleted userid 'protect'.");
         racfAdmin.deleteUser("cat");
```

```
            System.out.println("Successfully deleted userid 'cat'.");
         } catch (Exception e) {
            System.out.println("Exception deleting user protect: "+e.getMessage());
         }
      }
}
```

Showing attributes

Example C-4 refers to the TSO RACF command LISTUSER. This code will extract the user attributes for user `Martina` and the group attributes and membership attributes for group `SYS1`. The output is a string containing an HTML table that should be displayed in a Web browser.

Example: C-4 ShowAttributes

```
import com.ibm.eserver.zos.racf.userregistry.*;
import com.ibm.security.userregistry.*;

public class ShowAttributes {

   public static void main(String[] args)
   {
      SecAdmin racfAdmin = null;
      User ibmuser = null;
      UserGroup ibmgroup = null;

      //  Instantiate RACF_remote object with connection data:
      RACF_remote remote = new RACF_remote("ldap://9.12.4.18:389",
            "simple",
            "IBMUSER",
            "secret",       // password during testing
            "o=itsoracf");

      // Create a new RACF_SecAdmin object.  This will create connection to RACF
      // database with authority of userid provided in RACF_remote object.
      try {
         racfAdmin = new RACF_SecAdmin(remote);
      } catch (SecAdminException e) {
         System.out.println("Unable to connect to specified RACF database.
            "+e.getMessage());
         return;
      }

      try {
         ibmuser = racfAdmin.getUser("MARTINA");
         ibmgroup = racfAdmin.getGroup("SYS1");
      } catch (SecAdminException e) {
         e.printStackTrace();
      }

      System.out.print(" RACF_User.attributesHTML, RACF_Group.attributesHTML and
         RACF_Group.membershipAttributesHTML. ");
```

```
      System.out.println(" The output should be displayed in a web browser.");
      System.out.println(" ");

      System.out.println("-------------  Start of output from
         RACF_User.attributesHTML------------\n");
      System.out.println(((RACF_User) ibmuser).attributesHTML());

      System.out.println("-------------  End of output from
         RACF_User.attributesHTML------------\n");

      System.out.println("-------------  Start of output from
         RACF_Group.attributesHTML------------");
      System.out.println(((RACF_Group) ibmgroup).attributesHTML());

      System.out.println("-------------  End of output from
         RACF_Group.attributesHTML------------");
      System.out.println("-------------  Start of output from
         RACF_Group.membershipAttributesHTML------------");
      System.out.println(((RACF_Group) ibmgroup).membershipAttributesHTML());

      System.out.println("-------------  End of output from
         RACF_Group.membershipAttributesHTML------------");

   }
}
```

Search users and groups

The sample code shown in Example C-5 can be used to search a RACF database for users or groups that begin with a particular string. The default of an empty string shown here will return all users and groups. However, you could use, for example use the following command to find all users and groups beginning with the letter B:

```
java SearchUsersAndGroups b
```

Example: C-5 SearchUsersAndGroups

```
import java.util.Hashtable;
import com.ibm.security.userregistry.*;
import com.ibm.eserver.zos.racf.userregistry.*;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchControls;
import javax.naming.directory.SearchResult;
import javax.naming.*;

public class SearchUsersAndGroups {

   public static void main(String[] args)
   {
      String search_string;
      InitialDirContext ctx = null;
      NamingEnumeration answer = null;

      if (args.length > 0)
```

```
{
   search_string = args[0];
}
else search_string = "";

// We define a RACF_remote object not to get a RACF_SecAdmin object, but
// simply because this is how we have defined our connection information in
// all the other samples.   We use the RACF_remote object in such a way that
// the rest of the code could be cut and pasted into code that was using
// JSec.

RACF_remote remote = new RACF_remote("ldap://9.12.4.18:389",
      "simple",
      "IBMUSER",
      "secret",       // password during testing
      "o=itsoracf");

// The following code is using LDAP/SDBM to connect to RACF

String ldap_suffix = remote.getConnect_suffix();  // diff for each system

try
{
   SecAdmin racfAdmin = new RACF_SecAdmin(remote);
   if (racfAdmin != null)
   {
      Hashtable hashtable = new Hashtable(7);
      hashtable.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.ldap.LdapCtxFactory");
      hashtable.put(Context.PROVIDER_URL, remote.getConnect_url() );
      // if second parm to RACF_REMOTE is 'secure' then use 'ssl' here
      hashtable.put(Context.SECURITY_AUTHENTICATION, "simple");
      String dn = "racfid=" + remote.getConnect_principal() +
         ",profiletype=user," + remote.getConnect_suffix();
      hashtable.put(Context.SECURITY_PRINCIPAL, dn);
      hashtable.put(Context.SECURITY_CREDENTIALS,
         remote.getConnect_credentials());

      try {
         // Create initial context
         ctx = new InitialDirContext(hashtable);
      } catch (NamingException e) {
         System.out.println("Error initially connecting to
            LDAP/SDBM."+e.getMessage());
      }

      // Initialize some parameters we'll need

      String[] attrIDs = {"racfid"};
      SearchControls ctls = new SearchControls();
      ctls.setReturningAttributes(attrIDs);

      // Specify the search scope
      ctls.setSearchScope(SearchControls.SUBTREE_SCOPE);
```

```
            String filter = "racfid="+search_string+"*";
            System.out.println("filter looks like: "+filter);

            // the specific code for searching for users

            try {
               answer = ctx.search("profiletype=user,"+ldap_suffix, filter,ctls);
            } catch (javax.naming.NamingException ne) {
               String e_text = ne.getMessage();
               if (e_text.toUpperCase().indexOf("NO ENTRIES MEET SEARCH CRITERIA")
                  > -1) answer = null;
               else throw ne;
            }

            // Display any userids we find

            if (answer != null)
            {
               while (answer.hasMoreElements()) {
                  SearchResult sr = (SearchResult)answer.next();
                  System.out.println("Userid: " + deLDAP(sr.getName()));
               }
            }
            else System.out.println("System didn't find matching user");

            // the specific code for searching for groups

            try
            {
               answer = ctx.search("profiletype=group,"+ldap_suffix, filter,ctls);
            }
            catch (javax.naming.NamingException ne)
            {
               String e_text = ne.getMessage();
               if (e_text.toUpperCase().indexOf("NO ENTRIES MEET SEARCH CRITERIA")
                  > -1) answer = null;
               else throw ne;
            }

            // Display any groupnames we find

            if (answer != null)
            {
               while (answer.hasMoreElements()) {
                  SearchResult sr = (SearchResult)answer.next();
                  System.out.println("Group: " + deLDAP(sr.getName()));
               }
            }
            else System.out.println("System didn't find matching group");

         }    // end if racf_admin is not null
      } catch (Exception e) {
         System.out.println("Exception in SearchUsersAndGroups.java " +
            e.getMessage() + "\n");
         e.printStackTrace();
```

```
      }

   }

   /**
    *
    * @param in String that may or may not be a userid or groupname in LDAP DN
    * format
    * @return   String that is striped of any LDAP stuff
    *
    * example: in: "racfid=IBMUSER,profiletype=USER,o=racfdb,c=us"
    *          returns "IBMUSER"
    */
   protected static String deLDAP(String in)
   {

      if (in == null)             // protect against bad input
         return in;

      String out;
      String lower_in = in.toLowerCase();

      String racfid = "racfid=";
      int pos = lower_in.indexOf("racfid=");
      if (pos > -1)
      {
         int comma = in.indexOf(',',pos);
         if (comma > -1) out = in.substring(pos+racfid.length(),comma);
         else out = in.substring(pos+racfid.length());
         return out;
      }
      else return in;
   }

}
```

D

JSec attributes

This appendix provides an overview of all Java Security Administration API (JSec) attributes that can be returned for a user, a group, or for membership in a group by RACF.

It also includes a description of each attribute.

© Copyright IBM Corp. 2008. All rights reserved.

User attributes

Table D-1 lists and describes the 135 user attributes that can be returned from RACF using the JSEC API.

Table D-1 User attributes

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
BASE_ADSP	All permanent tape and DASD data sets created by user are automatically RACF-protected by discrete profiles.	Yes	No	Yes	No
BASE_AUDITOR	Indicates user has full responsibility for auditing the use of system resources, and is able to control the logging of detected accesses to any RACF-protected resources during RACF authorization checking and accesses to the RACF database.	Yes	No	Yes	No
BASE_CATEGORY	Names of installation-defined security categories, which must be defined as members of the CATEGORY profile in the SECDATA class.	Yes	No	No	Yes
BASE_CLAUTH	Classes in which user is allowed to define profiles to RACF for protection. Classes can be USER, and any resource classes defined in the class descriptor table.	Yes	No	No	Yes
BASE_CREATED	The date this user was defined to RACF.	No	No	No	No
BASE_DATA	Up to 255 characters of installation-defined data.	Yes	No	No	No
BASE_DAYS	Days of week user is allowed access system from a terminal - Allowed values: ANYDAY, WEEKDAYS, SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY.	Yes	No	No	Yes
BASE_DFLTGRP	Name of RACF group that is the default group for user.	Yes	No	No	No
BASE_GRPACC	Indicates that any group data sets protected by DATASET profiles defined by this user are automatically accessible to other users in the group.	Yes	No	Yes	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
BASE_LAST-ACCESS	The date and time the user last entered the system.	No	No	No	No
BASE_MODEL	Name of discrete data set profile used as model when new data set profiles are created that have this user ID as the high-level qualifier.	Yes	No	No	No
BASE_NAME	User's name - a name associated with user ID - maximum of 20 characters.	Yes	No	No	No
BASE_OPERATIONS	User has OPERATIONS segment.	Yes	No	Yes	No
BASE_OWNER	RACF user ID or groupname of owner of this user ID.	Yes	No	No	No
BASE_PASS-INTERVAL	The password change interval (in number of days).	No	No	No	No
BASE_PASSDATE	The date the user's password was last updated.	No	No	No	No
BASE_PASSWORD	When setting, value is new password. When getting, simply indicates if user has password or is restricted user ID (no password).	Yes	No	No	No
BASE_PASSWORD_ENV	User's password, encrypted in PKCS#7 envelope. Only returned if password enveloping has been set up and user ID that authenticated in RACF_SecAdmin constructor has digital certificate on IRR.PWENV.KEYRING keyring.	No	No	No	No
BASE_PHRASE	The user's pass phrase. A text string of 14-100 characters.	Yes	No	No	No
BASE_PHRASE_CHANGE_DATE	Date user's pass phrase was last changed.	No	No	No	No
BASE_RESTRICTED	Indicates global access checking is bypassed when resource access checking is performed for this user, and neither ID(*) on the access list or the UACC will allow access.	Yes	No	Yes	No
BASE_RESUME	Date when RACF will resume allowing the user access to the system. Date in format mm/dd/yy.	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
BASE_REVOKE	Date when RACF will stop allowing the user access to the system. Date in format mm/dd/yy.	Yes	No	No	No
BASE_REVOKED	User's access to the system is currently revoked.	No	No	Yes	No
BASE_SECLABEL	Installation-defined security label that is user's default security label.	Yes	No	No	No
BASE_SECLEVEL	User's security level, where seclevel-name is an installation-defined name that must be a member of the SECLEVEL profile in the SECDATA class.	Yes	No	No	No
BASE_SPECIAL	Indicates user is allowed to issue all RACF commands with all operands except operands that require AUDITOR attribute.	Yes	No	Yes	No
BASE_TIME	Time of day user is allowed access system from a terminal. Format is start-time:end-time and each time's format is hhmm, where hh is the hour (00-23) and mm is the minutes (00-59). But 0000 is not a valid time value. If start-time is greater than end-time, interval spans midnight.	Yes	No	No	No
BASE_UAUDIT	Indicates RACF is to log all RACROUTE REQUEST=AUTH and RACROUTE REQUEST=FASTAUTH services eligible for logging, and all RACROUTE REQUEST=DEFINE services issued for the user, and all RACF commands (except SEARCH, LISTDSD, LISTGRP, LISTUSER, and RLIST) issued by user.	Yes	No	Yes	No
BASE_USERID	User ID.	No	No	No	No
CICS	User has CICS segment.	Yes	Yes	Yes	No
CICS_OPCLASS	Numbers 1-24, representing classes assigned to this operator to which basic mapping support (BMS) messages are to be routed.	Yes	No	No	Yes
CICS_OPIDENT	A 1-3 character identification of the operator for use by BMS.	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
CICS_OPPRTY	Number from 0-255 that represents the priority of the operator.	Yes	No	No	No
CICS_RSLKEY	Specifies the resource security level (RSL) keys assigned to the user. Numbers from 1 - 24 or 0 (meaning no RSL keys are assigned to the user) or 99 (meaning 1 through 24 are assigned to the user).	Yes	No	No	Yes
CICS_TIMEOUT	Time, in hours and minutes, that the operator is allowed to be idle before being signed off. The value for TIMEOUT can be entered in the form m, mm, hmm, or hhmm, where the value for m or mm is 00-59, or 00-60 if h or hh is not specified or is specified as 0 or 00.	Yes	No	No	No
CICS_TSLKEY	Specifies the transaction security level (TSL) keys assigned to the user. Numbers from 1 - 64 or 0 (meaning no TSL keys are assigned to the user) or 99 (meaning 1 through 64 are assigned to the user).	Yes	No	No	Yes
CICS_XRFSOFF	Indicates whether the user is signed off by CICS when an XRF takeover occurs. Valid values 'FORCE', 'NOFORCE'.	Yes	No	No	No
DCE	User has DCE segment.	Yes	Yes	Yes	No
DCE_AUTOLOGIN	Indicates z/OS UNIX DCE is to log this user into z/OS UNIX DCE automatically.	Yes	No	Yes	No
DCE_DCENAME	The DCE principal name defined for this RACF user in the DCE registry. 1 - 1023 characters.	Yes	No	No	No
DCE_HOMECELL	The DCE cell name defined for this RACF user. 1 - 1023 characters. RACF checks that the HOMECELL name entered has a prefix of either /.../ or /.:/	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
DCE_HOMEUUID	The DCE universal unique identifier (UUID) for the cell that this user is defined to. The UUID is a 36-character string that consists of numeric and hexadecimal characters. This string must have the delimiter character (-) in positions 9, 14, 19, and 24. The general format for the UUID string is xxxxxxxx-xxxx-xxxx-xxxx-xxxxxx xxxxxx, in which x represents a valid numeric or hexadecimal character.	Yes	No	No	No
DCE_UUID	The DCE universal unique identifier (UUID) of the DCE principal defined in DCENAME. The UUID is a 36-character string that consists of numeric and hexadecimal characters. This string must have the delimiter character (-) in positions 9, 14, 19, and 24. The general format for the UUID string is xxxxxxxx-xxxx-xxxx-xxxx-xxxxxx xxxxxx, in which x represents a valid numeric or hexadecimal character.	Yes	No	No	No
DFP	User has DFP segment.	Yes	Yes	Yes	No
DFP_DATAAPPL	An 8-character DFP data application identifier.	Yes	No	No	No
DFP_DATACLAS	The default data class. 1-8 characters.	Yes	No	No	No
DFP_MGMTCLAS	The default management class. 1-8 characters.	Yes	No	No	No
DFP_STORCLAS	The default storage class. 1-8 characters.	Yes	No	No	No
EIM	User has EIM segment.	Yes	Yes	Yes	No
EIM_LDAPPROF	Name of a profile in the LDAPBIND class. The profile in the LDAPBIND class contains the name of an EIM domain and the bind information required to establish a connection with the EIM domain. 1-246 characters.	Yes	No	No	No
KERB	User has KERB segment.	Yes	Yes	Yes	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
KERB_ENCRYPT	ENCRYPT values are used to specify which keys are allowed for use based on the encryption algorithm used to generate them. Default values will be provided for any values not specified. Examples: 'DES','DES3' and 'DESD'.	Yes	No	No	Yes
KERB_KERBNAME	User's local kerberos-principal-name, may contain any characters except '@'. Must not be qualified with a realm name. However, RACF verifies that the local principal name, when fully qualified with the name of the local realm: '/.../local_realm_name/principal_name' does not exceed 240 characters.	Yes	No	No	No
KERB_KEYVERSION	Current Network Authentication Service key version.	No	No	No	No
KERB_MAXTKTLFE	The max-ticket-life in seconds, and is represented by a numeric value between 1 and 2147483647.	Yes	No	No	No
LANGUAGE	User has LANGUAGE segment.	Yes	Yes	Yes	No
LANGUAGE_PRIMARY	User's primary language. Specified as either an installation-defined name of a currently active language (maximum of 24 characters) or one of the language codes (three characters in length) for a language installed on your system.	Yes	No	No	No
LANGUAGE_SECONDARY	User's secondary language. Specified as either an installation-defined name of a currently active language (maximum of 24 characters) or one of the language codes (three characters in length) for a language installed on your system.	Yes	No	No	No
LNOTES	User has LNOTES segment.	Yes	Yes	Yes	No
LNOTES_SNAME	Lotus Notes for z/OS short-name of the user. 1-64 characters, consisting of alphanumeric characters or '&', '-', '.', '_', and a blank.	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
NDS	User has NDS segment.	Yes	Yes	Yes	No
NDS_UNAME	Novell® Directory Services for OS/390 user-name of the user. 1-246 characters excluding the following characters: '*', '+', '|', '=', ',', '"', '`', '/', ':', ';', '¢', '[', ']'	Yes	No	No	No
NETVIEW	User has NETVIEW segment.	Yes	Yes	Yes	No
NETVIEW_CONSNAME	Specifies the default master console station (MCS) console name used for this operator. 1 - 8 character name.	Yes	No	No	No
NETVIEW_CTL	Indicates whether a security check is performed for this NetView® operator when they try to use a span or try to do a cross-domain logon. Allowed values 'GENERAL','GLOBAL' or 'SPECIFIC'.	Yes	No	No	No
NETVIEW_DOMAINS	Specifies the identifiers of NetView programs in another NetView domain where this operator can start a cross-domain session. Each identifier is 1-5 characters, with valid characters being 0-9, A-Z, #, $, or @.	Yes	No	No	Yes
NETVIEW_IC	The command or command list (up to 255 characters) to be processed by NetView for this operator when this operator logs on to NetView.	Yes	No	No	No
NETVIEW_MSGRECVR	Indicates this operator is to receive unsolicited messages that are not routed to a specific NetView operator.	Yes	No	Yes	No
NETVIEW_NGMFADMN	Indicates a NetView operator has administrator authority to the NetView Graphic Monitor Facility (NGMF).	Yes	No	Yes	No
NETVIEW_OPCLASS	NetView scope classes for which the operator has authority. Each class is a number from 1 to 2040.	Yes	No	No	Yes
OMVS	User has OMVS segment.	Yes	Yes	Yes	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
OMVS_ASSIZEMAX	The RLIMIT_AS hard limit resource value (maximum address space region size) that processes receive when dubbed a process. Integer value between 10485760 and 2147483647.	Yes	No	No	No
OMVS_CPUTIMEMAX	The RLIMIT_CPU hard limit (maximum) resource value that user's z/OS UNIX processes receive when they are dubbed a process. Numeric value between 7 and 2147483647, indicates the cpu-time in seconds that a process is allowed to use.	Yes	No	No	No
OMVS_FILEPROCMAX	Maximum number of files this user is allowed to have concurrently active or open. Numeric value between 3 and 524287.	Yes	No	No	No
OMVS_HOME	User's z/OS UNIX initial directory pathname, 1-1023 characters.	Yes	No	No	No
OMVS_MEMLIMIT	Specifies the maximum number of bytes of nonshared memory that can be allocated by the user. The nonshared-memory-size you define to RACF is a numeric value between 0 and 16777215, followed by the letter M, G, or T. The M, G, or T letter indicates the multiplier to be used. (M=Megabyte, G Gigabyte, T=Terabyte, P=Petabyte). Maximum value is 16383P.	Yes	No	No	No
OMVS_MMAPAREAMAX	Maximum amount of data space storage, in pages, that can be allocated by the user for memory mappings of HFS files. Numeric value between 1 and 16,777,216.	Yes	No	No	No
OMVS_PROCUSERMAX	Maximum number of processes user is allowed to have active at the same time, regardless of how the process became a z/OS UNIX process. Numeric value between 3 and 32767.	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
OMVS_PROGRAM	Specifies the PROGRAM pathname (z/OS UNIX shell program). The first program started when TSO/E command OMVS is entered or when a batch job is started using the BPXBATCH program, 1-1023 characters.	Yes	No	No	No
OMVS_SHMEMMAX	The maximum number of bytes of shared memory that can be allocated by user. The shared-memory-size you define to RACF is a numeric value between 1 and 16,777,215, followed by the letter M, G, T, or P. The M, G, T, or P letter indicates the multiplier to be used. (M=Megabyte, G Gigabyte, T=Terabyte, P=Petabyte). Maximum value is 16383P.	Yes	No	No	No
OMVS_THREADSMAX	Maximum number of pthread_create threads, including those running, queued, and exited but not detached, that the user can have concurrently active. Numeric value between 0 and 100000.	Yes	No	No	No
OMVS_UID	The UID, numeric value between 0 and 2147483647. 'AUTOUID' value can be used when BPX.NEXT.USER profile is defined in the FACILITY class. SHARED value can be used when the SHARED.IDS profile in the UNIXPRIV class is defined. See *z/OS Security Server RACF Security Administrator's Guide* for details.	Yes	No	No	Yes
OPERPARM	User has OPERPARM segment.	Yes	Yes	Yes	No
OPERPARM_ALTGRP	The console group used in recovery. 1-8 characters, with valid characters being 0-9, A-Z, #, $, or @.	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
OPERPARM_AUTH	Authority this console has to issue operator commands. Valid values, 'MASTER', 'ALL', 'INFO' (these three cannot be combined with other values) and 'CONS', 'IO' and 'SYS'. See ALTUSER in *z/OS Security Server RACF Command Language Reference* for more detailed description.	Yes	No	No	Yes
OPERPARM_AUTO	Indicates the extended console can receive messages that have been automated by the Message Processing Facility (MPF) in the sysplex.	Yes	No	Yes	No
OPERPARM_CMDSYS	Indicates the system to which commands issued from this console are to be sent. 1-8 characters, with valid characters being A-Z, 0-9, @ (X'7C'), # (X'7B'), and $ (X'5B'). If * is specified, commands are processed on the local system where the console is attached.	Yes	No	No	No
OPERPARM_DOM	Indicates whether this console receives delete operator message (DOM) requests. Allowed values 'NORMAL','ALL','NONE'.	Yes	No	No	No
OPERPARM_HC	Indicates this console is to receive hardcopy messages.	Yes	No	Yes	No
OPERPARM_INTIDS	Indicates this console is to receive messages directed to console ID 0 (the internal console).	Yes	No	Yes	No
OPERPARM_KEY	A 1-8 byte character name that can be used to display information for all consoles with the specified key by using the MVS command DISPLAY CONSOLES,KEY. Valid characters are A-Z, 0-9, # (X'7B'), $ (X'5B'), or @ (X'7C').	Yes	No	No	No
OPERPARM_LEVEL	Specifies the messages that this console is to receive. Can be a list of R, I, CE, E, IN, NB or ALL. If you specify ALL, you cannot specify R, I, CE, E, or IN.	Yes	No	No	Yes

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
OPERPARM_LOGCMDRESP	Indicates if command responses are to be logged. Value of 'SYSTEM' specifies that command responses are logged in the hardcopy log. Value of 'NO' specifies that command responses are not logged.	Yes	No	No	No
OPERPARM_MFORM	Specifies the format in which messages are displayed at the console. Can be a combination of J, M, S, T, and X.	Yes	No	No	Yes
OPERPARM_MIGID	Indicates a 1-byte migration ID is assigned to this console.	Yes	No	Yes	No
OPERPARM_MONITOR	Specifies which information should be displayed when jobs, TSO sessions, or data set status are being monitored. Allowed values, 'JOBNAMES' OR 'JOBNAMEST' (mutually exclusive), 'SESS' or 'SESST' (mutually exclusive) or 'STATUS'. See ALTUSER in *z/OS Security Server RACF Command Language Reference* for a more detailed description.	Yes	No	No	Yes
OPERPARM_MSCOPE	Specifies the systems from which this console can receive messages that are not directed to a specific console. Each system-name can be any combination of A-Z, 0-9, #, $, or @. A name of '*' indicates the system on which the console is currently active.	Yes	No	No	Yes
OPERPARM_ROUTCODE	Routing codes of messages this console is to receive. Valid values are 'ALL' or one or more routing codes or sequences of routing codes. The routing codes can be list of n and n1:n2, where n, n1, and n2 are integers 1-128, and n2 is greater than n1.	Yes	No	No	Yes
OPERPARM_STORAGE	Amount of storage in the TSO/E user's address space that can be used for message queuing to this console. Valid values are 1 - 2000.	Yes	No	No	No
OPERPARM_UD	Indicates that this console is to receive undelivered messages.	Yes	No	Yes	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
OPERPARM_UNKNIDS	Indicates this console is to receive messages directed to console ID 0 (the internal console).	Yes	No	Yes	No
OVM	User has OVM segment.	Yes	Yes	Yes	No
OVM_FSROOT	The pathname for the file system root. 1 - 1023 characters.	Yes	No	No	No
OVM_HOME	The initial directory pathname. 1 - 1023 characters.	Yes	No	No	No
OVM_PROGRAM	Specifies the PROGRAM pathname. 1 - 1023 characters. First program started when the OPENVM SHELL command is entered.	Yes	No	No	No
OVM_UID	OpenExtensions VM user identifier, UID. Numeric value between 0 and 2147483647.	Yes	No	No	Yes
PROXY	User has PROXY segment.	Yes	Yes	Yes	No
PROXY_BINDDN	The distinguished name (DN) which the z/OS LDAP Server will use when acting as a proxy on behalf of a requester. 1 - 1023 characters.	Yes	No	No	No
PROXY_BINDPW	Password which the z/OS LDAP Server will use when acting as a proxy on behalf of a requester. 1 - 128 characters.	Yes	No	No	No
PROXY_LDAPHOST	The URL of the LDAP server which the z/OS LDAP Server will contact when acting as a proxy on behalf of a requester. The URL should be in a format such as ldap://123.45.6:389 10-1023 characters. A valid URL must start with either ldap:// or ldaps:// and is not case-sensitive.	Yes	No	No	No
TSO	User has TSO segment.	Yes	Yes	Yes	No
TSO_ACCTNUM	User's default TSO account number when logging on through the TSO/E logon panel (1-39 characters).	Yes	No	No	No
TSO_COMMAND	Command to be run during TSO/E logon (1 - 80 characters).	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
TSO_DEST	Default destination to which the user can route dynamically allocated SYSOUT data sets. The specified value must be 1-7 alphanumeric characters, beginning with an alphabetic or national character.	Yes	No	No	No
TSO_HOLDCLASS	User's default hold class. The specified value must be 1 alphanumeric character, excluding national characters.	Yes	No	No	No
TSO_JOBCLASS	Specifies the user's default job class. The specified value must be 1 alphanumeric character, excluding national characters.	Yes	No	No	No
TSO_MAXSIZE	Maximum region size user can request at logon. Number of 1024-byte units of virtual storage that TSO can create for the user's private address space. Integer between 0 and 65535 (inclusive) if database is shared with any MVS systems, or 0 through 2096128 if not shared.	Yes	No	No	No
TSO_MSGCLASS	User's default message class. The specified value must be 1 alphanumeric character, excluding national characters.	Yes	No	No	No
TSO_PROC	Name of the user's default logon procedure when logging on through the TSO/E logon panel. The name must be 1-8 alphanumeric characters and begin with an alphabetic character.	Yes	No	No	No
TSO_SECLABEL	User's security label if the user specifies one on the TSO logon panel.	Yes	No	No	No
TSO_SIZE	Region size - number of 1024-byte units of virtual storage available in user's private address space at logon when user does not request a region size at logon. Integer between 0 and 65535 (inclusive) if database is shared with any MVS systems, or 0 through 2096128 if not shared.	Yes	No	No	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
TSO_SYSOUTCLASS	User's default SYSOUT class. The specified value must be 1 alphanumeric character, excluding national characters.	Yes	No	No	No
TSO_UNIT	Default name of a device or group of devices that a procedure uses for allocations. The specified value must be 1-8 alphanumeric characters.	Yes	No	No	No
TSO_USERDATA	Optional installation data, 4 characters where valid characters are 0 through 9 and A through F.	Yes	No	No	No
WORKATTR	User has WORKATTR segment.	Yes	Yes	Yes	No
WORKATTR_WAACCNT	An account number for APPC/MVS processing. 1 to 255 characters.	Yes	No	No	No
WORKATTR_WAADDR1	Address Line 1 that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No
WORKATTR_WAADDR2	Address Line 2 that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No
WORKATTR_WAADDR3	Address Line 3 that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No
WORKATTR_WAADDR4	Address Line 4 that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No
WORKATTR_WABLDG	Building that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No
WORKATTR_WADEPT	Department that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No
WORKATTR_WANAME	Name of the user that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No
WORKATTR_WAROOM	Room that SYSOUT information is to be delivered to. 1 to 60 characters.	Yes	No	No	No

Group attributes

Table D-2 lists and describes the 18 group attributes that can be returned from RACF using JSEC API.

Table D-2 Group attributes

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
BASE_CREATED	The date this group was defined to RACF.	No	No	No	No
BASE_DATA	Up to 255 characters of installation-defined data.	Yes	No	No	No
BASE_GROUPNAME	Name of the group.	No	No	No	No
BASE_MODEL	Name of a data set profile that RACF is to use as a model when new data set profiles are created that have groupname as the high-level qualifier.	Yes	No	No	No
BASE_OWNER	RACF user ID or groupname of owner of this group.	Yes	No	No	No
BASE_SUBGROUPS	Groups that have this group as their superior group.	No	No	No	Yes
BASE_SUPGROUP	Name of the RACF-defined group that is the superior group for this group.	Yes	No	No	No
BASE_TERMUACC	Indicates during terminal authorization checking, RACF is to allow the use of the universal access authority for a terminal when it checks whether a user in the group is authorized to access a terminal.	Yes	No	Yes	No
BASE_UNIVERSAL	Specifies that this is a universal group that allows an effectively unlimited number of users to be connected to it for the purpose of resource access.	Yes	No	Yes	No
DFP	Group has DFP segment.	Yes	Yes	Yes	No
DFP_DATAAPPL	An 8-character DFP data application identifier.	Yes	No	No	No
DFP_DATACLAS	The default data class. 1-8 characters.	Yes	No	No	No
DFP_MGMTCLAS	The default management class. 1-8 characters.	Yes	No	No	No
DFP_STORCLAS	The default storage class. 1-8 characters.	Yes	No	No	No
OMVS	Group has OMVS segment.	Yes	Yes	Yes	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
OMVS_GID	The group ID, GID, numeric value between 0 and 2147483647. 'AUTOGID' value can be used when BPX.NEXT.USER profile is defined in the FACILITY class. SHARED value can be used when the SHARED.IDS profile in the UNIXPRIV class is defined. See *z/OS Security Server RACF Security Administrator's Guide* for details.	Yes	No	No	Yes
OVM	Group has OVM segment.	Yes	Yes	Yes	No
OVM_GID	OpenExtensions VM group identifier. The GID is a numeric value between 0 and 2147483647.	Yes	No	No	No

Membership attributes

Table D-3 lists and describes the 14 membership attributes that can be returned from RACF using JSEC API.

Table D-3 Membership attributes

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
BASE_ADSP	Indicates when user is connected to this group, all permanent tape and DASD data sets created by the user are RACF-protected by discrete profiles.	Yes	No	Yes	No
BASE_AUDITOR	Indicates user is to have the group-AUDITOR attribute when connected to this group.	Yes	No	Yes	No
BASE_AUTHORITY	Specifies the level of authority user is to have in the group. Valid group authority values are 'USE', 'CREATE', 'CONNECT', and 'JOIN'.	Yes, Not Deletable	No	No	No
BASE_CONNECT-DATE	Date user was added to group.	No	No	No	No
BASE_CONNECTS	Number of times user has entered the system with this group as the current connect group.	No	No	No	No
BASE_GRPACC	Indicates when user is connected to this group, any group data sets defined by the user are automatically accessible to other users in the group.	Yes	No	Yes	No

Attribute name	Description	Modifiable	Segment	Boolean attribute	Multi-value attribute
BASE_LAST-CONNECT	Date user last entered the system using this group as the current connect group.	No	No	No	No
BASE_OPERATIONS	Indicates user is to have the group-OPERATIONS attribute when connected to this group. The group-OPERATIONS user has authorization to do maintenance operations on all RACF-protected DASD data sets, tape volumes, and DASD volumes within the scope of the group unless the access list for a resource specifically limits the OPERATIONS user to an access authority that is less than the operation requires.	Yes	No	Yes	No
BASE_OWNER	RACF-defined user or group to be assigned as the owner of the membership (connect profile). Defaults to the user who added user to group.	Yes, Not Deletable	No	No	No
BASE_RESUME	Date when user's membership in the group will be restored or resumed.	Yes	No	No	No
BASE_REVOKE	Date when user's membership in the group will be revoked.	Yes	No	No	No
BASE_REVOKED	User's membership to the group is currently revoked.	No	No	Yes	No
BASE_SPECIAL	User is to have the group-SPECIAL attribute when connected to this group.	Yes	No	Yes	No
BASE_UACC	Default value for the universal access authority for new resource profiles the user defines while connected to the group. Valid values are: ALTER, CONTROL, UPDATE, READ, and NONE.	Yes	No	No	No

E

EIM example setup program

This appendix provides a sample setup program to administer EIM. It also contains an authentication program and a main program.

© Copyright IBM Corp. 2008. All rights reserved.

Setup program: EimJavaSetup.java

The setup program administers EIM. As shown in Figure E-1, it sets up the necessary domains, registries, identifiers, and associations for the EimJavaDemo.

```
package eimjavademo;

import java.util.Iterator;
import java.util.Set;

import com.ibm.eim.Association;
import com.ibm.eim.ConnectInfo;
import com.ibm.eim.Domain;
import com.ibm.eim.DomainManager;
import com.ibm.eim.Eid;
import com.ibm.eim.EimException;
import com.ibm.eim.Registry;

public class EimJavaSetup {

   /**
    * @param args
    */
   public static void main(String[] args) throws EimException {

      // Cleanup first
      delete();

      // Specify the connection information
      ConnectInfo connectInfo = new ConnectInfo("cn=ldap_administrator", "secret");

      // Get an instance of the domain manager
      DomainManager domainMgr = DomainManager.getInstance();

      // Retrieve a domain
      String ldapUrl = "ldap://9.12.4.18/ibm-eimdomainname=eimJavaDemo,o=itso,c=us";
      Domain myJavaDomain = domainMgr.createDomain(ldapUrl, connectInfo, "");

      // Add a system registry
      Registry racfReg = myJavaDomain.addSystemRegistry("demoSysReg_RACF",
Registry.EIM_REGTYPE_RACF, "description", "uri");

      // Add a system registry
      Registry javaReg = myJavaDomain.addSystemRegistry("demoSysReg_JAVADEMO",
Registry.EIM_REGTYPE_LDAP, "description", "uri");

      // Add a user identifier
      Eid identifier = myJavaDomain.addEid("demoIdentifier", "description");

      // Associate the identifier with the users in various registries
      identifier.addAssociation(Association.EIM_SOURCE, javaReg.getName(), "javauser@us.ibm.com");
      identifier.addAssociation(Association.EIM_TARGET, racfReg.getName(), "javausr");

   }

   /**
    * Ensure that the domain is deleted before creating
    * @throws EimException
    */
   private static void delete(){
```

Figure E-1 EimJavaSetup - part 1 of 2

```
        // Specify the connection information
        ConnectInfo connectInfo = new ConnectInfo("cn=ldap_administrator", "secret");

        // Get an instance of the domain manager
        DomainManager domainMgr = DomainManager.getInstance();

        try{

            // Retrieve a domain
            String ldapUrl = "ldap://9.12.4.18/ibm-eimdomainname=eimJavaDemo,o=itso,c=us";
            Domain myJavaDomain = domainMgr.getDomain(ldapUrl, connectInfo);

            // Delete all registries
             Set regs = myJavaDomain.getRegistries();
             Iterator regIter = regs.iterator();
             while(regIter.hasNext())
             ((Registry) (regIter.next())).delete();

             // Delete all identifiers
             Set eids = myJavaDomain.getEids();
             Iterator eidIter = eids.iterator();
             while(eidIter.hasNext())
                ((Eid) (eidIter.next())).delete();

            myJavaDomain.delete();

        } catch(EimException e){} // Ignore exception that domain already exists

    }

}
```

Figure E-2 EimJavaSetup - part 2 of 2

Authentication program: EimJavaAuth.java

The client is responsible for authenticating the external user before giving access to z/OS resources. The stub shown in Figure E-2 is in place of an authentication mechanism that the client application would implement for the external users.

```
package eimjavademo;

import java.util.Hashtable;

public class EimJavaAuth {

   private static Hashtable lookupTable = new Hashtable();

   // Populate the lookup table with userid/password pairs
   static {
       lookupTable.put("javauser@us.ibm.com", "secret");
       lookupTable.put("javausr2@us.ibm.com", "secret2");
   }

   /**
    * Authenticates the off-platfrom userid and password. This API is just a
    * stub in place of an authentication mechanism.
    *
    * @param userid The off-platform userid.
    * @param password The off-platform password.
    * @return true, if verification succeeds and false, if verification fails.
    */
   public static boolean login(String userid, String password){
       boolean result = false;

       // Lookup the userid to retrieve the password
       String pw = (String) lookupTable.get(userid);

       // If the passwords match, then login is successful
       if(pw != null && pw.equals(password))
           result = true;

       return result;

   }

}
```

Figure E-3 EimJavaAuth

Main program: EimJavaDemo.java

This is the main code for the demo. It authenticates the external user and performs the mapping lookup of that user to a z/OS user. It then generates a PassTicket for the mapped user which can be passed to the JSec application to create a new RACF user. Refer to Figure E-4 on page 257 and Figure E-5 on page 258.

```
package eimjavademo;

import java.util.Set;

import javax.naming.directory.BasicAttributes;

import com.ibm.eim.ConnectInfo;
import com.ibm.eim.Domain;
import com.ibm.eim.DomainManager;
import com.ibm.eim.RegistryUser;
import com.ibm.eserver.zos.racf.IRRPassTicket;
import com.ibm.eserver.zos.racf.IRRPassTicketGenerationException;
import com.ibm.eserver.zos.racf.userregistry.RACF_SecAdmin;
import com.ibm.eserver.zos.racf.userregistry.RACF_remote;
import com.ibm.security.userregistry.SecAdminException;
import com.ibm.security.userregistry.User;

public class EimJavaDemo {

   /**
    * This method authenticates the specified non-z/OS userid and password. Then
    * it maps that user to a z/OS user. It creates a passticket for the z/OS user
    * which is used to bind to JSEC and create the new RACF userid specified.
    *
    * @param args args[0]=off-platform userid, args[1]=off-platform password,
    *     args[2]= RACF user to be created
    */
   public static void main(String[] args) throws Exception {

      // Verify the three arguments are passed in
      if(args.length != 3){
         System.out.println("Usage: java EimJavaDemo [userid] [password] [newRACFuserid]");
         System.exit(0);
      }

      // Authenticate the off-platform user
      String userid = args[0];
      String password = args[1];
      String newRACFuserid = args[2];
```

Figure E-4 EimJavaDemo - part 1 of 2

```
        if(!EimJavaAuth.login(userid, password)){
            System.out.println("LOGIN Error: Please re-enter the userid and password");
            System.exit(0);
        }

        // Connect to the EIM domain
        ConnectInfo connectInfo = new ConnectInfo("cn=ldap_administrator", "secret");
        DomainManager domainMgr = DomainManager.getInstance();
        Domain myJavaDomain =
domainMgr.getDomain("ldap://9.12.4.18/ibm-eimdomainname=eimJavaDemo,o=itso,c=us", connectInfo);

        // Use EIM to map the off-platform user id to a RACF user id
        Set regUsers = myJavaDomain.findTargetFromSource(userid, "demoSysReg_JAVADEMO",
"demoSysReg_RACF");
        RegistryUser regUser = ((RegistryUser) regUsers.iterator().next());
        String racfUser = regUser.getTargetUserName();

        // Generate a passticket for the RACF user
        String racfPassticket = null;
        String appl = "ITDS1";  // Must match PTKTDATA IRRPTAUTH.ITDS1.JUSTJAVA profile
        try {
            IRRPassTicket ticket = new IRRPassTicket();
            racfPassticket = ticket.generate(racfUser, appl);
        } catch (IRRPassTicketGenerationException e) {
            System.out.println("Passticket ERROR: " + e);
            e.printStackTrace();
        }

        // Create a new user with JSEC
        RACF_remote racfConnection = new RACF_remote("ldap://9.12.4.18", "simple", racfUser,
racfPassticket, "o=itsoracf");
        RACF_SecAdmin admin = new RACF_SecAdmin(racfConnection);
        try{ // Delete the user if it already exists
            admin.deleteUser(newRACFuserid);
        } catch (SecAdminException e){} // Ignore exception that user doesn't already exist
        try{
            BasicAttributes userAttr = new BasicAttributes();
            admin.createUser(newRACFuserid, userAttr);
            User user = admin.getUser(newRACFuserid);

            // List the user
            System.out.println(user.getAttributes().toString());

        } catch (SecAdminException e){
            System.out.println("JSEC ERROR: User already exists or could not create user");
            e.printStackTrace();
        }

    }
}
```

Figure E-5 EimJavaDemo - part 2 of 2

F

Basics of cryptography

This appendix introduces some basic cryptographic concepts. The following topics are covered:

- Cryptographic algorithms
  - Symmetric key algorithms
  - Asymmetric key algorithms
- Additional functions
  - Padding
  - Encryption modes
- Hybrid encryption
- Digital signature
- Certificates

© Copyright IBM Corp. 2008. All rights reserved.

Introduction to cryptography

The word cryptography originates from the Greek words “kryptos”, which means hidden and “gráfo,” which means to write or to speak. Cryptography deals with securing information by transforming it using cryptographic algorithms. This transformation is performed so that the real form cannot be revealed without special information. This special information is referred to as “the key”.

The process of transforming data from clear text into a hidden form is called *encryption*. And the reverse transformation, from a hidden form into clear text, is called *decryption*.

Cryptographic algorithms

Currently there are two categories of cryptographic algorithms intended for encrypting and decrypting data: symmetric key algorithms and asymmetric key algorithms.

Symmetric key algorithms

In symmetric key algorithms, the same key is used for both encryption and decryption. Because a symmetric key can be used to decrypt everything that has been encrypted with it, it needs to be kept secret. Because of this, it is often referred to as a “secret key”.

If several parties want to share secret information using a symmetric key, they all need to know the key. This introduces the problem of distributing the key to whomever is authorized to use it, without exposing it to those which are not authorized to get it.

Figure F-1 shows the flow of encryption and decryption using symmetric key algorithms. As shown, the same key is used for both encryption and decryption.

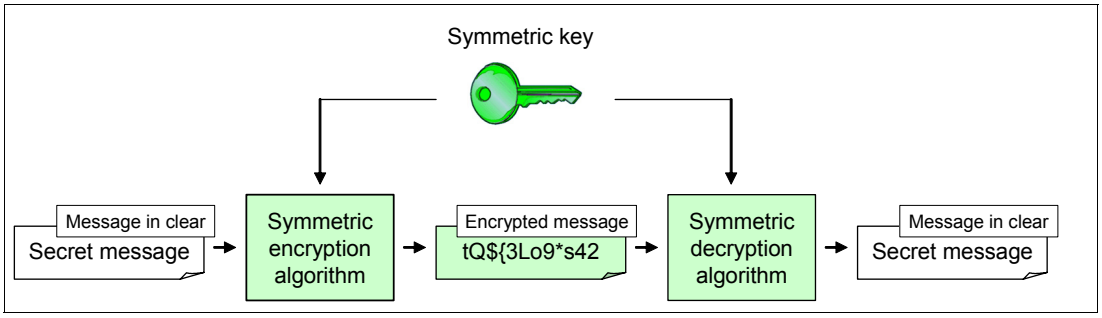


Figure F-1 Symmetric key algorithm

Symmetric key algorithms perform comparatively faster than asymmetric algorithms, but the key management stage can increase rapidly in complexity when several parties are involved, because the secret key needs to be distributed securely to everybody involved.

Today, widely used symmetric algorithms include: DES, Triple-DES, Blowfish, and AES.

Note: Even though DES is still in use, it is now considered to have too short a key for ensuring proper encryption strength. Currently it is recommended that you migrate to algorithms with longer keys, such as Triple-DES or AES.

Asymmetric key algorithms

In asymmetric key algorithms, there are two related keys (the “key pair”) involved. When data is encrypted with one of the keys, only the other key of the pair can be used for decryption. The owner of the key pair selects one of the keys to become a private key, which needs to be kept secret. The other key then becomes a public key, and can be freely distributed. The key construction algorithm is such that it is extremely difficult (that is, impractical) to derive the value of the private key from the public key value.

Because the public key is freely distributed, anybody can encrypt messages with it, but only the holder of the private key can decrypt those messages. The benefit of this is that no secret key needs to be distributed before secure communication can take place (as is the case with symmetric algorithms). If two parties want to communicate securely using an asymmetric algorithm, they both use the recipient’s public key for encrypting messages, but use their own private key for decrypting what they receive.

Figure F-2 shows the flow of encrypting and decryption using asymmetric key algorithms. Encryption is done using the recipient’s public key, and decryption can then only occur using the recipient’s private key.

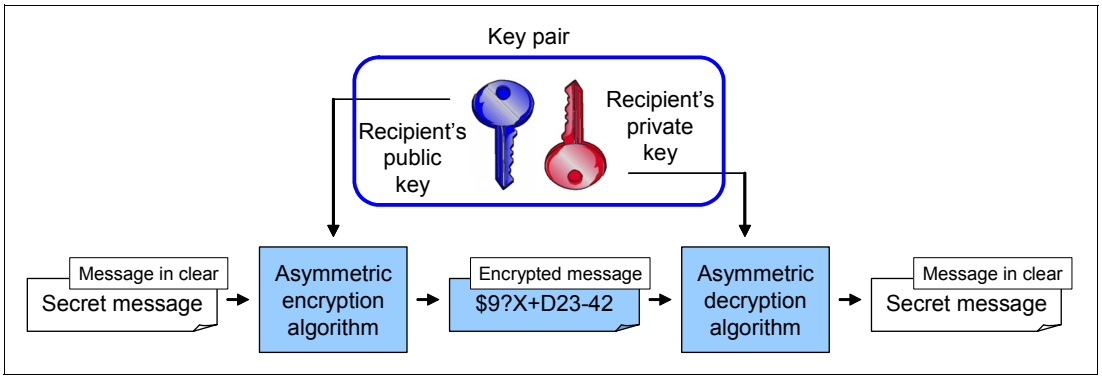


Figure F-2 Asymmetric key algorithm

Compared to symmetric key algorithms, asymmetric algorithms are extremely heavy consumers of computing resources. They are usually reserved for encryption of short bursts of data, and are used in combination with symmetric key algorithms. One of the most widely used asymmetric algorithms today is RSA (Rivest-Shamir-Adleman).

Padding

Many encryption algorithms work by encrypting one block of clear text at a time. The size of these blocks depends on the algorithm used and the size of the keys. If the size of the data that needs to be encrypted does not reach a whole number of blocks, the data needs to be “padded.” This padding takes place before the data is encrypted, and it is removed at decryption time. Several padding schemes are in use today.

Encryption modes

When encrypting using standard encryption algorithms, two identical data elements that are encrypted with the same key and algorithm would end up looking the same when encrypted. Because normal encryption algorithms work in blocks, repeating patterns in the clear data can end up as repeating patterns in the encrypted data, as well. This would make it easier for someone to break the encryption.

One way to avoid this is by using the output of each block encryption to modify the input of the next sequential block's encryption (for example, by an exclusive OR operation). In this way, all encrypted data blocks contribute to changing the pattern of the next encrypted block. To start up this process, a randomly generated initial value is used to change the encrypted output of the first block of data. This value is called an *initialization vector*, and it must also be known by the recipient of the encrypted data.

Hybrid encryption

Both symmetric and asymmetric cryptographic algorithms offer advantages and disadvantages. Symmetric algorithms are fast, but securely distributing the symmetric keys to many users may prove to be a very complex and cumbersome process. Conversely, asymmetric algorithms are slow and extremely demanding of computing resources, but they solve the key distribution problem because a public key does not require to be secured.

Hybrid encryption attempts to exploit the advantages of both kinds of algorithm classes, while avoiding their disadvantages.

Figure F-3 on page 263 shows how hybrid encryption works. When the sender of a message wants to send it encrypted to a recipient, the sender starts up the process by generating a random symmetric key that is used encrypt the secret message. The symmetric key is then encrypted using the recipient's asymmetric public key. The final message that is sent to the recipient comprises two parts: the encrypted symmetric data key that will be recovered by the recipient using the recipient's asymmetric private key, and the recovered symmetric key that will be used to decrypt the message.

Note that this approach allows the exploitation of the symmetric encryption/decryption inherent efficiency for the transmitted data part. It also exploits the much simpler key management processes that asymmetric encryption/decryption offers. The cost of asymmetric encryption is kept to a minimum because it is expected that the symmetric key will be considerably shorter than the secret message itself.

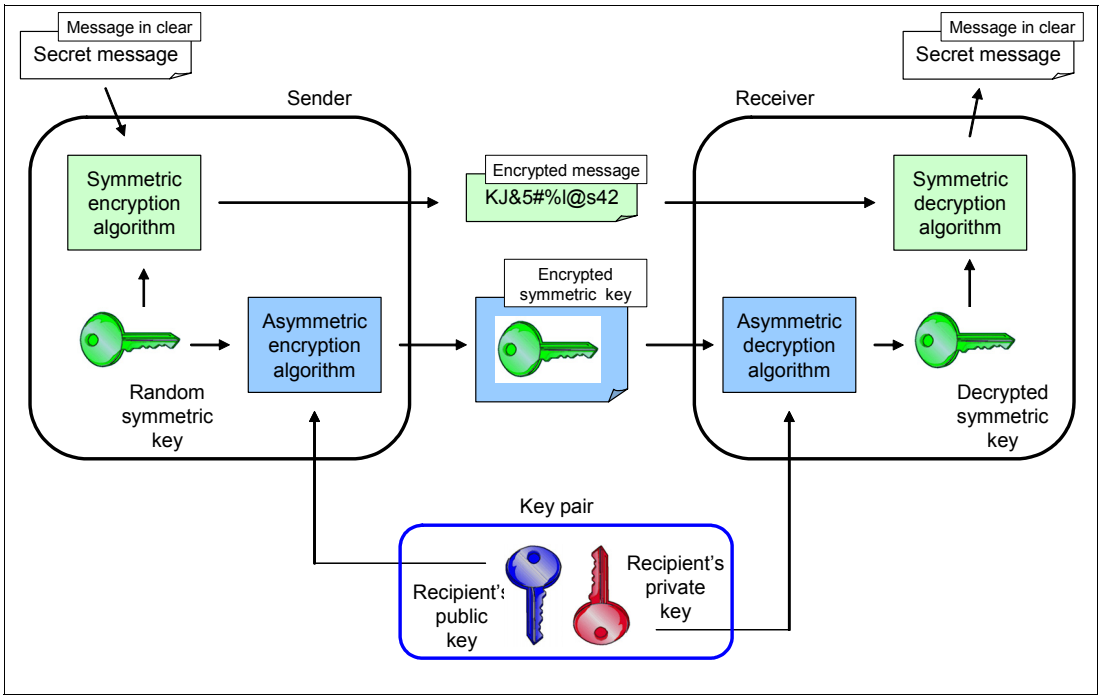


Figure F-3 Hybrid encryption

With hybrid encryption, the use of the random symmetric key is not necessarily limited to only encrypting one message. Instead, the key can be reused over the course of a communication session. In that case, the random symmetric key is sometimes referred as a *session key*.

Hybrid encryption techniques are heavily used; a prominent example is the SSL/TLS protocol.

Digital signatures

Digital signature technology aims to provide the same guarantees that you expect from an handwritten signature, but at the scale, distance, and speed that electronics permit. A handwritten signature on a document makes it evidential, and a digital signature provides this same property to a transmitted message.

The algorithms used for making up digital signatures are usually a combination of asymmetric cryptographic algorithms and one-way hash algorithms.

One-way hash algorithms

One-way hash algorithms produce *hash values*, that is, fixed-length binary values computed from an input message. These hash values are usually in the order of a few tens or hundreds of bits in length. However, the input message can be of any length, and it cannot be retrieved from the hash value (thus, “one-way hash”).

A hash value is used as a checksum, or “fingerprint”, for the message that produced it, because changing a single bit in the message would change the hash value. The recipient of a message can then verify the integrity of a received message by matching the initial hash value sent with the message and the hash value computed on the final received message.

However, the number of possible combinations produced by the fixed length of the hash value is much smaller than the combinations that the message length itself is expected to

produce. It may happen that two different messages produce the same hash value. This is called a “collision” and is unavoidable when using one-way hashes.

Cryptographic techniques are therefore used when designing one-way hash algorithms to make it extremely difficult for an attacker to find the necessary modifications to a given message so that it could again, though modified, produce the same hash value.

Several families of algorithms aim to produce one-way hashes: Message Authentication Code (MAC), Message Detection Code (MDC), Message Digest (MD), Secure Hash Algorithm (SHA), and so on.

Digital signature generation

Figure F-4 shows the generation of a digital signature for a given message. First, the hash value of the message is calculated using a chosen one-way hash algorithm. Then the hash value is encrypted using the signer’s asymmetric private key. This last step implies the signer is using the unique private key that the signer owns.

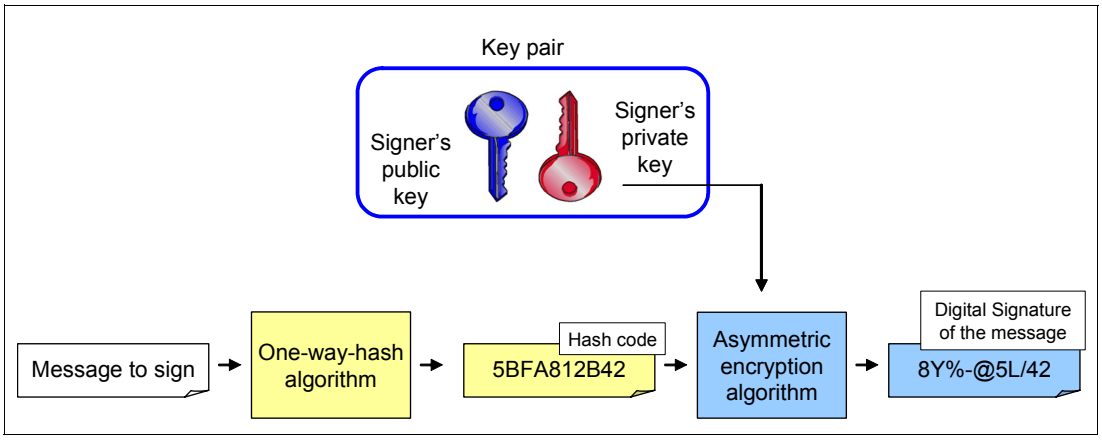


Figure F-4 Digital signature generation

Digital signature verification

Figure F-5 on page 265 illustrates how a digital signature is verified. First the signature is “decrypted” using the signer’s public key, thus yielding the hash value that was computed just prior to signing the message. A hash value is then calculated against the received message and the two hash values are compared. If they match, the process has then established that:

1. The message went unmodified to the recipient.
2. The message was actually sent by the declared signer because the digital signature (that is the initial hash value encrypted with the signer’s private key) could be decrypted with the signer’s public key.

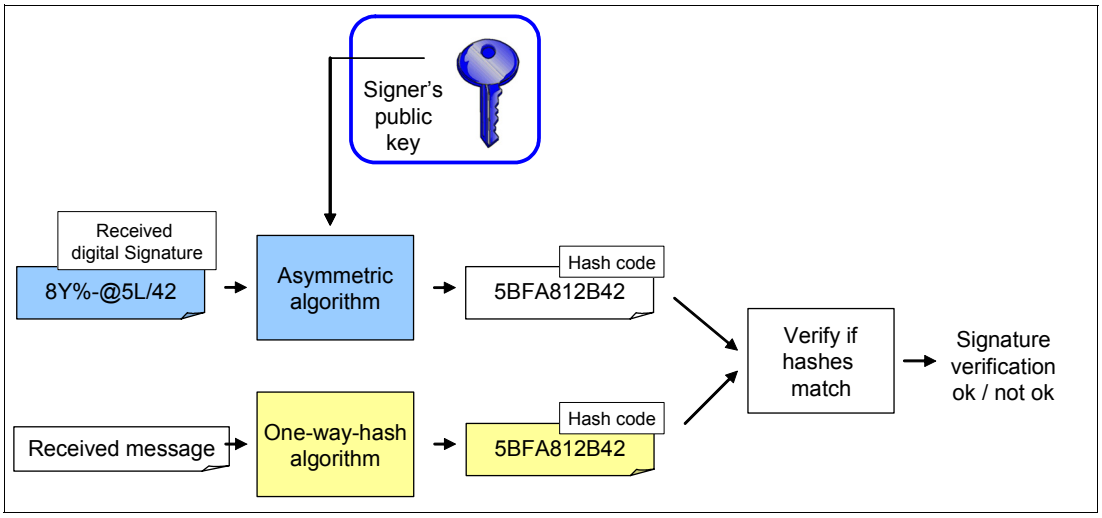


Figure F-5 Digital signature verification

Digital signatures also play a key role in the generation of digital certificates as a proof of the integrity and origin of the digital certificate.

Digital signatures are therefore a combination of one-way hash and asymmetric algorithms. Widely used combinations include MD2 with RSA, MD5 with RSA, SHA1 with RSA, SHA1 with DSA.

Digital certificates

In large uncontrolled networks, where everybody can publish their own public keys, there is a requirement to certify that the declared identity is actually the key owner's identity. The method widely in use today to achieve this certification is to package one's public key in a file called a "digital certificate." The digital certificate file contains, among other information, the public key value and the public key owner's name, and is digitally signed by a Certificate Authority (CA).

Certificate authorities have an administrative process in place to verify the identity of requestor and whether this identity, as it will be shown in the certificate, is unique. After this administrative process is successfully performed, a digital certificate is issued to the requestor. The content of the certificate is digitally signed by the CA with its private key.

Because the CA makes its public key public in a CA certificate, all recipients of the certificate can use the CA's public key to verify the certificate digital signature, thus binding the public key value in the certificate to the owner's (the "subject's") identity, which is also shown in the certificate.

Note that a certificate verification is to be performed by software, and the program that verifies certificates is set up to accept certificates of only selected CAs. These are the CAs that are trusted as per the installation trust policy.

The prominent format in use today for digital certificates is X.509 V3 format, which is promoted by the IETF PKIX group. 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.

An X.509 V3 certificate contains the following information:

- Certificate fields
  - 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 mentioned, the digital certificate signature is used at verification time to check the integrity of a received certificate and its origin; that is, the CA that signed the certificate.

G

Case study: IBM Encryption Key Manager

This appendix describes the exploitation of Java Security by the IBM Encryption Key Manager (EKM) program product as a typical example of an industry-class use of Java security and java cryptography. Currently, EKM is bundled as a part of the IBM JDK to provide support for the IBM Tape Encryption Solution on TS1120 and TS1040 encryption capable drives. A full description of that solution can be found in the Redbooks document *IBM System Storage TS1120 Tape Encryption: Planning, Implementation, and Usage Guide,* SG24-7320.

Further specific details about EKM can be found in *IBM Encryption Key Manager Component for the Java Platform Introduction, Planning, and User's Guide*, GA76-0418.

EKM can execute on any Java-capable platform. This appendix explains how EKM exploits different keystores and Java cryptographic providers when it is executing on z/OS. It also addresses the multi-EKM instances configuration where EKMs are synchronized with communications protected by the JSSE provider.

Note: The information presented in this appendix may not fully apply to EKM instances executing on a non-z/OS platform.

© Copyright IBM Corp. 2008. All rights reserved.

EKM overview

IBM Encryption Key Manager (EKM) is a stand-alone server that is bundled with the IBM JDK. It is written in pure Java and is used as a key server for the IBM TS1120 and TS1040 Tape Units. EKM can therefore execute on any Java-capable platform.

These hardware units use the keys that EKM sends to encrypt or decrypt data at the same time as they are written on or read from the tapes. EKM is a lightweight application that reads a keystore into memory and then uses the keys in that keystore to satisfy the drive requests. The tape units communicate with EKM over a TCP/IP connection. In a multi-EKM configuration, EKM instances communicate together, also using TCP/IP connections. EKM instances share mainly configuration information so that tape units get an EKM instance assigned to them, as well as a backup EKM that is able to take over for a failing EKM instance.

When starting, an EKM instance binds to two network sockets, reads its keystore into memory, and waits for either a tape unit or another EKM instance to initiate a handshake with it. This is a normal SSL/TLS handshake occurring between EKMs. In contrast, the connection between a tape unit and EKM, although implementing an SSL/TLS-like connection, uses a proprietary protocol.

Java cryptographic providers used by EKM on z/OS

Depending on how EKM is configured, it may use a combination of the following cryptographic providers:

- IBMJCE
- IBMJCECCA
- IBMJSSE
- IBMJCEFIPS

IBMJCE

When EKM is configured to deal with software keys and keystores, it uses the IBMJCE provider. The keystores that EKM uses on z/OS are therefore JCEKS and JCERACFKS.

IBMJCECCA

The IBMJCECCA provider allows applications that are configured to use it to obtain hardware assistance from z/OS integrated hardware cryptography. If EKM is set up to use a JCECCAKS or JCECCARACFKS keystore, it then implies that it uses the IBMJCECCA provider to exploit the cryptographic hardware coprocessors.

The TS1120 tape unit is sent by EKM as a randomly generated AES 256 key to be used for encryption of the data on the tape. When on z/OS, EKM can be configured to generate that key as a clear key or, if the IBMJCECCA provider is specified, to generate and use the key as an ICSF secure key. In that case, a Triple-DES secure key is randomly generated by ICSF, using the cryptographic hardware coprocessor because, at the time of writing, there is no support for AES secure keys in ICSF.

This Triple-DES key never appears in clear in the z/OS platform, and it is sent to the tape unit encrypted by the tape unit‘s RSA public key. The Triple-DES key is then recovered in the tape unit hardware cryptography facility and is properly padded to meet the 256-bit length of an AES key.

IBMJSSE

The IBMJSSE provider is used for SSL/TLS-protected communications, such as the communications established between different instances of EKM. As mentioned, the JSSE provider itself does not perform any encryption. It calls either the IBMJCE or IBMJCECCA provider, depending on the specification in the java.security file. This is an important consideration to keep in mind when FIPS compliance settings for the EKM are discussed.

IBMJCEFIPS

The IBMJCEFIPS provider is a subset of the IBMJCE provider that has a stabilized codebase that was approved for FIPS certification. Some customers have a requirement to use only FIPS-certified cryptographic applications.

Because EKM itself does not perform any cryptographic operations, it does not need to be FIPS-certified. However, it should then use the IBMJCEFIPS provider. This provides FIPS-certified algorithm implementations for securing EKM-to-EKM SSL/TLS communications, EKM - tape units SSL/TLS-like communications, and the generation and wrapping of the data key used by the tape unit.

Note, however, that the IBMJCCECCA provider is not FIPS-certified and cannot therefore be used with EKM on z/OS when the use of a FIPS-certified provider is required.

EKM network traffic security

This section briefly describes the differences and similarities behind EKM SSL/TLS-protected communications and the EKM used to drive network security.

EKM-to-EKM SSL/TLS

SSL/TLS communication is initiated by an EKM instance (the SSL/TLS client) that needs to synchronize another instance (the SSL/TLS server) with a set of information to be shared. EKM SSL/TLS communication can be set up so that SSL/TLS client authentication is performed during the handshake. By default, there is no client authentication during the handshake.

EKM can therefore act as an SSL/TLS client or server. When acting as a client, EKM can provide a digital certificate for client authentication. As a consequence, an EKM instance would need four keystores to support SSL/TLS communication. It would need a keystore and truststore (that is, a keystore hosting trusted certificates only) when acting as a client, and another keystore and truststore when acting as a server.

If we include the keystore that is holding the keys exploited for tape unit encryption, this brings the total number of configurable keystores for an EKM instance up to five. In practice, however, EKM users often use a single keystore to host all keys and certificates in order to simplify the keystore.

EKM-to-tape unit security

This section describes the communication between EKM and a TS1040 tape unit. For information about EKM communication with a TS1120 tape unit (which involves several additional steps), refer to the IBM Redbooks publication *IBM System Storage TS1120 Tape Encryption: Planning, Implementation, and Usage Guide*, SG24-7320.

Each IBM tape unit contains a digital certificate signed by IBM and installed in the tape unit at the factory. This certificate is used by the tape units when they identify themselves to the EKM. All EKM instances also have a copy of the IBM root certificate so that they can verify any certificate sent to them by the tape units.

When initializing, a tape unit initiates a connection to EKM and sends to EKM a public key signed by the tape unit certificate's private key. EKM will verify that the tape unit's certificate is signed by the IBM root certificate.

If that verifies, EKM generates an AES 256 data key (unless EKM executes on z/OS with secure key specified, as explained in "IBMJCECCA" on page 268). The data key is sent to the tape unit encrypted with the public key that the same tape unit sent previously.

The drive has the corresponding private key, and is able to decrypt the public key encrypted data key. It can then use that data key to encrypt data sent to the tape.

EKM and z/OS keystores

EKM supports several different types of keystores across the IBM servers platforms. When executing on z/OS, EKM supports the following keystores:

- JCEKS
- JCECCAKS
- JCERACFKS
- JCECCARACFKS

JCEKS

Entries of the JCEKS keystore are encrypted with Triple-DES. This is a UNIX file-based keystore, and its access is protected by UNIX permission bits or ACL. This keystore is a purely software implementation and does not support secure keys. This keystore allows keys to be exported and transmitted in files in the PKCS#12 format.

JCEKS hosts both symmetric and asymmetric keys.

JCERACFKS

JCERACFKS is the "externalization" of the z/OS external security manager, RACF for IBM, as a Java keystore. It allows Java users to store their keys and certificates in RACF key rings or an equivalent. Access to JCERACFKS contents is more strongly protected than JCEKS.

This keystore is a purely software implementation and does not support secure keys. It hosts asymmetric keys only (DSA or RSA). It also allows keys to be exported in a PKCS#12 format file if required.

JCECCARACFKS

JCECCARACFKS is similar to JCECRACFKS in that it uses the external security manager repository to store certificates. However, the keys themselves are stored in the ICSF PKDS and secure keys are supported.This keystore only hosts RSA asymmetric keys.

JCECCAKS

JCECCAKS is similar to the JCEKS keystore in that it supports both symmetric keys and asymmetric keys. The difference is that this keystore also supports secure keys that are stored in the ICSF CKDS or PKDS.

Currently, the *EKM Installation Planning and Usage Guide* states that this keystore supports secure asymmetric keys and clear symmetric keys only.

Synchronization of multiple keystores

JCEKS

Key management between JCEKS keystores falls into a standard practice of exporting keys in PKCS#12 files, and importing these PKCS12 files into another JCEKS keystore. Typically in an EKM installation, customers will simply copy one EKM instance keystore to all other EKM instances. This makes the synchronization of keys between EKMs very easy.

JCERACFKS

Keeping keys synchronized between different RACF databases through JCERACFKS is again a very standard practice. Users simply export RSA or DSA keys in a PKCS#12 file from one RACF database and then import the file into another RACF database.

JCECCARACFKS

The JCECCARACFKS keystore does not allow the exportation of RSA private keys in a PKCS#12 file because those private keys are actually kept encrypted with the coprocessor asymmetric Master Key in the PKDS.

Sharing private keys between different JCECCARACFKS requires exchanging the encrypted private keys between z/OS instances using the same coprocessor asymmetric Master Key. The KEYXFER utility, which can be downloaded from the IBM tools and toys Web site (at `ftp://ftp.software.ibm.com/s390/zos/tools/keyxfer/keyxfer.rexx.txt`), is a REXX-based program that allows the user to pull a private key by label out of the PKDS. The key is in the Master Key encrypted format and put into a dataset.

From there, the private key can be moved to another system and put into that other system's PKDS. Again, this target system would need to be set up with the same asymmetric Master Keys as the source system. After the private key has been entered into the destination PKDS, then the public key and certificate associated with the private key can be exported out of the RACF database and imported using the regular RACF RACDCERT command options.

It is useful to first generate the RSA asymmetric keys in software, export them in a PKCS#12 format, and then import them into the target JCECCARACFKS instances. However, if the keys are generated first by software and later migrated into a format where they are kept encrypted with the coprocessor asymmetric Master Key, they cannot be given a true status of secure key because they were first created as clear keys.

JCECCAKS

The capability of sharing keys in a JCEKS by exporting them to other JCEKSs depends on whether the keys are kept as secure keys (no export possible) or clear keys (export possible).

A possible extension to EKM capabilities

As mentioned, EKM does not currently support using symmetric secure keys with the JCECCAKS keystore. The symmetric keys are always clear keys, meaning that they are not actually stored in the ICSF CKDS but remain in a UNIX file.

To apply what we learned in this book, here we propose a possible approach. Assuming it might meet some EKM users' requirements, have EKM eventually use ICSF secure symmetric keys.

The process would be as follows:

- Create an ICSF secure Triple-DES key in the ICSF CKDS.
- Use Java APIs to achieve mapping the key's CKDS key label to a JCECCAKS key store.
- Set up the EKM with the property zOSCompatability = true.
- Start the EKM, list the keys it has access to, and then try to encrypt a tape using the secure key.

Creating a symmetric secure key in the ICSF CKDS

In this section, we show how to use an interactive invocation of the Key Generator Utility (KGUP program) via the ICSF ISPF panels. (For more information about KGUP invocation and execution, refer to *z/OS Cryptographic Services Integrated Cryptographic Service Facility Administrator's Guide*, SA22-7521.)

To create the secure key, follow these steps:

1. Invoke KGUP (option 8) from the ICSF main panel.
2. From the KGUP menu, select **1** for **create key generator control statements**.
3. At the Control Dataset Spec panel, enter *<userid>*.CSFIN for the data set name, and press Enter. Substitute your TSO userid for *<userid>*. This will be the CSFIN input dataset that will hold your control statements for creating the keys.
4. At the allocation panel, press Enter to take the defaults. Your data set will be allocated and you will see the KGUP Control Statement Menu.
5. At the KGUP Control Statement Menu, select **4** to edit the statement storage data set.
6. You will now be editing the empty *<userid>*.CSFIN data set. This is where the following KGUP statement is entered, requesting the generation of a random secure Triple-DES (24-byte) data key.

   ```
   ADD TYPE(DATA) LENGTH(24),LAB(tekmkey)
   ```

7. Press PF3 as many times as required to get back to the ICSF Key Administration menu. Select **2** for **specify data sets for processing**.
8. At the Specify KGUP Datasets panel, fill in these data set names, in single quotes:
   - For Cryptographic Keys, enter the name of your CKDS. If you do not know the name of your CKDS, you can find it by going to the ICSF main menu and selecting the **OPSTAT** option. From there, select the **OPTIONS** option to **Display Installation Options**. Your CKDS will be listed as the Active CKDS.
   - For Control Statement Input, enter *<userid>*.CSFIN.
   - For Diagnostics, enter *. This will expose any key values specified in output in the SDSF queues.
   - For Key Output, enter *<userid>*.CSFKEYS. When the CLEAR parameter is specified on the ADD key statement, the undeciphered key value displays in the CSFDIAG DD output file.
   - For Control Statement Output, enter *<userid>*.CSFSTMNT.
9. Press PF3 to save.
10. At the ICSF Key Administration menu, select **3** to submit your KGUP job.
11. At the Set KGUP JCL Job Card panel, set Special Security Mode (at the bottom of the screen) to YES, and change the job card information to be valid in your environment.
12. On the Set KGUP JCL Job Card panel, at the command prompt, enter S to submit.

13. Check the KGUP job SYSOUT to make sure you get a zero return code. If a nonzero return code is received, an explanation of why such a return code was received is displayed after the offending statement in the CSFDIAG DD file.
14. At the ICSF Key Administration menu (the KGUP main menu), select **4** to refresh the cryptographic key data set. At the Refresh in-storage CKDS panel, for New CKDS, enter your CKDS name and press Enter. `REFRESH SUCCESSFUL` should display at the upper right corner of the screen.

    Refresh only updates the in-storage CKDS on the LPAR where the request is made. If the key value is required on other LPARs sharing this CKDS, a refresh must also be performed at the ICSF instances executing in those LPARs.

Referring to the secure symmetric key in the CKDS

At this point you should have a secure key with a label “tekmkey” in the ICSF CKDS. This key must now be mapped to a JCECCAKS keystore so that Java applications can refer to it. This is done programmatically.

Writing the complete Java program to achieve this mapping is the reader’s responsibility. However, here we provide excerpts showing which Java functions your program should exploit.

First, the objects have to be loaded into memory with the following statements:

```
SecretKeyFactory myKeyFactory =
SecretKeyFactory.getInstance(algorithm,"IBMJCECCA");

            KeyLabelKeySpec spec = new KeyLabelKeySpec(tekmkey);

            SecretKey key = myKeyFactory.generateSecret(spec);
```

Next, the keystore itself should be loaded into memory with the following statements:

```
KeyStore ks = KeyStore.getInstance(keystore_type);

ks.load(fis, keystore_password.toCharArray());
```

Finally, map your keystore label entry to this keystore:

```
            ks.setKeyEntry(keystore_alias, key, keystore_password.toCharArray(),
null);
```

Using the secure symmetric key with EKM

Now that you have a keystore with entries mapped to the CKDS, EKM has to be set up to point to the keystore. The following property must also be set up so that EKM loads the secure Triple-DES key and sends it to the tape unit, where, after padding, it will be used as an AES 256 data key.

```
zOSCompatibility=true
```

H

Performance case study: IBM Encryption Facility for z/OS OpenPGP support

IBM Encryption Facility for z/OS is a program product (Program Product 5655-P97) that implements the OpenPGP encryption protocol for the encryption and decryption of data in datasets or z/OS UNIX files, with the optional compression of data prior to encryption. Because OpenPGP (RFC 2440) is widely supported on many systems, the Encryption Facility for z/OS OpenPGP support can be used to securely exchange data between z/OS and numerous non-z/OS, including non-IBM, systems.

The OpenPGP support has been implemented as a set of Java programs in the Encryption Facility for z/OS program product. Because the support could be invoked to handle a large amount of data in a limited time, it is of prime importance to understand how its performance could be optimized and what product features can be exploited to achieve this objective.

This appendix discusses the capabilities offered by the OpenPGP support for performance optimization as typical examples of possible approaches to improve the performance of any Java program executing on z/OS.

© Copyright IBM Corp. 2008. All rights reserved.

Introduction

The following two performance metrics are used to discuss the performance-related implications of deploying the IBM Encryption Facility for z/OS OpenPGP support:

- The execution time, measured by a standard watch or clock and the amount of work accomplished within that time
- The cost, measured by the CPU consumption on general purpose processors

Applications written in Java on z/OS require the IBM Java Runtime Environment for z/OS (JRE) for their execution environment. This environment consists of an address space where the JRE executes. The JRE interprets compiled Java byte code that serves here 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 immediately impacted. Moreover, the cryptographic nature of the Encryption Facility products requires the completion of computationally-intensive operations that could further increase the amount of CPU time consumed and expand the execution time.

Finally, OpenPGP's main purpose is to provide data integrity services on datasets and z/OS UNIX files. This involves I/O-intensive operations that may result in inefficient utilization of the CPU.

To address these issues, performance enhancing options in the following three areas should be considered:

1. IBM Java 6 SDK Runtime Environment for z/OS
2. z/OS specialized hardware devices: zAAP and CPACF specialized processors
3. OpenPGP support code performance options

IBM Java SDK 6 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, while also 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 optimize the compilation to exploit the latest hardware technology available on the processor.

It is strongly recommended that the JIT compiler always remain enabled. Thus, this case study will not discuss the performance implications of disabling the JIT compiler. If disabled, the performance of any Java application may be significantly degraded.

System z specialized Hardware: zAAP and CPACF

The CP Assist for Cryptographic Function (CPACF) is the System z cryptographic device that can be used to accelerate symmetric encryption and decryption and hash calculations. The CPACF performs cryptographic functions at an extremely high speed, helping to reduce the total elapsed time and CPU time when such functions are involved.

Unlike previous processors where there is a CPACF assigned to each PU, the System z10 processor (where these measurements were made) has a CPACF shared by a pair of PUs. Consequently, the number of CPACFs available to process a request may be fewer than the number of PUs processing the instruction flows. For workloads such as the one we ran (AES

with 128-bit keys encryption algorithm) that heavily uses the CPACF feature, the performance characteristics may be affected if the PUs happen to compete for an available CPACF.

The System z Application Assist Processor (zAAP) is a specialized PU (a System z “Specialty Engine”) that only executes Java workloads. When Java workloads are dispatched on a zAAP processor, no general CP service units are consumed by these workloads. That is, the general CPs capacity, which is usually the basis for charging for the use of licensed programs, is unaffected by these workloads.

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 allow the execution, in parallel on different PUs whenever possible, the different tasks that the OpenPGP support is required to perform: encryption, I/O accesses, and data compression.

This case study discusses the OpenPGP support performance in the context of the two given metrics. 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 compatibility while still 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.

Notes on the performance data: The performance data presented in this appendix was obtained in a controlled environment with specific performance benchmarks and tools. This information is presented along with general recommendations to assist reader understanding of IBM products. Results obtained in other environments may vary significantly. The data presented here does not predict performance in a specific installation's environment. In addition, readers should be aware of the following:

- All tests were run on zSeries System z10.
- All data was gathered using a 1.5 GB input file.
- All I/O is done to and from data sets on DASD.
- AES with 128-bit keys was the encryption algorithm used.
- Due to compression, the output file may not be the same size as the input file.
- The CPU utilization percentage given in many tables is the average utilization measured over all the enabled processors of the same type. That is, an average may be given for all the zAAP processors and a different average may be given for all the general CPs. As a result, when multiple processors are online, the calculated average may appear to be impacted. However, the aggregate of CPU time consumption is roughly equivalent. For example, there may not be a reduction of overall CPU time consumed if a CPU utilization value of 50% when only one general CP is available is reduced to 25% when two general CPs go online.
- The elapsed time is given in seconds.
- When giving statistics for zAAP processors, the internal throughput number given is the throughput for the general CPU and zAAP combined.
- When doing compression, the zip compression algorithm was used with level 1. The level value of 1 yielded the best performance. Tests have shown that a compression level of 9, which is the best possible compression ratio, results in a very significant performance reduction.

Compression performance and compression percentages are greatly impacted by the type of data being compressed.

General CP time reduction using System z specialized hardware

Two specialized hardware devices may be leveraged to positively impact performance: the zAAP specialty engine and the CPACF hardware cryptographic facility. 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 the user may specify the -jce-providers command line option with value com.ibm.crypto.hdwrCCA.provider.IBMJCECCA. This enables the hardware cryptographic JCE provider. This provider invokes z/OS ICSF to exploit the System z hardware cryptography. In our case, the CPACF was used to perform the AES 128 encryption.

If these options are not specified and the security.provider.1 keyword in $(java_home)/lib/security/java.security file has not been updated to list the com.ibm.crypto.hdwrCCA.provider.IBMJCECCA provider, then the cryptographic functions will be performed exclusively by software (that is, the Java code within the JCE component of the JRE).

Hardware cryptographic acceleration

Table 12-1 lists the performance statistics measured without the hardware provider being enabled, only one general CP online, and compression level 1 enabled.

Table 12-1 Performance with one CP and no 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	97.72	14.70	60.87%	24.15

Table 12-2 lists the statistics measured with the hardware provider being enabled and only one general CPU and one CPACF online.

Table 12-2 Performance with one CP and the hardware provider 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	86.67	16.58	56.31%	29.44

The hardware provider reduced the elapsed time by 11% and improved the MB/CPU seconds by 22%.

zAAP usage

Table 12-3 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 12-3 Performance with one CP and one zAAP, without the hardware provider

Elapsed time (sec)	MB/sec	zAAP utilization %	General CPU utilization %	MB/General + zAAP CPU sec
97.11	14.79	62.28%	1.65%	23.14

Comparing the results in Table 12-1 and Table 12-3, the general CP utilization is reduced by more than 97%. Table 12-3 dissociates the 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. The System z10 shifted almost all of the work to the zAAP processor.

Finally, the introduction of the zAAP processor resulted in a slight reduction in elapsed time of about 1%. Using a zAAP, the MB/zAAP+General CPU seconds throughputs about 4% lower than the total MB/CPU seconds without a zAAP.

These comparisons are also valid for the case when the specialized cryptographic processor is being leveraged. Table 12-4 on page 280 lists the statistics measured with the hardware provider being enabled and one general CPU online and one zAAP process online.

Table 12-4 Performance with one CP and one zAAP, with the hardware provider enabled

Elapsed time (sec)	MB/sec	zAAP utilization %	General CPU utilization %	MB/General + zAAP CPU Sec
85.11	16.88	58.51%	1.10%	28.32

Comparing the results in Table 12-2 on page 279 and Table 12-4, the general CPU utilization is reduced by over 98%. Using a zAAP, the MB/zAAP+General CPU seconds was about 4% lower than the total MB/CPU seconds without a zAAP.

Execution time reduction using parallel processing

OpenPGP support processing characteristics allow for the mitigation of a system's processing inefficiencies, specifically while the OpenPGP support task is held off awaiting the completion of an I/O operation. This holds true for both general purpose CPs and the zAAP specialty engines. An idle task can be seen as lost opportunity to reduce the total execution time.

OpenPGP support provides three configuration options that enable a multi-threaded approach to processing, as shown in Table 12-5.

Table 12-5 Processing time optimization 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 the main line processing. Further, multiple CPUs can be used to handle processing in a concurrent fashion. This results in a reduction of the total execution elapsed time.

Table 12-2 on page 279 lists some performance statistics for encryption and compression without any of the options enabled and only one general CPU and one CPACF online.

In contrast, Table 12-6 lists the measurements done when USE_ASYNC_IO, USE_ASYNC_COMPRESS, and USE_ASYNC_CIPHER are enabled and only one general CPU and one CPACF is online.

Table 12-6 Performance with all optimization options, one CP and the hardware provider 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	65.25	22.02	89.02%	24.73

With one general CP and one CPACF available, full parallel processing shows a decrease of elapsed time of about 25%. However, due to significantly higher CPU utilization, the MB/CPU second drops by 16%.

Table 12-7 lists the statistics when USE_ASYNC_IO, USE_ASYNC_COMPRESS, and USE_ASYNC_CIPHER are enabled, and four general CPUs and four CPACFs are online.

Table 12-7 Performance with all optimization options, four CPs and the hardware provider 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	55.82	25.74	37.79%	17.03

A 36% improvement is seen when comparing the elapsed times of enabling all the parallel processing features and with four general CPs online to no parallel processing and only one general CP online.

External throughput (MB/sec) is significantly improved. Internal throughput (MB/CPU sec) is lower because of as many as four CPs are used.

Putting it all together

The previous sections separately show the effectiveness of exploiting the zAAP specialty engine and hardware cryptography. Also, when multiple CPs 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 CPs and specialized hardware.

Table 12-8 lists the statistics for the same tests when USE_ASYNC_IO, USE_ASYNC_COMPRESS, and USE_ASYNC_CIPHER are enabled and two general CPs and two CPACFs are online, and two zAAP specialty engines are online.

Table 12-8 Performance with all optimization options, two CPs, two zAAPs and the hardware provider

Elapsed time (sec)	MB/sec	zAAP utilization %	General CPU utilization %	MB/General + zAAP CPU sec
55.22	26.02	57.56%	1.68%	21.96

Table 12-9 is a repetition of Table 12-1 on page 279, shown here again for convenience, which lists the statistics when one general CP is online, no parallel processing options are enabled, and the hardware provider is not specified.

Table 12-9 Performance with one CP and no hardware provider 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	97.72	14.70	60.87%	24.15

Table 12-8 shows significant improvement over Table 12-9:

- Elapsed time was reduced by over 43%.
- MB/second increased by approximately 77%.
- A full 97% of the CPU activity was able to run on a zAAP. For installations with multiple CPs and zAAPs configured, this probably is the lowest cost option.

Conclusion

Two metrics that are key to the analysis of performance are the elapsed time needed to complete a task, and the CPU time used to complete that task. This appendix demonstrates some of the techniques that can used to achieve enhanced performance when using the IBM Encryption Facility for z/OS OpenPGP support, taken as a typical industry-class Java application.

Clearly, all installations where CPACF are enabled should exploit this technology. Installations with multiple general CPs and zAAP processors online can benefit from enabling all of the parallel processing options for OpenPGP support.

Index

Symbols

A

B

C

© Copyright IBM Corp. 2008. All rights reserved.

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X

Z

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 Redbooks” on page 290. Note that some of the documents referenced here may be available in softcopy only.

- *Java Stand-alone Applications on z/OS Volume 1,* SG24-7177
- *Java Stand-alone Applications on z/OS Volume 2*, SG24-7291
- *System z Cryptographic Services and z/OS PKI Services*, SG24-7470
- *IBM System Storage TS1120 Tape Encryption: Planning, Implementation, and Usage Guide*, SG24-7320

Other publications

These publications are also relevant as further information sources:

- *z/OS UNIX System Services Planning*, GA22-7800
- *z/OS Security Server RACF General User’s Guide*, SA22-7685
- *z/OS Security Server RACF Security Administrator’s Guide*, SA22-7683
- *z/OS Security Server RACF Callable Services*, SA22-7691
- *z/OS Security Server RACF Diagnosis Guide*, GA22-7689
- *z/OS Security Server RACF Command Language Reference*, SA22-7687
- *z/OS Integrated Security Services Enterprise Identity Mapping (EIM) Guide and Reference*, SA22-2875
- *z/OS Cryptographic Services Integrated Cryptographic Services Facility Administrator's Guide*, SA22-7521
- *z/OS Cryptographic Services ICSF Application Programmer's Guide*, SA22-7522
- *z/OS Cryptographic Services ICSF Trusted Key Entry PCIX Workstation User’s Guide*, SA23-2211
- *z/OS Cryptographic Services Integrated Cryptographic Service Facility Writing PKCS #11 Applications*, SA23-2231
- *z/OS Cryptographic Services System Secure Sockets Layer Programming*, SC24-5901
- *System z10 Enterprise Class Processor Resource/Systems Manager Planning Guide*, SB10-7153
- *IBM Encryption Key Manager Component for the Java Platform Introduction, Planning, and User's Guide*, GA76-0418

© Copyright IBM Corp. 2008. All rights reserved.

Online resources

These Web sites are also relevant as further information sources:

- A basic introduction to Java programming with JNI, available at IBM developerWorks

  http://www-128.ibm.com/developerworks/edu/j-dw-javajni-i.html
- Java Record I/O API download

  http://www-03.ibm.com/servers/eserver/zseries/software/java/jrio/overview.html/
- zAAP specialty engine information

  http://www-03.ibm.com/systems/z/zaap/
- Java JAAS information

  http://www-03.ibm.com/servers/eserver/zseries/software/java/jaas.html

  http://www.ibm.com/developerworks/java/jdk/security/60/secguides/JaasDocs/api.html
- Java SAF classes information

  http://www-03.ibm.com/servers/eserver/zseries/software/java/j5security.html
- Java Jsec API download

  http://www.ibm.com/servers/eserver/zseries/zos/racf/racfjsec/doc/index.html
- EIM APIs download

  http://www-1.ibm.com/servers/eserver/security/eim/availability.html
- IBMJSSE2 Guide

  http://www.ibm.com/developerworks/java/jdk/security/60/secguides/jsse2Docs/JSSE2RefGuide.html#plug

How to get Redbooks

You can search for, view, or download Redbooks, 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

Java Security on z/OS - The Complete View

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

Java Security on z/OS - The Complete View

Comprehensively describes z/OS security services for Java applications

Provides use cases illustrated with Java program examples

Discusses industry-class Java applications

This IBM Redbooks publication describes and explains which z/OS security services can be exploited by Java stand-alone applications executing on z/OS. It is intended for experienced z/OS users with a moderate knowledge of Java, and experienced Java users with some knowledge of z/OS. For experimentation and customization it provides use cases that were composed and tested on a z/OS platform at z/OS V1R10 and SDK 6 SR1.

The book describes the role of the major infrastructure components such as Security Manager, Access Controller, Class Loader and Byte Code Verifier. It addresses specific z/OS-provided facilities including the JZOS Toolkit and Java record I/O (JRIO), and explains how they fit within both security models. Java Authentication and Authorization Services (JAAS) is covered and practical examples illustrating its use in z/OS, including the LoginModules that interact with the SAF interface, are given. The relationship of these services to z/OS built-in security functions such as APF, Program Control, and so on is explained. The specific security-relevant services provided to Java applications executing on the z/OS platform are covered, along with practical examples of their setup and use.

Java SAF classes, the JSec API, exploitation of RACF PassTickets, and the use of the z/OS Enterprise Identity Mapping (EIM) infrastructure are explained. Exploitation of z/OS integrated hardware cryptography by Java applications is detailed, along with numerous practical examples of the use of these services. z/OS cryptographic key management features are also discussed.

Finally, the book addresses two industry-class IBM Java products that exploit z/OS hardware cryptography, IBM Encryption Key Manager and IBM Encryption Facility for z/OS OpenPGP Support, and highlights the exploited functionalities and performance optimization.

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

SG24-7610-00 ISBN 0738431869