Entrust Authority™

Security Toolkit for Java™ Programmer's Guide

Software Release: 6.1

Document Issue: 1.1

Date: March 2003

Contents

Chapter 1

Introduction

This section of the Programmer's Guide introduces the **Entrust Authority™ Security Toolkit for Java™** (formerly Entrust/Toolkit™ for Java™) and explains where you can find documentation related to the Toolkit and to cryptography in general. Also included in this section is information about how to obtain technical support and how to provide comments and suggestions about the Toolkit and its documentation.

Topics in this section:

- About this guide
- Related documentation
- Recommended reading
- Technical support
- Documentation feedback

About this guide

The **Security Toolkit for Java Programmer's Guide** is a handbook for application developers who want to build applications that take advantage of the cryptographic operations in a Public Key Infrastructure (PKI). The Security Toolkit for Java, referred to as the Toolkit throughout this document, is one of a number of Entrust Authority™ Toolkits. The Toolkit uses the Java programming language, and its cross-platform capabilities, to give you the ability to add trusted security to your applications. The interoperability of the Toolkit is further enhanced in Release 6.1 with the following capabilities:

- The ability to apply XML encryption based upon proposals under review by the World Wide Web Consortium's (W3C) XML Encryption Working Group and specifically on the XML Encryption Syntax and Processing W3C Candidate Recommendation 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-core-20020802/).
- The ability to apply XML digital signatures (detached, enveloping, and enveloped) in accordance with the XML syntax and processing rules for creating and representing digital signatures — as specified in the World Wide Web Consortium's (W3C) XML-Signature Syntax and Processing W3C Recommendation 12 February 2002 (http://www.w3.org/TR/xmldsig-core/) and Decryption Transform for XML Signature W3C Candidate Recommendation 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-decrypt-20020802).
- The ability to to connect users to the Directory and CA key management servers of a PKI using the HTTP or HTTPS communications protocols.
- Support for Microsoft® Active Directory®, so that developers can include in their applications, the capability to read digital certificates and certificate revocation lists (CRLs) from Microsoft Active Directory.
- The recognition of CRL distribution points (CDPs) expressed as LDAP and HTTP URLs in X.509 digital certificates. Earlier versions of the Toolkit recognized CDPs only in the X.500 distinguished name format.
- An operating mode designed to conform to security level 1 of the Federal Information Processing Standards (FIPS) PUB 140-2: Security Requirements for Cryptographic Modules, May 2001.
- The ability to connect users to the Directory and Certification Authority (CA) key management servers of a PKI with HTTPS communications protocols secured using SSL.
- An implementation of the Java Security KeyStore model — a container, or common repository in a file or in memory, that holds cryptographic keys and certificates — providing a high-level interface for gaining access to credentials (Entrust Profile, PKCS #12, and PKCS #11) and certificate stores.
- Support for developing and building applications with the Java 2 Software Development Kit (J2SDK) version 1.4 and the Java Cryptography Extension (JCE) version 1.2.2.
- Support for the Advanced Encryption Standard (AES) algorithm using 128, 192, and 256-bit AES in both cipher block chaining (CBC) and electronic code book (ECB) modes.

- Support for Elliptic Curve Cryptography (ECC) in the form of the elliptic curve digital signature algorithm — providing digital signature capabilities using the JCE for both stand-alone applications and applications that work with Entrust/PKI.

In addition to the capabilities listed, the `etjava\examples\soap` folder includes a sample application that demonstrates how the Toolkit's XML digital signature capabilities work with the signature and messaging features of the Simple Object Access Protocol (SOAP). The sample application relies on Apache SOAP (http://xml.apache.org/soap/index.html) , a subproject of the Apache XML Project (http://xml.apache.org/index.html). Refer to the SOAP sample documentation for more details.

The Programmer's Guide describes the architecture of the Toolkit and contains information about how to use the Toolkit's application programming interface (API).

To use the information in this Guide, you should have a basic understanding of cryptography and key management in a PKI, and be familiar with the Java programming language.

Note

The procedures in the Programmer's Guide use code fragments to illustrate the specific steps you should consider including in your application. Not all code necessary for a complete program is shown. The `examples` folder in the Toolkit's installation directory contains complete code samples.

Related documentation

The Developers' section (https://www.entrust.com/developer/index.htm) of the Entrust Web site is a good source of background information about the use of Entrust Authority Toolkits and the cryptographic standards and algorithms they implement. The site also contains information about cryptography in general and about Entrust/PKI. The resources section (http://www.entrust.com/resources/index.htm) of the Web site has links to technical documents, white papers, and articles covering a variety of cryptography and security topics.

Accompanying the Toolkit is a complete set of Javadoc-generated HTML files that describes the public interface of the Toolkit's API. The **Description** link on the overview page explains the dependencies among the Toolkit's jar files, and displays tables that show their contents. The Javadoc reference documentation includes detailed information about the IAIK (the Institute for Applied Information Processing and Communications in Austria — http://www.iaik.tu-graz.ac.at) packages included in the Toolkit.

The Toolkit package includes a comprehensive set of sample applications and accompanying documentation. Refer to the **StartHere** page in the `etjava` folder for a list of samples and links to their respective readme documents.

Recommended reading

The following list of resources on the World Wide Web provide reference material you

might find useful when working with the Toolkit's API:

- Home page for Java resources (http://java.sun.com/)
- Java Developer Connection™ (http://developer.java.sun.com/developer/)
- API specification of the Java 2 Platform, Standard Edition, version 1.3 (http://java.sun.com/j2se/1.3/docs/api/index.html) and version 1.4 (http://java.sun.com/j2se/1.4/docs/api/index.html).
- Java Cryptography Extension (JCE) Reference Guide for the J2SDK , Standard Edition, version 1.4 (http://java.sun.com/j2se/1.4/docs/guide/security/jce/JCERefGuide.html)
- Documentation for the tools included with the Java 2 SDK version 1.3 (http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html) and version 1.4 (http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html).

  Note

  If you are new to Java, there is information on this page about setting the `-classpath` option and the `CLASSPATH` environment variable.

- Java Security page (http://java.sun.com/security/).
- Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation 6 October 2000 (http://www.w3.org/TR/2000/REC-xml-20001006).
- XML Path Language (XPath) Version 1.0 W3C Recommendation 16 November 1999 (http://www.w3.org/TR/xpath).
- Namespaces in XML World Wide Web Consortium 14 January 1999 (http://www.w3.org/TR/REC-xml-names/).
- XML Encryption Syntax and Processing W3C Candidate Recommendation 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-core-20020802/).
- XML Signature Syntax and Processing W3C Recommendation 12 February 2002 (http://www.w3.org/TR/xmldsig-core/).
- Decryption Transform for XML Signature W3C Candidate Recommendation 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-decrypt-20020802).

Technical support

Entrust recognizes the importance of providing quick and easy access to our support resources. Our developer support program has been created to provide a flexible and cost-effective way for developers to get the support they need by phone or email.

Please visit the Customer support section (http://www.entrust.com/support/index.htm) of the Entrust Web site for further information.

Documentation feedback

We are continually trying to improve the quality and coverage of information related to Entrust Authority Toolkit information products. We are particularly interested in how you are using Entrust Authority Toolkits in your development environment and whether the information presented in this document provides direction in the areas in which you

are interested.

Please send an email message to documentation@entrust.com if you have any comments, questions, or other remarks about any aspect of Entrust Authority Toolkit information products. We look forward to hearing from you!

Chapter 2

Toolkit overview

This section presents information about how the Toolkit is organized, discussing its architecture and some of the cryptographic operations commonly performed by application developers working with this Toolkit.

Topics in this section:

- Overview
- Architecture
- Modularization
- Toolkit utilities

Overview

Entrust Authority Toolkits help you develop and build applications that protect the privacy, integrity, and authenticity of information communicated among the workstations and servers of a network whose security is managed using a Public Key Infrastructure (PKI). A PKI accomplishes its management tasks, which include the generation, transmission, and storage of its users' cryptographic keys, using a Certification Authority (CA), secure encryption and decryption algorithms to provide privacy, and digital signatures to assure the integrity and authenticity of data.

The Security Toolkit for Java gives an application access to the underlying security structure of a PKI and its automated key and life cycle management capabilities, relieving you of the need to write code to handle the application's low-level cryptographic and security life cycle functions.

The Toolkit offers both high-level and low-level APIs that enable a Java application to perform security related tasks. The Toolkit adheres to the Java Cryptography Architecture (JCA) (http://java.sun.com/security/) and supplements it with numerous cryptographic algorithms. If you are familiar with other Entrust Authority Toolkits, the Security Toolkit for Java combines the capabilities of the EntrustFile™ Toolkit and the Entrust/Toolkit for SSL/TLS. Such a blend of capabilities allows the applications developed with the Toolkit to use open, publicly available standards such as the Public Key Cryptography Standard #7 (PKCS #7) — for store and forward applications, PKCS #12 — for importing and exporting critical information (private keys and certificates), and the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols.

The applications you build with the Toolkit can perform the following high-level tasks:

- The management of users' credentials (the user's private and public cryptographic keys, and information that describes a user's relationship to the CA) — involving the creation and recovery of credentials (including those on hardware tokens), key rollover, distinguished name (DN) changes, CA key updates, certificate validation, and the import and export of credentials that conform to PKCS #12.
- The creation and processing of PKCS #7 messages — messages and data that have been cryptographically secured, digital signatures, and digital envelopes.
- The incorporation of S/MIME capabilities into your application — for exchanging messages and data using transport protocols capable of conveying MIME data.
- The protection of XML data — including arbitrary XML data, complete XML documents, XML elements, and the content of XML elements.
- The creation of XML digital signatures — adding authentication and data integrity to the data you exchange across the World Wide Web.
- The use of Secure Sockets Layer (SSL) version 3.0 and Transport Layer Security (TLS) version 1.0 communications, with socket control — providing security and privacy on the Internet and other networks.

The Toolkit is not based on the same runtime components as are the Entrust Authority Toolkit C and C++ products. Instead, the Java Cryptographic Extension (JCE) is the cryptographic framework that underlies the Security Toolkit for Java. The JCE supports secure streams and provides implementations for algorithms supporting key generation

and agreement, symmetric and asymmetric encryption, and Message Authentication Codes (MACs).

Two versions of the JCE are available from Sun Microsystems:

- JCE version 1.2 — an optional extension to versions 1.2 and 1.3 of the J2SDK and available only in Canada and the United States.
- JCE version 1.2.2— (maintenance release of JCE version 1.2.1) a self-contained optional extension to versions 1.2 and 1.3 of the J2SDK, or part of version 1.4 of the J2SDK, incorporating a security mechanism to ensure that it recognizes only qualified cryptographic security providers.

The architecture of version 1.2.2 of the JCE allows trusted cryptographic service providers (CSP) to be plugged into its framework. A CSP comprises one or more packages containing classes that provide concrete implementations of cryptographic concepts encapsulated in the JCE. Trusted CSPs are those whose code has been signed by certification authorities at Sun Microsystems or at IBM. The Toolkit's CSP, Entrust, includes a certificate that authenticates the Entrust provider to the JCE when it is loaded, making its algorithm implementations available for use in your applications. The Security Toolkit for Java will continue to work with JCE version 1.2. The JCE version 1.2.2 limits your use of cryptographic algorithms and their cryptographic strengths depending upon the import restrictions in your geographical location. To determine which, if any, restrictions apply, the JCE reads jurisdiction policy files that are part of the JCE software.

Algorithm selection among the following algorithms is made available through the JCE.

Algorithms

Message Digest (hash) algorithms	SHA1
	MD2
	MD5
Public key algorithms (asymmetric encryption algorithms)	RSA
	DSA
	ECDSA
Symmetric encryption algorithms	AES
	DES
	Triple-DES
	IDEA
	CAST-128
	RC2
	RC4

Using the JCE with versions 1.2 or 1.3 of the J2SDK

Sun Microsystems provides two options to install the JCE version 1.2.2 for use with versions 1.2 or 1.3 of the J2SDK:

1 Putting the JCE framework and providers on the classpath, as a bundled optional package (bundled extension)

2 Installing the JCE as an installed optional package (installed extension)

Note

Optional packages were formerly called standard extensions — refer to the Sun Microsystems document entitled **Extension Mechanism Architecture** (http://java.sun.com/j2se/1.3/docs/guide/extensions/spec.html) for more information about installed and bundled optional packages.

Refer to the Sun Microsystems document entitled **Support for Extensions and Applications in Version 1.2 of the Java Platform** (http://java.sun.com/products/jdk/1.2/docs/guide/extensions/spec.htmll) for information about installed and bundled extenstions.

The installation instructions (http://java.sun.com/products/jce/jce122_install.html) for the JCE version 1.2.2 provide information about these two options. If you are using the JCE as an installed optional package, you can find out about, and download, jurisdiction policy files from the JCE 1.2.2 Web page http://java.sun.com/products/jce/index-122.html).

Using the JCE with version 1.4 of the J2SDK

The jurisdiction policy files that are shipped with the J2DSK version 1.4 allow users to implement cryptographic algorithms that are strong, but limited. If you are using version 1.4 of the J2SDK and its integrated JCE, you must use the JCE's unlimited strength jurisdiction policy files. You can find out about, and, if you are permitted to do so from your locality, download the unlimited strength jurisdiction policy files from the **JCE for the J2SDK version 1.4** Web page (http://java.sun.com/products/jce/index-14.html).

The readme document (`readme.txt`) that is part of the unlimited strength jurisdiction policy files package, describes how to install the policy files on your computer. The document also contains advice concerning where to obtain information about import and export restrictions governing cryptographic software.

You can find links to Sun Microsystems' documentation for all versions of the JCE at the Java Cryptography Extension Web site (http://java.sun.com/products/jce/).

Interoperability

The Security Toolkit for Java is able to integrate cryptographic applications with any PKI that offers support for the following open standards:

- The Internet X.509 Public Key Infrastructure Certificate Management Protocols (PKIX-CMP). PKIX-CMP is fully explained in RFC 2510.
- The Public Key Cryptography Message Standard (PKCS #7)
- The Certification Request Standard (PKCS #10)
- The Personal Information Exchange Syntax Standard (PKCS #12)

  Note

  The Toolkit has been tested with Microsoft's PKCS #12 files (files having a `.pfx` file name extension) and with Netscape's PKCS #12 files (files having a `.p12` file name extension).

- X.509 v3 certificates

At a lower level, you can use the Toolkit to work directly with raw Distinguished Encoding Rules (DER) files. DER, a subset of the Basic Encoding Rules (BER), is defined in ISO standard X.690.

The Toolkit implements the following standards and protocols:

- PKCS #5 — Password-based Cryptography Standard (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-5/index.html)
- PKCS #7 — Cryptographic Message Syntax Standard (http://www.ietf.org/rfc/rfc2315.txt?number=2315)
- PKCS #10 — Certification Request Syntax Standard (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-10/index.html)
- PKCS #11 — Cryptographic Token Interface Standard (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html)
- PKCS #12 — Personal Information Exchange Syntax Standard (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-12/index.html)
- Secure Sockets Layer (SSL) protocol, version 3.0 (http://home.netscape.com/eng/ssl3/ssl-toc.html)
- Transport Layer Security (TLS) protocol, version 1.0 (http://www.ietf.org/rfc/rfc2246.txt)

  Note

  TLS version 1.0 is an updated version of SSL version 3.0.

- Internet X.509 Public Key Infrastructure Certificate and CRL Profile (RFC 2459 — http://www.ietf.org/rfc/rfc2459.txt?number=2459)
- S/MIME version 2 Message Specification (RFC 2311) and S/MIME version 2 Certificate Handling (RFC 2312 — http://www.ietf.org/rfc/rfc2312.txt)
- S/MIME version 3 Message Specification (RFC 2633 — http://www.ietf.org/rfc/rfc2633.txt) and S/MIME version 3 Certificate Handling (RFC 2632 — http://www.ietf.org/rfc/rfc2632.txt)
- The Cryptographic Message Syntax (CMS) (RFC 2630 — http://www.ietf.org/rfc/rfc2630.txt)
- XML Encryption Syntax and Processing W3C Candidate Recommendation 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-core-20020802/)

- XML Signature Syntax and Processing W3C Recommendation 12 February 2002 (http://www.w3.org/TR/xmldsig-core/)
- Decryption Transform for XML Signature W3C Candidate Recommendation 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-decrypt-20020802)

Using the AES algorithm with the Security Toolkitfor Java

If you intend to develop an application that takes advantage of support for the Advanced Encryption Standard (AES) algorithm in the Security Toolkit for Java Release 6.1 and the EntrustFile Toolkit Release 6.0, you should be aware of how these products support the AES algorithm.

EntrustFile — AES support

This Toolkit can encrypt data using the AES algorithm in cipher block chaining (CBC) mode with a key length of 256-bits. It can also decrypt data that has been encrypted with the AES algorithm in CBC mode with key lengths of 128-bits, 192-bits, and 256-bits. The Toolkit cannot decrypt data that has been encrypted in electronic code book (ECB) mode.

Security Toolkit for Java — AES support

This Toolkit can encrypt data using the AES algorithm with key lengths of 128-bits, 192-bits, and 256-bits in both CBC and ECB modes. It is also capable of decrypting data that has been encrypted with the AES algorithm in both CBC and ECB modes with key lengths of 128-bits, 192-bits, and 256-bits.

Data encrypted with the AES algorithm using the Security Toolkit for Java Release 6.1 that you want to be able to read using an application developed with the EntrustFile Toolkit Release 6.0, must be encrypted in CBC mode only. Do not use the following algorithm IDs in your application:

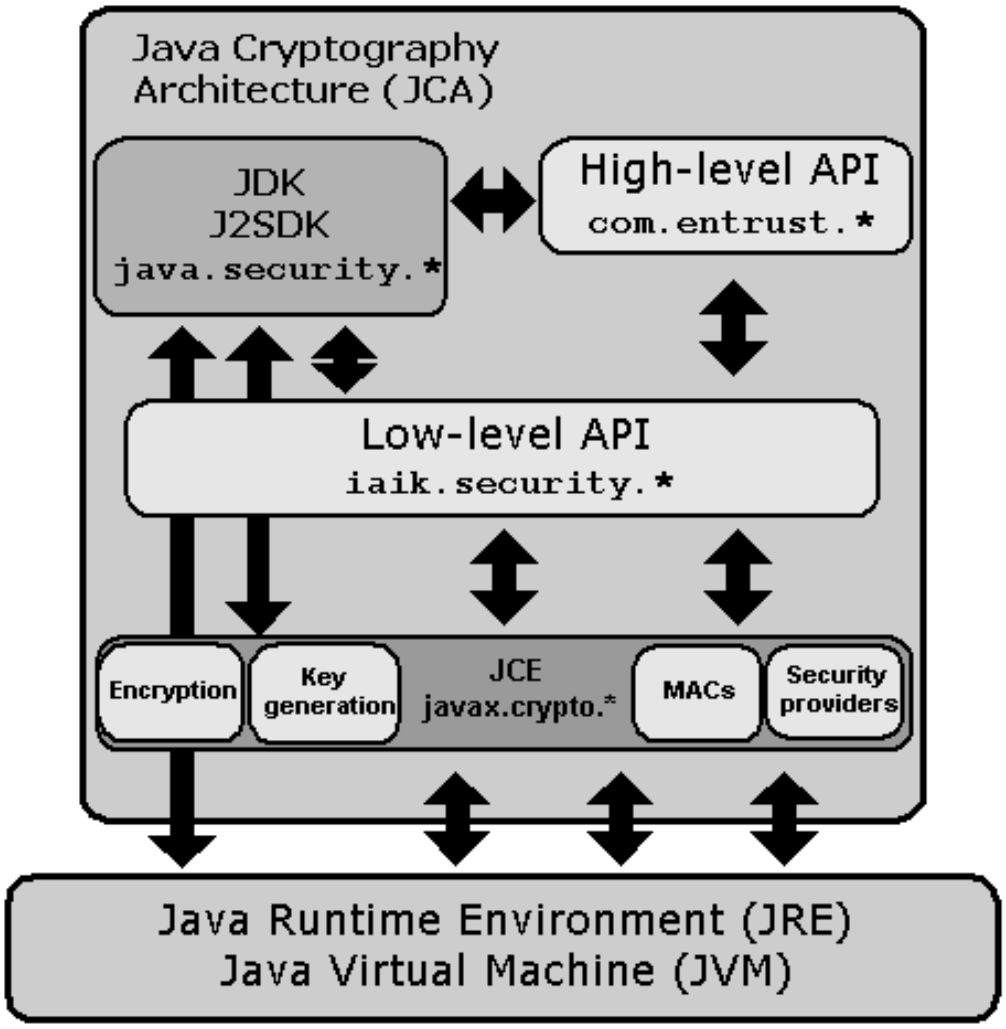
- `iaik.asn1.structures.AlgorithmID.aes_128_ECB`
- `iaik.asn1.structures.AlgorithmID.aes_192_ECB`
- `iaik.asn1.structures.AlgorithmID.aes_256_ECB`

Architecture

Java Security is a collection of Application Programming Interfaces (APIs), some of them part of the Java 2 Software Development Kit (J2SDK), that underlie the Toolkit and provide the base for its cryptographic services. Such APIs include the built-in security pacakages of the J2SDK, the Java Cryptography Extension (JCE), and the Java Secure Sockets Extension (JSSE). The Java Cryptography Architecture (JCA), which governs the design of the security packages of the J2SDK, is extended by the JCE to include APIs for digital signatures, hash functions, encryption, key exchange, and Message Authentication Codes (MAC). Using the Service Provider Interface (SPI) of the JCA, custom cryptographic service providers (usually a package or set of packages), such as those provided by the Toolkit, can implement the cryptographic features of the JSA to invoke their own message digests, encryption algorithms, and other utilities that a

security application might require.

Entrust/Toolkit for Java basic architecture



High-level classes

The Toolkit's high-level API, found in the `com.entrust.toolkit` package and its subpackages, provides classes that implement frequently used cryptographic tasks. These classes, because of their high level of abstraction, are generally more useful than their low-level IAIK or JCA counterparts and you will rarely need to use the raw JCA interfaces. The PKCS #7 classes make extensive use of streams to represent both sources and repositories of data.

You will find the high-level classes of the Toolkit in the `com.entrust.toolkit` package and its subpackages. The `User` class represents an entity, or end user, in a PKI domain and is the primary class in the high-level API. By instantiating a `User`, you have access, by way of accessor methods, to the user's verification public certificate and

encryption public certificate, the user's decryption private key and signing private key, a securely obtained copy of the Certification Authority's (CA) verification public certificate, and methods that allow you to manage a user's credentials. There is one `User` instance per entity, but any number of end users can be logged in at the same time without interfering with each other. If there are several threads running concurrently, each thread uses, or accesses, the single `User` instance, and, to ensure proper credentials management, only one thread has control of, or responsibility for, that `User` instance.

Cryptographic service providers

The Toolkit provides the following custom cryptographic service Providers:

- The **Entrust** cryptographic service Provider — supports specialized implementations of the RSA and DSA algorithms.
- The **IAIK** cryptographic service Provider — implements key generation and other utilities, as well as the most commonly used symmetric encryption algorithms and message digests (hash functions).
- **EntrustPKCS11**
- **IAIKJSSEProvider**

Note

In this release of the Toolkit, the **Entrust** CSP is not registered automatically as the default cryptographic service provider. It is registered after the default SUN provider in the second position of the preference order in which providers are searched for requested algorithms.

If you implement your own Providers, you must add them to the list of installed Providers. Use the `java.security.Security.addProvider()` method to add a new Provider to the Provider list. Refer to the Java API reference documentation for more details.

Initializing the Toolkit

To initialize the Toolkit and prepare it to perform cryptographic operations, use the `com.entrust.toolkit.security.provider.Initializer` class and its methods. Your applications should retrieve an instance of `Initializer` and invoke its `setProviders(int mode)` method before the application performs any cryptographic operations. To do this, use the `Initializer.setProviders` method. For example:

```
Initializer.getInstance().setproviders(Initializer.<appropriate mode>);
```

The Toolkit can operate in one of three modes:

- Uninitialized (`MODE_UNINITIALIZED`) — none of the Toolkit's security providers is available
- Normal (`MODE_NORMAL`) — Entrust and IAIK providers are available
- Token (`MODE_TOKEN`) — private cryptographic operations are performed on a hardware token

Some of the Toolkit's operations perform initialization internally. The `CredentialReader` constructor, for example, sets the Toolkit mode to `MODE_NORMAL`

and the `TokenReader` constructor sets the Toolkit mode to `MODE_TOKEN`.

Note

Use this method to initialize the Toolkit if you do not want to perform a login operation (if you want to perform cryptographic operations immediately, for example). When you invoke the `User.login` method, the cryptographic providers are set automatically as required.

For detailed information about the `Initializer` class, and other high-level classes of the Toolkit, refer to the Javadoc reference documentation.

Low-level classes

The low-level API of the Toolkit comprises the `IAIK` packages, the `java.security` packages, and the `javax.crypto` packages. Classes in the low-level API support the Public Key Cryptography standards and give you access directly to the JCE and its algorithms, to the IAIK cryptographic service Provider, and to IAIK's SSL and XML digital signatures interfaces.

Object identifiers

The Toolkit includes a class (`iaik.asn1.ObjectID`) that implements the ASN.1 `OBJECT IDENTIFIER` universal type. The class generates object identifiers (OIDs) for X.500, PKCS #7, PKCS #9, PKCS #12, and PKIX objects. You can use these objects in your applications by specifying the name corresponding to the OID. The Javadoc reference documentation for the `iaik.asn1.ObjectID` class lists the OIDs that the Toolkit registers by default. Your application should register, by creating new `ObjectID` instances and upon initialization only, any additional OIDs it might need.

Modularization

You can use the Toolkit to create both Java applications and Java applets. The Toolkit classes are packed into Java archive (`jar`) files to provide a modular distribution of the Toolkit's capabilities. Such an arrangement gives you the ability to work with only those jar files containing the classes whose capabilities you want your application to use. This is particularly useful for minimizing download size and reducing download time when you are writing applets.

Using the Toolkit's jar files

When you are building or running an applet or application that uses the Toolkit API, ensure that the appropriate jar files are specified in your operating system's `CLASSPATH` environment variable, or that you specify the jar files in the `-classpath` parameter of the Java compiler (`javac`), and the Java interpreter (`java`). If you need information about either of these techniques, read the appropriate documentation on Sun's Tools and Utilities page (http://java.sun.com/products/jdk/1.4/docs/tooldocs/tools.html).

Two jar files should always be in your classpath: `entbase.jar` and `entuser.jar`

For example, if you are developing an application that uses the PKCS #7 capabilities of

the Toolkit, you must include the following jar files in your classpath:

- `entbase.jar`
- `entuser.jar`
- `entp7.jar`

The Toolkit distribution includes both signed and unsigned versions of it's jar files, located in the `etjava\lib` and `etjava\lib\unsigned` folders respectively.

Signed jar files

The `etjava\lib` folder contains the signed jar files you should use if you are using any of the environments listed in this subsection. When you work with the signed jar files, you must use the JCE's unlimited strength jurisdiction policy files. You can find out about, and, if you are permitted to do so from your locality, download the unlimited strength jurisdiction policy files from the JCE for the J2SDK version 1.4 Web page (http://java.sun.com/products/jce/index-14.html).

Use the Toolkit's signed jar files when your environment is one of the following:

- J2SDK version 1.2 with version 1.2.2 of Sun's JCE
- J2SDK version 1.3 with version 1.2.2 of Sun's JCE
- J2SDK version 1.4 (which includes version 1.2.2 of Sun's JCE)
- IBM Java developer kit version 1.3 (http://www-105.ibm.com/developerworks/tools.nsf/dw/java-devkits-byname) for IBM platforms and operating systems
- OS/390 with IBM Java developer kit version 1.3.1

Unsigned jar files

The etjava\lib\unsigned folder contains unsigned versions of the Toolkit's jar files. Use the unsigned jar files when you are working with any of the environments listed in this subsection, and include `entjavaxcrypto.jar` in your classpath. The `entjavaxcrypto.jar` file contains the IAIK implementation of classes in the `java.security and javax.crypto` packages.

Use the Toolkit's unsigned jar files (including `entjavaxcrypto.jar`) when your environment is one of the following:

- J2SDK version 1.2
- J2SDK version 1.3
- J2SDK version 1.2 or 1.3 with version 1.2 of Sun's JCE (export restricted version)

You do not need `entjavaxcrypto.jar` in your class path if you have installed the JCE version 1.2.2 as an installed optional package (see the Sun Microsystems document entitled **Extension Mechanism Architecture** at http://java.sun.com/j2se/1.3/docs/guide/extensions/spec.html) or if you are using the J2SDK version 1.4 unless you use the `-Xbootclasspath` option with the `java` launcher or `javac` compiler, as you would when working with XML and the Apache Xalan XSLT processor. Refer to the subsection entitled **Using Apache Xalan-Java in a J2SDK version 1.4 environment** in the **XML Digital Signatures and XML Encryption** samples document (`etjava\examples\xml\xml_readme.html`) for information about how to use Xalan

and the `-Xbootclasspath` option.

Jar file capabilities

The **File locations** section in the **Readme** file (`etjava\docs\ettkjava_readme.html`) lists the jar files that make up the Toolkit and briefly describes their capabilities.

Jar file dependencies

The Javadoc reference documentation contains tables showing the dependencies among the Toolkit's jar files, details about their capabilities, and the packages they contain. Refer to the **StartHere** document (`etjava\StartHere.html`) for links that directly open these sections in the reference documentation.

Toolkit utilities

This section describes Toolkit utilities that you might find useful when designing, building, and debugging Toolkit-based applications.

Using configuration files

You can use configuration files with the applications you build using the Toolkit. Configuration files allow you to modify the data that an application operates upon without having to change, or recompile, the application's source code.

Structuring Configuration files

Configuration files provide the means to separate applications from the data they use, so that you can edit that data without changing the source code of the application. A convenient format for referring to an application's data in a configuration file is to use key-value pairs in a structure similar to that of initialization (`.ini`) files, grouping related data into sections.

```
[Section name]
key1 = value1
key2 = value2
key3 = value3
```

The S/MIME sample application in the `etjava\examples\smime` folder contains an example of a configuration file called `smime_example.ini`.

```
[SMimeSend]
attrToFind=userCertificate
searchbase=<searchbase X500>
searchexpr=<searchexpression e.g. CN>
to=<email of recipient>
from=<email adress of sender>
host=<mailhost>
Profile=<epf filename of sender>
password=<password for login of sender>
firstName=Certificate
lastName=Request

[SMimeShow]
```

```
Profile=<epf filename of recipient>
protocol=<protocol of your mailserver>
host=<mailhost>
user=<recipient>
user_password=<password of recipient>
recipient_password=<password for login of recipient>
mbox=<name of mailbox>
```

The `smime_example.ini` file has two sections: `[SMimeSend]` and `[SMimeShow]`, which contain data used by the email sample applications, `SMimeSend.java` and `SMimeShow.java`. Under each section is a list of keys, or attributes, with associated values. Applications can operate on the data by referring to the particular section and key in the configuration file.

Using the IniFile class

The Toolkit includes several utility classes in the `com.entrust.toolkit.utility` package, among which is the `IniFile` class. Working with the methods in the `IniFile` class, you can create `IniFile` objects and use them to manipulate the entries in configuration files.

To make use of a configuration file, include the following steps in your application:

1 Create a configuration file with sections and key-value pairs.

2 Import the `IniFile` class into your source code.

```
import com.entrust.toolkit.utility.IniFile;
```

3 Create a reference to the configuration file.

```
IniFile properties = new IniFile(<configuration file>);
```

Note

The `IniFile` class has three constructors:

- `public IniFile()` — initializes a configuration file.
- `public IniFile(java.lang.String filename)` — initializes a named configuration file.
- `public IniFile(java.io.InputStream is)`— initializes the configuration file pointed to by the input stream.

4 Read data from the configuration file.

```
String to = null;
String from = null;

to = properties.getString("SMimeSend", "to");
from = properties.getString("SMimeSend", "from");
```

The Javadoc reference documentation for the `com.entrust.toolkit.utility.IniFile` class contains a complete list of the methods available for you to work with configuration files.

Using the 'trace' system property

The Toolkit's `trace` system property allows you to see the step-by-step results of Toolkit operations that would normally be hidden from view. Several Toolkit classes recognize the `trace` system property:

- `com.entrust.toolkit.x509.directory.JNDIDirectory`
- `com.entrust.toolkit.x509.revocation.X509CRLRS`
- `com.entrust.toolkit.x509.certstore.CollectionCS`
- `com.entrust.toolkit.x509.certstore.CertificateGraph`
- `com.entrust.toolkit.util.HttpManagerClient`
- `com.entrust.toolkit.util.HttpsManagerClient`
- `com.entrust.toolkit.util.HttpDirectoryClient`
- `com.entrust.toolkit.xencrypt.core.TransformImplDecryption`
- `com.entrust.toolkit.security.fips.SecurityEngine`
- `com.entrust.toolkit.security.fips.JarAuthenticator`
- `com.entrust.toolkit.security.provider.Initializer`
- `com.entrust.toolkit.security.provider.JCEVerifier`
- `com.entrust.toolkit.security.provider.JCEVerifierImpl`

The Java launcher's `-D` option allows you to set a system property. Refer to the J2SDK Web page (http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html) for a description of the `java` launcher and its options.

Set the `trace` system property by appending it to the class name and assigning it an integer greater than zero. The following example shows how to set the `trace` system property for the `TransformImplDecryption` class:

```
java -classpath <classpath>
-Dcom.entrust.toolkit.xencrypt.core.TransformImplDecryption.trace=1
<class containing your application's main method> <your application's options and arguments>
```

To see a complete example of the use of the `trace` system property, refer to the Readme document for the Decryption Transform for XML Signature sample application (`xml_decryptionTransform_readme.html`) in the `etjava\examples\xml\decryptionTransform` folder.

Chapter 3

User and credentials management

This section deals with the operations your application must perform to manage users of a PKI.

Topics in this section:

- Credentials
- Creating a set of credentials
- Identifying a user
- Recovering a user's credentials
- Using PKCS #12 credentials
- Working with hardware tokens PKCS #11
- Using the Entrust key store implementation
- Using Single Login
- Using the Server Login feature
- Using the Toolkit in FIPS mode
- Security considerations

Credentials

As the term relates to a public-key infrastructure (PKI), a **user** defines an entity that has been identified and approved by a Certification Authority (CA). In an Entrust PKI, an Entrust Profile is a set of data that contains an Entrust user's public and private credentials and is a specific type of the more general concept of **credentials**.

The CA digitally signs and publishes the public portion of a user's credentials as digital certificates to the Directory, where they are accessible to other users within the CA's domain. The contents of a user's digital certificate include the following:

- User's distinguished name (DN)
- User's verification public key
- User's encryption public key
- CA's digital signature and verification public certificate

Such digital certificates can be likened to a passport or to a driver's licence, and constitute a user's public proof of identity.

Users keep secret the private portion of their credentials — the decryption private key and the signing private key. In an Entrust PKI, these private credentials remain in an Entrust Profile, which is stored as a file with an `.epf` file name extension. The contents of an Entrust Profile include the following:

- User's distinguished name (DN)
- User's decryption private key
- User's signing private key
- Decryption private key history
- Verification and encryption public certificates
- CA's verification public certificate
- Options specific to an Entrust PKI

Similar concepts hold true in a generic PKI. Digital certificates published in the PKI's Directory are signed by the CA and present a user's public credentials. The user's private cryptographic data (encryption private key and signing private key) are stored using conventions similar to that used in an Entrust Profile — the PKCS #12 portable storage format (*.p12 and *.pfx files), for example.

The term **credentials** is used throughout this document to describe a set of data in a generic PKI that defines an entity and that contains a user's critical cryptographic information. The Security Toolkit for Java treats Profiles and credentials alike, as the source of keying information for a particular entity, and decouples the key source from key usage by defining separate classes for user management (the `User` class) and credentials management (the `Credentials` package).

The application, or applet, you are building must be able to manage users and user credentials. User management tasks include such activities as the creation of credentials, logging in, the recovery of users' credentials, updating keys, and so on. This section describes procedures that enable your application to perform tasks related to user and credentials management.

Credential readers and credential writiers

The Toolkit uses the concepts of credential readers and credential writers to create, read from, and write to credentials, that are accessible through the I/O streams you have chosen to open in your code — a design that makes the Toolkit flexible and allows for the easy addition of new key sources. Classes in the Toolkit that extend the abstract superclasses of `CredentialReader` and `CredentialWriter` are listed and briefly described in the following table.

Credential readers and credential writers

Credential Readers (classes that are subclasses of `CredentialReader`)	**Credential Writers** (classes that are subclasses of `CredentialWriter`)
`StreamProfileReader` — reads Entrust Profiles from the specified input stream	`StreamProfileWriter` — writes an Entrust profile to the specified output stream
`FilenameProfileReader` — reads an Entrust Profile from an `.epf` file	`FilenameProfileWriter` — writes an Entrust Profile to an `.epf` file
`TokenReader` — reads credentials on a hardware token, smart card	`TokenWriter` — writes credentials to a smart card
`PKCS12Reader` — reads data formatted according to PKCS #12	`PKCS12Writer` — writes credentials that conform to PKCS #12 to the specified output stream
`UALCredentialReader` — reads `.ual` files created when a profile is bound to a computer (Server Login)	
`SingleLoginReader` — used with Entrust Login Interface (ELI), or Single Login, to read an Entrust Profile	
`CredentialCreator` — creates credentials	
`CredentialRecoverer` — recovers credentials	

Distinguished Name (DN) change, CA key update, and key rollover

The Toolkit handles various key management tasks automatically. Such tasks include Distinguished Name (DN) changes, Certification Authority (CA) key updates, and

automatic key rollover operations.

DN changes

The Toolkit triggers a DN change during user login if the following conditions exist:

- There is a connection to the Directory.
- There is a connection to the Manager (CA key management server).
- A `CredentialWriter` has been created to write the credentials in the form of an Entrust Profile (an `.epf` file)
- The Directory does not contain a user's encryption public certificate.

If there is a connection to the Directory, but no connection to the CA key management server, the Toolkit determines the DN change status and provides a warning (`WARNING_DN_CHANGE_REQUIRED`) if a DN change is required. The user's Entrust Profile remains valid until the DN change has been performed. When a DN change has occurred, the Toolkit provides the warning, `DN_CHANGE_PERFORMED`. A new Entrust Profile is written to disk only at the end of a successful DN change operation.

A DN change comprises the following steps:

1. Perform an encryption public key update.
2. Verify that the DN in the new encryption public certificate is different from the DN in the former encryption public certificate.
3. Update the signing key.
4. Verify that the DN in the new verification public certificate is the same as the DN in the new encryption public certificate.
5. Write the new Entrust Profile to disk.

CA signing private key updates

A Certification Authority has a signing key pair. The CA key update operation extends the lifetime of a CA verification public certificate and enables it to recover from a situation in which the root CA's signing private key has been compromised. Without this capability, the PKI would be required to cease operation when a CA signing private key expires, or when it becomes too old to be certain that it has never been compromised. The PKI might also update the CA signing private key to change the key algorithm or key size.

Note

After a CA's signing private key has been updated, users' certificates are signed with the new CA key during the next scheduled key update or key recovery. The CA can force an update of users' certificates if necessary — because of a CA key compromise, for example.

The Toolkit supports CA signing private key update operations by performing the following tasks:

- Detects when a CA signing private key update has occurred at the PKI.
- Verifies end user certificates after the CA signing private key update has occurred.
- Imports the new CA verification public certificate from the PKI into the client's

Entrust Profile before the old CA signing private key has expired or becomes revoked.

Automatic key rollover

Cryptographic key pairs should not be used forever — they must be updated periodically. The Toolkit handles the process of updating user keys automatically and transparently, requiring no user intervention.

When encryption keys are updated, a history of previous decryption keys is maintained. To ensure that data encrypted using former encryption keys is still accessible to users, the Toolkit automatically manages decryption key histories.

Key rollover occurs at login time when the `User.login` method is called and provided that the connections to the Directory and CA key management server have been set. Refer to step 4 of the Toolkit procedure in the section **Identifying a user** for details.

A number of methods in the `com.entrust.toolkit.User` class are available to you either to force a key update operation, or to determine if keys have already been updated. These are:

- `keyUpdateRequired()` — determines whether either key pair needs to be updated.
- `encryptionKeyUpdateRequired()` — determines whether a user's encryption public key needs to be updated.
- `updateEncryptionKeys()` — allows a user to update an encryption public key.
- `signingKeyUpdateRequired()` — determines whether a user's signing private key needs to be updated.
- `updateSigningKeys()` — allows a user to update a signing private key.
- `keyUpdatePerformed()` — allows users to determine whether either key pair has been updated.

The Toolkit provides the following messages related to the key rollover operations it performs:

- `WARNING_ENC_KEY_UPDATED` — to advise a user that his or her encryption public key has been updated.
- `WARNING_ENCRYPTION_KEY_NEEDS_UPDATE` — to advise a user that his or her encryption public key needs to be updated.
- `WARNING_SIGN_KEY_UPDATED` — to advise a user that his or her signing private key has been updated.
- `WARNING_SIGNING_KEY_NEEDS_UPDATE`— to advise a user that his or her signing private key needs to be updated.

Note

In PKI environments that use either proto-PKIX or PKIX-CMP protocols, updates to the encryption public key and signing private key in Entrust Profiles (`.epf` files) are always based upon client settings.

Creating a set of credentials

A credentials file is a set of critical cryptographic information about a user in a PKI. In an Entrust PKI, a user's credentials file, also called an Entrust Profile, or user Profile, is usually stored in a file, with an `.epf` file name extension. Creating a set of credentials for a new user involves obtaining, or creating, the credentials data and writing it to the credentials file. Before a user can begin the process of creating a new set of credentials, he or she must obtain a reference number and an authorization code from the PKI Administrator. The application you create should request these values from the user.

Note

In PKI environments that use PKIX-CMP protocols to create or recover credentials for users on both hardware tokens and as Entrust Profiles (`.epf` files), the encryption public key and signing private key are always based upon client settings unless the `User` instance was created using ECDSA signing keys (refer to the Javadoc reference for the `com.entrust.toolkit.credentials.CredentialCreator` class).

The following list is a summary of the procedure you should incorporate into your application to create a set of credentials:

- Obtain a reference number, authorization code, and password from the user.
- Instantiate a `User` object to represent the user.
- Establish the credential reader and writer depending on the type of credentials you are creating.
- Set up connections to the PKI CA key management server and to the Directory.
- Use a credential writer to write the new credentials.
- Log in the user to complete the procedure.

Use the credential writers as follows:

- `FilenameProfileWriter` — writes an Entrust Profile to a file.
- `StreamProfileWriter` — writes an Entrust Profile to a stream (a socket, for example).
- `PKCS12Writer` — writes credentials that conform to PKCS #12.
- `TokenWriter` — writes credentials to a smart card.

Toolkit procedure — creating an Entrust Profile

The following steps show the procedure your application should follow to create a new Entrust Profile (`.epf` file) using a `FilenameProfileWriter`:

1 Request the following information from the user of your application:
   - Authorization code (from the PKI Administrator)
   - Reference number (from the PKI Administrator)
   - Password

```
SecureStringBuffer pwd =
    new SecureStringBuffer(new StringBuffer(args[0]));
```

```
AuthorizationCode authCode =
    new AuthorizationCode(new StringBuffer(args[1]));
SecureStringBuffer refNo =
    new SecureStringBuffer(new StringBuffer(args[2]));
```

Note

Your application might use a GUI to prompt the user for this information — this example uses a simple command line interface.

2 Instantiate a `CredentialCreator` object, using the authorization code and reference number obtained in step 1.

```
CredentialCreator credCreator =
    new CredentialCreator(refNo,
                          authCode,
                          CredentialCreator.DSASignature,
                          algorithm_strength,
                          CredentialCreator.PKIX5Version);
```

Note

The `CredentialCreator` constructor also requires that you specify the signing key algorithm, algorithm strength, and the PKIX version you want to use. When you are using PKIX-CMP (specified as an integer, `5`, or as a static variable `PKIX5Version`) the values of the arguments for the signing algorithm and algorithm strength are taken from the user's policy certificate and any values that you set for these parameters are ignored.

3 Create a `User` object.

```
User user = new User();
```

4 Open a connection to the CA key management server (Manager) and establish a connection with the Directory, specifying IP addresses and port numbers for both entities.

```
ManagerTransport mt =
    new ManagerTransport(<IP address>, <port number>);
JNDIDirectory jndiDir =
    new JNDIDirectory(<IP address>, <port number>);
user.setConnections(jndiDir, mt);
```

5 Set the `CredentialWriter`.

```
FilenameProfileWriter profileWriter =
    new FilenameProfileWriter(<path to Profile>, null, 0);
user.setCredentialWriter(profileWriter);
```

6 Call the `login` method passing the `CredentialCreator` object and the user's password, obtained in step 1, as arguments.

```
user.login(credCreator, pwd);
```

Note

This step creates the user's credentials and stores them in the credentials file.

Remarks

The procedure for creating user credentials is essentially the same for every type of key

source, Entrust Profile, smart card, or credentials in PKCS #12 format. Ensure that you select the credential reader and credential writer appropriate for the key source you are working with.

To create a set of credentials, you need to use classes from the following Toolkit packages:

- `com.entrust.toolkit`
- `com.entrust.toolkit.credentials`
- `com.entrust.toolkit.util`
- `com.entrust.toolkit.x509.directory`

Identifying a user

Before users of your application can encrypt or decrypt messages, encrypt and sign data, or communicate securely over a network, they must identify themselves. A PKI, in turn, must be able to recognize users as trusted members of an organization. A user's credentials contain the information needed to authenticate the user to the PKI and to enable a user to perform cryptographic operations.

The process of logging in involves reading and verifying a user's credentials to ensure that the keys they contain are intact and valid. If any of the user's keys are out of date or need to be updated, the convenient time to make this determination and to perform the update operation is during the login process, so it is good practice to instantiate and set up a `CredentialWriter` for this purpose.

The following list is a summary of the procedure you should incorporate into your application to identify (authenticate), or log in, a user:

- Obtain the user's password and the location of his, or her, credentials file. The streams-based design of the Toolkit allows you to read credentials from sources such as files, smart cards, or over the Internet.
- Instantiate a `User` object to represent the user.
- Select a credential reader depending on the kind of credentials you want to read. The example code in this section deals with credentials stored as an Entrust Profile (`.epf` file) and uses the `FilenameProfileReader` as the chosen credential reader.
- Call the `User.login` method to complete the authentication procedure.

If you want your application to update a user's keys automatically at login time, you must connect to the CA key management server and to the Directory and instantiate a credential writer to write the new data.

Toolkit procedure — identifying a user

The following procedure illustrates how to authenticate a user:

1 Obtain the user's password as a `SecureStringBuffer`.

```
SecureStringBuffer password =
    new SecureStringBuffer(new StringBuffer(args[0]));
```

2 Determine the location of the credentials file.

```
String credentials = new String(args[0]);
```

3 Instantiate a user.

```
User user = new User();
```

4 Optionally, for automatic key update, connect to the Directory and to the CA key management server (or CA Registration Authority).

```
JNDIDirectory jndiDir =
    new JNDIDirectory(<IP address>, <port number>);
ManagerTransport man_trans =
    new ManagerTransport(<IP address>, <port number>);
user.setConnections(jndiDir, man_trans);
```

Note

You might not always need the connection to the CA key management server. If this is the case, you can set the `ManagerTransport` argument in the `User.setConnections` method to `null`. However, certificates are validated as soon as they are returned from the PKI, so if you use the `User.setConnections` method, you must have instantiated a `JNDIDirectory` object, and you must pass it as an argument in the method call. You cannot set the `JNDIDirectory` argument to `null`.

5 Instantiate a credential reader, in this case a `FilenameProfileReader`, passing the location of the Profile as an argument.

```
FilenameProfileReader credReader =
    new FileInputStream(<path to credentials file>);
```

Note

If you use `StreamProfileReader` and `StreamProfileWriter` to read from, and write to, a credentials file managed by instances of the `FileInputStream` and `FileOutputStream` classes, ensure that you specify different file names for the input and output streams. When you open a file with `FileOutputStream`, the file is always overwritten and when you try to read the same file using `FileInputStream`, the file will be empty. To prevent the destruction of the credentials file you are working with, ensure that you read from, and write to, files with different file names. If you use `StreamProfileReader` and `StreamProfileWriter`, use a `ByteArrayOutputStream` to write the credentials temporarily to memory as a precaution against the destruction of the credentials file.

The Toolkit classes called `FilenameProfileReader` and `FilenameProfileWriter` — used to read and write Entrust Profiles — prevent the possibility of overwriting or destroying credentials stored as files.

6 Instantiate a `FilenameCredentialWriter` object (in case key updates are required).

```
CredentialWriter credWriter =
    new FilenameProfileWriter(<.epf file - path and file name>, null, 0);
user.setCredentialWriter(credWriter);
```

Note

If you use `null` and zero as the second and third arguments for algorithm and hash count respectively, the constructor uses the default values for these arguments.

7 Call the `login` method to complete the user's authentication.

```
user.login(credReader, password);
```

Notes

The procedure for logging in, or authenticating, a user is essentially the same for every type of key source, Entrust Profile, smart card, or credentials in PKCS #12 format. Ensure that you select the credential reader and credential writer appropriate for the key source you are working with.

The `User` class has another constructor, `User(String profile, SecureStringBuffer password, String iniFile)`, that, given the following information, performs all the steps described in this procedure:

- The user's Entrust Profile.
- The user's password.
- The path to the user's `entrust.ini` file.

Disallowed login

Under certain conditions, the Toolkit can issue a number of warnings, but still allow user login to proceed. The Toolkit can also disallow user login altogether.

Some warnings can be considered informative, but others, once set, prevent `User` objects from invoking certain methods in the `User` class. If the Toolkit sets the warnings `WARNING_PW_EXPIRED` or `WARNING_PW_NOT_VALID`, use the `User.getClientSettings.getPasswordRuleTester` method to apply password rules appropriate for the user, and then the `User.changePassword` method to change the user's password.

Refer to the Javadoc reference documentation for the `User` class for more information about the warnings that the Toolkit can set during a user login operation.

Remarks

The procedure described in this subsection logs in a user using a simple command line interface to obtain user-supplied information. The user has a set of credentials (in this case an Entrust Profile stored as a file on the local computer), which is used to accomplish the log in task. To authenticate (or identify) and login a user, your application should use classes from the following Toolkit packages:

- `com.entrust.toolkit`
- `com.entrust.toolkit.credentials`
- `com.entrust.toolkit.util`
- `com.entrust.toolkit.x509.directory (optional)`
- `com.entrust.toolkit.exceptions`

Recovering a user's credentials

If a user loses, or in some way corrupts his or her credentials, your application should be able to perform a recovery operation. Recovering a user's credentials involves the generation of a new signing key pair and the secure retrieval, from the CA, of the user's current encryption public key certificate, decryption private key history, verification public key certificate, and CA verification public certificate.

Toolkit procedure — recovering credentials

From your application's point of view, the processes of recovering credentials and creating credentials are similar. The following procedure illustrates how to recover a user's credentials:

1 Request the following information from the user of your application:
- Authorization code (from the PKI Administrator)
- Reference number (from the PKI Administrator)
- Password

```
SecureStringBuffer pwd =
    new SecureStringBuffer(new StringBuffer(args[0]));
AuthorizationCode authCode =
    new AuthorizationCode(new StringBuffer(args[1]));
SecureStringBuffer refNo =
    new SecureStringBuffer(new StringBuffer(args[2]));
```

Note

Your application might use a GUI to prompt the user for this information — this example uses a simple command line interface.

2 Instantiate a `CredentialRecoverer` (a subclass of `CredentialReader`) object.

```
CredentialRecoverer credRecoverer =
    new CredentialRecoverer(refNo,
                            authCode,
                            CredentialRecoverer.DSASignature,
                            algorithm_strength,
                            CredentialRecoverer.PKIX5Version);
```

Note

Use the authorization code and reference number obtained in step 1. The `CredentialCreator` constructor also requires that you specify the signing key algorithm, algorithm strength, and the PKIX version you want to use. When you are using PKIX-CMP (specified as an integer, `5`, or as a static variable `PKIX5Version`) the values of the arguments for the signing algorithm and algorithm strength are taken from the user's policy certificate and the values that you set for these parameters are ignored.

3 Create a `User` object.

```
User user = new User();
```

4 Open a connection to the PKI CA key management server and establish a

connection with the Directory, specifying IP addresses and port numbers for both entities.

```
ManagerTransport mt =
    new ManagerTransport(<IP address>, <port number>);
JNDIDirectory jndiDir =
    new JNDIDirectory(<IP address>, <port number>);
user.setConnections(jndiDir, mt);
```

5 Set the `CredentialWriter`.

```
FileOutputStream epfFile = new FileOutputStream(<path to epf file>);
FilenameProfileWriter profileWriter = new FilenameProfileWriter(epfFile, null, 0);
user.setCredentialWriter(profileWriter);
```

6 Call the `login` method, passing the `CredentialRecoverer` object and the user's password, obtained in step 1, as arguments.

```
user.login(cred_recoverer, pwd);
```

Note

This step recovers the user's credentials and stores them in the credentials file.

Using PKCS #12 credentials

Using the Security Toolkit for Java, you can write applications that allow users to import and export private credentials that conform to the syntax specified in the Personal Information Exchange Syntax Standard, PKCS #12. This capability provides your application with increased interoperability among PKIs as the PKCS #12 portable storage format allows keys and certificates to be shared among different applications.

Exporting and importing credentials

The Toolkit's PKCS #12 export feature allows the export of a user's decryption private key and signing private key, but ignores the decryption private key history. If you export a user's private keys to PKCS #12 format and then import them into an Entrust Profile (as an `.epf` file), the decryption private key history is permanently lost. The options in an Entrust Profile, which are specific to an Entrust PKI, are not stored in the PKCS #12 file format so they too are lost in the `.epf`-to-PKCS #12 conversion. This means that you cannot use the PKCS #12 export feature of the Toolkit to convert back and forth between PKCS #12 files and Entrust Profiles without the loss of information.

The Toolkit supports only those credentials that have two key pairs — a pair for encryption and a pair for signing — and treats a set of credentials having only a single key pair as invalid. When the Toolkit imports PKCS #12 credentials with a single key pair, it checks for a certificate KeyUsage extension that determines how the key pair should be used. The Toolkit imports the credentials successfully if one of the following conditions apply:

- The KeyUsage extension allows the key pair to be used for both signing and encryption
- The KeyUsage is not critical

- There is no KeyUsage extension

If the KeyUsage extension allows the key pair to be used either for signing or for encryption, but not for both operations, the Toolkit cannot create valid credentials and throws an exception. When the Toolkit exports credentials to a PKCS #12 credentials file, it determines whether the credentials contain two identical key pairs. If the key pairs are identical, the Toolkit exports a single key pair, otherwise it exports two key pairs.

Note

To maintain security, you should import and export credentials as described in this section only if there is a legitimate need to do so. For interoperability, the Toolkit does not enforce password rules for PKCS #12 objects, such as credentials files, created or retrieved from third-party software (browsers, for example).

Toolkit procedure — exporting credentials

The following procedure illustrates how to export private keying material to a PKCS #12 file.

1 Instantiate a `User` object.

```
com.entrust.toolkit.User user = new User();
```

2 If working in online mode, connect to the CA key management server (Manager) and to the Directory.

```
com.entrust.toolkit.util.ManagerTransport mt =
    new ManagerTransport(<Manager>, <port number>);
com.entrust.toolkit.x509.directory.JNDIDirectory dir =
    new JNDIDirectory(<Directory>, <port number>);
user.setConnections(dir, mt);
```

Note

If you are working in offline mode, you must have access to an existing cache archive file.

3 Instantiate a `PKCS12Writer` object.

```
com.entrust.toolkit.credentials.FilenameProfileReader reader =
    new FilenameProfileReader(<epf file>);
com.entrust.toolkit.credentials.PKCS12Writer writer =
    new PKCS12Writer(new FileOutputStream(p12),
PKCS12Writer.SIGNING_AND_DECRYPTION_KEYS, 2000);
```

Note

The constructor, `PKCS12Writer(OutputStream outputStream, int keyExportMode, int hashCount)`, requires the following arguments:

- The output stream to which you want to write the PKCS #12 file
- A parameter specifying the keys you want to export — one of the following:
  - `PKCS12Writer.SIGNING_KEY`
  - `PKCS12Writer.DECRYPTION_KEY`

- `PKCS12Writer.SIGNING_AND_DECRYPTION_KEYS`

- The hashcount you want to use to protect the PKCS #12 file — should usually be 2000

4 Set the credential writer.

```
user.setCredentialWriter(writer);
```

5 Log in.

```
user.login(reader, new SecureStringBuffer(<password>));
```

6 Export the PKCS #12 file using the `user.write()` method.

```
user.write();
```

Note

To enable the export of a user's credentials to a PKCS #12 file, the user's policy certificates and public key certificates must contain the appropriate attributes and extensions. Before a PKI administrator can use Entrust/RA to set certificate types and to make changes to policy certificate settings, the `master.certspec` file must contain the appropriate entries. The following extract from a `master.certspec` file illustrates the entries that must be present.

- In the `[Certificate Types]` section, add

```
; ------------------------------------------------------------------
; PKCS-12 Exportable User
ent_p12export=enterprise,PKCS12 Export, PKCS12 Export Certificates
```

- In the `[Extension Definitions] [ent_p12export Common Extensions]` section, add

```
; ------------------------------------------------------------------
; P12 Exportable user.  Has both verification and encryption set.
;  If only encryption keys should be exportable, replace Common
;  Extensions with Encryption Extensions in the section name.
; ------------------------------------------------------------------
p12ex=2.16.840.1.114027.30.1,n,m,IA5String,"The keys corresponding to this
certificate are PKCS-12 Exportable."
```

- In the `[Attribute Definitions]` section, add

```
; ------------------------------------------------------------------
; PKCS-12 Export settings
; ------------------------------------------------------------------
p12excliset=1.2.840.113533.7.77.23,Boolean,<p12export_permitted>
p12minimum_hash=1.2.840.113533.7.77.24,Integer,<p12min_hash>
```

- In the `[Variables]` section, add

```
; PKCS-12 Export
p12export_permitted=Boolean,PKCS12 Export:,Allow users to export their
private key material to PKCS-12.,Range,0,1
p12min_hash=Integer,PKCS12 Min Hash:,Minimum allowed hash value for
PKCS #12 Export.,Range,1,10000
```

For details on how to use Entrust/RA to make changes to the `master.certspec`

file and how to set certificate types and policy certificate attributes, refer to the Entrust/PKI documentation **Administering Entrust/PKI 6.0 on Windows** or **Administering Entrust/PKI 6.0 on UNIX**.

The default value for the PKCS #12 minimum hash count is 2000. If you want to export credentials in PKCS #12 format to work with older browsers, change this value to 1. For maximum security, use a hash count of 10000.

Toolkit procedure — importing credentials

To import and use credentials that are stored as a PKCS #12 file, include the following steps in your application.

1 Create a `User` object.

```
com.entrust.toolkit.User user = new User();
```

2 Instantiate a `PKCS12Reader` object.

```
com.entrust.toolkit.credentials.PKCS12Reader reader =
    new PKCS12Reader(new FileInputStream(<path to p12 credentials file>));
```

3 If working in online mode, connect to the CA key management server (Manager) and Directory.

```
com.entrust.toolkit.utility.ManagerTransport transport =
    new ManagerTransport(manager, manPort);
com.entrust.toolkit.x509.directory.JNDIDirectory dir =
    new JNDIDirectory(directory, dirPort);
user.setConnections(dir, transport);
```

4 Log in the user using the PKCS #12 credential reader.

```
user.login(reader, new SecureStringBuffer(<password>));
```

Working with hardware tokens PKCS #11

The Toolkit provides support for the use of cryptographic operations with hardware tokens such as smart cards. This support is based on PKCS #11 version 2.01, the Cryptographic Token Interface Standard, also known as Cryptoki. For detailed information on PKCS #11, visit the PKCS #11 pages at the RSA Web site (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html). Cryptoki is an API that defines an interface between portable cryptographic devices, called cryptographic tokens, and software applications.

Configuring your computer environment

The steps you should take to configure your computer to work with cryptographic tokens depend on your computer's operating system. The following subsections describe how to enable support for PKCS #11 in a Microsoft® Windows® operating system, and in the AIX, HP-UX, and Solaris versions of the UNIX operating system.

Microsoft Windows operating systems

The JNIPKCS11.dll file is the native application extension, or dynamic link library (DLL), for use by applications built with the Toolkit when working with cryptographic tokens. The DLL is distributed in etjava61lib.zip, which you can download from the Entrust Customer Support Extranet by way of the Entrust Portal (https://www.entrust.com/index.htm). Unzipping this file puts JNIPKCS11.dll into the Toolkit's etjava\lib\win32 folder. Copy or move JNIPKCS11.dll into your computer's $windir\system32 folder (c:\winnt\system32, in a Microsoft® Windows NT® operating system, for example).

UNIX operating systems

The shared libraries for PKCS #11 support are distributed in etjava61lib.zip, which you can download from the Entrust Customer Support Extranet by way of the Entrust Portal (https://www.entrust.com/index.htm). Unzip this file and use the shared library that matches your operating system.

- AIX — use libJNIPKCS11.so in the Toolkit's etjava\lib\aix folder
- HP-UX — use libJNIPKCS11.sl in the Toolkit's etjava\lib\hpux folder
- Solaris — use libJNIPKCS11.so in the Toolkit's etjava\lib\solaris folder

To enable PKCS #11 support, put the appropriate shared library file for your operating system into the shared library search path on your computer — LD_LIBRARY_PATH on Solaris, SHLIB_PATH on HP-UX, or LIBPATH on AIX.

Connecting to the token's PKCS #11 library

The first step in working with tokens is to make a connection with the PKCS #11 library for the smart card you are using. The Toolkit class that enables you to make this connection is PKCS11LibraryConnection in the com.entrust.toolkit.pkcs11 package. Once connected to the library, you can use classes in the same package to obtain information about the library and about active tokens. The following list shows the important classes in com.entrust.toolkit.pkcs11.

- PKCS11LibraryConnection — contains methods that make, and test, the connection to the token's PKCS #11 library.
- PKCS11Information — contains methods that provide information about the token's PKCS #11 library, the Cryptoki interface, mechanisms, and slots .
- Info — represents a structure that provides general information about the Cryptographic Token Interface Standard.
- TokenInfo — contains methods that provide general information about the active token.
- SlotList — contains methods that provide information about the number of active slots and slot IDs.
- SlotInfo — represents a structure that provides information about a slot.
- MechanismList — contains methods that provide information about the cryptographic operations, or mechanisms, supported by the token.
- MechanismInfo — represents a structure that provides information about a

particular mechanism supported by a token.

Public certificates on tokens

The Toolkit supports a policy setting that controls where public key certificates are stored on a hardware token. This subsection describes the entries in the master certspec file that allow the policy certificate client attribute to be set, using Entrust/RA, allowing public certificates to be stored in the token's public memory. If a user's policy certificate does not include this policy setting, or if it is set to `false`, the Toolkit stores public certificates in the token's private memory.

Use Entrust/RA to export the master certspec file as a text file for editing and complete the following steps:

1 In the `[policy_clisetAttributes]` section, add

```
public_token_certs=1.2.840.113533.7.77.49,Boolean,<public_token_certs>
```

2 In the `[Variables]` section under `Base Settings Variables`, add

```
public_token_certs=Boolean,Public token certificates:,
Certificates will be stored in public memory on the token.,Range,0,1
```

3 Save the changes and import them back into Entrust/RA.

4 Open `entmgr.ini` for editing.

5 In the `[Default Variables Values]` section, add

```
public_token_certs=0
```

Note

This step gives the `<public_token_certs>` variable, defined in step 1, a default value.

If the `[Default Variables Values]` section does not exist, add it to the end of the `entmgr.ini` file.

6 Save the changes to `entmgr.ini`.

Use Entrust/RA to create a **User Policy** with the **Public token certificates** value checked, and create a **User Role** for this new policy. Users you create with this new role can now store their public key certificates in the public memory on a hardware token.

Note

For details on how to use Entrust/RA to make changes to the `master.certspec` file and how to set certificate types and policy certificate attributes, refer to the Entrust/PKI documentation **Administering Entrust/PKI 6.0 on Windows** or **Administering Entrust/PKI 6.0 on UNIX**.

Working with credentials

The Toolkit treats smart cards as a source and repository for credentials. To read a set of credentials from a smart card, your application should instantiate a `CredentialReader` object (a `TokenReader`). When creating credentials, or performing automatic key

rollover, your application should instantiate a specific type of `CredentialWriter` object (a `TokenWriter`). The `TokenWriter` class writes credentials to a hardware token that have been created, read, or recovered by one of three kinds of credential reader:

- `TokenReader`
- `TokenCredentialCreator`
- `TokenCredentialRecoverer`

Auxiliary profiles

Writing credentials to a token, using a `TokenWriter`, creates an auxiliary profile to store a list of decryption private keys — a decryption private key history. Every time the encryption public keys are updated, the decryption private key history is stored on the token, as long as it has sufficient memory, and in the auxiliary profile. If one of the decryption private keys cannot be stored on the token, it is deleted to allow room for the new key. The decryption key history in the auxiliary profile, however, is available to decrypt old data.

Note

The `TokenWriter` has a method, `createAuxProfile(boolean creatAuxProfile)`, that allows you to prevent the creation of an auxiliary profile by calling the method with its argument set to `false`. Without an auxiliary profile, it is possible that you might lose superceded decryption private keys, preventing you from retrieving data that had been encrypted using those keys. If this happens, you will have to perform a key recovery operation before you can gain access to your encrypted data.

Refer to the Javadoc reference documentation for more details.

Toolkit procedure — creating credentials on a token

The steps in this procedure illustrate how to create credentials on a token using the `TokenCredentialCreator` class.

1 Create a connection to the token's PKCS #11 library.

```
PKCS11LibraryConnection pkcs11LC =
    new PKCS11LibraryConnection("tokenlib.dll");
```

2 Specify a password (the user PIN) to log in to the token.

```
SecureStringBuffer pwd = new SecureStringBuffer("userInitPwd");
```

3 Specify the authorization code and reference number.

```
AuthorizationCode authCode =
    new AuthorizationCode(new StringBuffer(<authorization code>));
SecureStringBuffer refNumber =
    new SecureStringBuffer(new StringBuffer(<reference number>));
```

4 Create a `PKCS11InformationObject` and retrieve information about the available slot IDs.

```
PKCS11Information pkcs11Information = new PKCS11Information(pkcs11LC);
SlotList slotList = pkcs11Information.getSlotList(true);
long[] slotListIDs = slotList.getSlotListIDs();
```

5 Specify a label for the token.

```
String label = "Token Test";
```

6 Specify the security officer's PIN.

```
SecureStringBuffer soPin = new SecureStringBuffer(new StringBuffer(<SO's PIN>));
```

7 Instantiate a `TokenCredentialCreator`.

```
TokenCredentialCreator tcc =
    new TokenCredentialCreator(refNumber,
                               authCode,
                               TokenCredentialCreator.RSASignature,
                               1024,
                               TokenCredentialCreator.PKIX5Version,
                               slotListIDs[0],
                               pkcs11LC,
                               soPin,
                               label);
```

8 Create a User and set connections to the CA key management server and to the Directory.

```
User user = new User();
ManagerTransport emt =
    new ManagerTransport(<ip address>, <port number>);
JNDIDirectory dir = new JNDIDirectory(<ip address>, <port number>);
user.setConnections(dir, emt);
```

9 Specify a label for the token, the path to the auxiliary profile, and a name for the auxiliary profile.

```
String label = "Token Test";
String entrustPath = <path to profile>;
String auxProfName = <auxiliary profile name>;
```

Note

Always specify the auxiliary profile name, which must be the Entrust user name, even if you do not want to create an auxiliary profile. Refer to the section entitled **Working with credentials** for details about auxiliary profiles.

10 Instantiate and set a `TokenWriter` as the `CredentialWriter`.

```
TokenWriter tw = new TokenWriter(entrustPath,
                                 auxProfName,
                                 null,
                                 0);
user.setCredentialWriter(tw);
```

11 Log in the user to create the new set of credentials.

```
user.login(tcc, pwd);
```

12 Log out and close the connection to the token library.

```
user.logout();
pkcs11LC.closeConnection();
```

Toolkit procedure — cryptographic operations with a token

The steps in this procedure show how to log into a token and how to perform cryptographic operations such as encryption, decryption, signing, and verification.

1 Perform steps 1 to 4 of the procedure for **writing an Entrust Profile to a token**.

Note

The password you specify in step 2 is the password used to log in to the token.

2 Create a `TokenReader` and a `TokenWriter`.

```
TokenReader tokenReader = new TokenReader(pkcs11LC, slotListIDs[0]);
TokenWriter tokenWriter = new TokenWriter(null, null, null, 0);
```

Note

The `TokenWriter`'s arguments are set to `null` and zero because there is to be no change to the auxiliary profile.

If you make changes to the auxiliary profile, the default value for the hash count used for its protection is 2. Use the default value to maintain compatibility with Entrust Authority Toolkits and applications that use the C and C++ runtime components.

3 Create a `User`, set the `CredentialWriter`, and log into the token.

```
User user = new User();
user.setCredentialWriter(tokenWriter);
user.login(tokenReader, <password>);
```

4 Perform encryption and decryption operations.

```
Cipher cipher = Cipher.getInstance("RSA/2/PKCS1Padding");
String data = "Test string.";

cipher.init(Cipher.ENCRYPT_MODE, user.getEncryptionCertificate().getPublicKey());
byte[] encryptedData = cipher.doFinal(data.getBytes());

cipher.init(Cipher.DECRYPT_MODE, user.getDecryptionKey());
String decryptedData = new String(cipher.doFinal(encryptedData));
System.out.println("Decrypted data: " + decryptedData);
```

5 Perform signing and verification operations.

```
Signature signer = Signature.getInstance("SHA-1/RSA");

signer.initSign(user.getSigningKey());
signer.update(data.getBytes());
byte[] signature = signer.sign();

X509Certificate verCert = user.getVerificationCertificate();
signer.initVerify(verCert.getPublicKey());
signer.update(data.getBytes());
boolean test = signer.verify(signature);
if(test) {
    System.out.println("Verification successful!");
}
```

6 Log out and close the connection to the token library.

```
user.logout();
pkcs11LC.closeConnection();
```

Using the Entrust key store implementation

The Security Toolkit for Java implements the concept of a key store — a container that holds cryptographic keys and certificates. Key stores are key management tools. A single key store contains all the information required by a particular entity (a user or an application) for authentication.

A key store can contain two types of entries:

- Multiple key entries — corresponding to private keys and public key certificates, or certificate chains, that authenticate the matching public keys
- Multiple trusted certificate entries — corresponding to certificates holding the public keys of other entities

Key store entries are identified using unique aliases that associate the entries with their functions (with email, or digital signing, for example), or with their particular entities (Bob, or Alice).

Note

You can use PKCS #12 files as read/write and read only certificate stores by recording the paths to such files in an Entrust Keystore initialization file (`*.kst`) as described in the subsection entitled **KeyStore ini file**. These entries must be associated with an alias, so they cannot be PKCS #12 files directly exported from a set of Entrust user credentials (`*.epf`).

EntrustKeyStoreSpi class

The Toolkit's implementation of key stores provides, through the `com.entrust.toolkit.credentials.EntrustKeyStoreSPi` class, a common interface for the user to work with a variety of credentials.

Key store ini file

The information required for a user to log in using a key store is stored in a key store initialization file. You should provide the key store initialization file as an `InputStream` argument to the `EntrustKeyStore.engineLoad(InputStream is, char[] password)` or `KeyStore.load(InputStream stream, char[] password)` methods when you initialize the key store. Although you can name the key store initialization file anything you like, to avoid confusion the file should have a `.kst` file name extension and a name that associates it with its user. For example, if the key store initialization file is associated with a user whose Entrust profile is `jsmith.epf`, call the key store initialization file `jsmith.kst`.

Entries in the key store initialization file are in the form of key-value pairs grouped into sections containing related entries.

```
[Section name]
key1 = value1
key2 = value2
key3 = value3
```

The following example illustrates the contents of a key store initialization file associated with the Entrust profile, `jsmith.epf`.

```
[Password Token]
Magic Number=1789
Version=6.0
SaltValue=BiuTBuMs91U=
Token=841B5E736C8250B3
Protection=128
MAC Algorithm=pbeWithSHA1AndCAST5-CBC
HashCount=1000

[Credential Store]
CRType=EntrustProfile
CRPath=jsmith.epf

[Certificate Store]
CSP12WriteCount=1
CSP12Write1=ReadWriteCertStore.p12
CSP12WritePwd1=qxrS+F1jvmwD4IGe2vaHlixrGIsg+9zq/twML42oizMCxyowIQxQ9oUvm

[Ldap]
LdapServer=10.96.8.116+389

[Manager]
Manager=10.96.8.116+829

[Integrity]
MAC=57DD2B4E5FD5A45A=79+BOBr
```

The entries in the key store initialization file, some of which are mandatory and some optional, are grouped into six sections. The following table shows the six sections, their entries, and whether or not an entry is mandatory.

Entries in the key store initialization file

Section name	Entry	Value	Mandatory or optional
[Password Token]	SaltValue	Random value created when the key store ini file is created	Mandatory
	Token	Value used to check the password	Mandatory
	Protection	Length of the symmetric protection key — default is 128-bit	Mandatory
	MAC algorithm	Algorithm used to protect the key store ini file	Mandatory

	HashCount	Number of iterations of the hash function used to create the symmetric key	Mandatory
	Magic Number	1789	Mandatory
	Version	60	Mandatory
[Credential Store]	CRType	One of: **EntrustProfile**, **PKCS12**, or **PKCS11**	Mandatory
	CRPath	Path to an Entrust profile (`.epf` file) or to a set of PKCS #12 credentials (`*.p12` file)	Mandatory when CRType is EntrustProfile or PKCS12
[Certificate Store]	CSP12WriteCount	Number of read/write certificate stores	Optional
	CSPWrite#	Path to PKCS #12 file used as read/write certificate store, where # indicates the number of the certificate store — 1,2, ... , n	Optional
	&CSP12WritePwd#	Protected password to the certificate store, where # indicates the number of the certificate store — 1,2, ... , n	Optional
	CSP12ReadCount	Number of read-only certificate stores	Optional
	CSP12Read#	Path to the PKCS #12 file used as a read-only certificate store, where # indicates the number of the certificate store — 1,2, ... , n	Optional
	&CSP12ReadPwd#	Protected password to the certificate store	Optional
[Ldap]	LdapServer	IP address of the LDAP directory + port number For example, `10.96.8.116+389`	Optional
[Manager]	Authority	IP address of the Manager	Optional

		+ port number For example, `10.96.8.116+829`	
[Integrity]	MAC	Message authentication code (MAC) value	Mandatory

Password Token section

The `Password Token` section of the key store initialization file contains information associated with the protection of certain entries — entries whose names begin with an ampersand (`&`). The values associated with these entries are the results of cryptographic hash functions whose inputs were the original, plain text values to which the salt value had been appended. Other files, such as an Entrust Profile (`.epf` file) or a PKCS #12 credentials file (`.p12` file), also use the initialization file format and have a Password Token section. The values associated with the Magic Number (`1789`) and Version uniquely identify the `.kst` file as a key store initialization file.

Credential Store section

The `Credential Store` section of the key store initialization file contains the information required to read a set of credentials. Each key store initialization file contains one `CRType` entry that identifies the type of credential store associated with the key store. The `CRType` must be one of the following:

- `CRType=EntrustProfile` — identifies an Entrust Profile (`.epf` file)
- `CRType=PKCS12` — identifies a PKCS #12 credentials file (`.p12` file)
- `CRType=PKCS11` — identifies credentials stored on a hardware token

When the credential store is an Entrust Profile or a PKCS #12 file, the value associated with the `CRPath` entry is the path to the `.epf` or `.p12` file. When the credential store is a set of credentials on a hardware token, the PKCS #11 library and hardware slot number are the key store initialization file values associated with the `CRLibrary` and `CRSlotNr` entries respectively.

Certificate Store section

Although a key store initialization file contains just one credential store entry, it can contain several certificate store entries that point to PKCS #12 files. Certificate stores are numbered consequtively beginning at 1, and the entries in the key store initialization file define which certificate stores are read-only and which are read/write.

Ldap section

The `LdapServer` entry in the key store initialization file specifies the IP address and port number of a server that has a directory conforming to the lightweight directory access protocol (LDAP).

Manager section

The `Authority` entry in the key store initialization file specifies an IP address and port number of a server designated as the PKI's CA key management server.

Integrity section

The `MAC` entry in the key store initialization file holds the value of the message authentication code (MAC) calculated over the entire key store initialization file.

High-level key store classes

The Toolkit provides high-level classes for working with key stores. The classes belong to two packages distributed in the `keystore.jar` file.

com.entrust.toolkit.keystore

This package contains the `CertStore` class — used to read and write public key certificates, and the `KSIniFileCreator` class — used to create key store initialization files.

com.entrust.toolkit.credentials

This package contains the `EntrustKeyStoreSpi` class — which provides the service provider interface for the Toolkit's key store implementation.

Toolkit procedure — creating a key store initialization file

The following procedure illustrates how you can use the Toolkit to create a key store initialization file.

1 Instantiate a `User` and log in.

Note

Refer to the procedure entitled **Identifying a user** in the **User and credentials management section**.

2 Create an instance of the `com.entrust.toolkit.keystore.KSIniFileCreator` class.

```
KSIniFileCreator ksIniFileCreator = new KSIniFileCreator(<path to credentials file>);
```

Note

The `KSIniFileCreator` has three constructors:

- `KSIniFileCreator(java.lang.String path)` — where the `path` argument is the full path to either an Entrust Profile (`.epf` file) or to a set of PKCS #12 credentials (`.p12` file)
- `KSIniFileCreator(java.lang.String pkcs11Library, int slotNr)` — used to create a key store initialization file for a PKCS #11 credential store
- `KSIniFileCreator(java.lang.String ksIniFile,`

`SecureStringBuffer password)` — used to create a key store initialization file by passing an existing key store initialization file and its password as arguments

3 Add any number of read/write certificate stores to the key store initialization file.

```
ksIniFileCreator.addWriteCertificateStore(<path to the .p12 file cert store>, <cert
store password>);
```

Note

If you are adding a certificate store for the logged in user (step 1 of this procedure), specify the password used for logging in. The certificate store password argument is sensitive data and must be an instance of the `SecureStringBuffer` class.

You can add certificate stores to the key store initialization file only by using the `KSIniFileCreator` class. The `KSIniFileCreator` class ensures that the passwords associated with the certificate stores stored in the key store initialization file are properly protected. For this reason do not write passwords manually to a key store initialization file.

4 Establish a connection to a Certification Authority (CA) and to a certificate repository.

```
ksIniFileCreator.setLdap(<IP address of certificate repository>, <port number of
certificate repository>);
ksIniFileCreator.setAuthority(<IP address of certification authority>, <port number of
certification authority>);
```

Note

This step is necessary if you have to retrieve public key certificates from the Directory. The sample application in the `etjava\examples\keystore` folder includes these steps, but, to keep the sample uncomplicated, does not establish the connection.

5 Create the new key store initialization file.

```
ksIniFileCreator.store(new FileOutputStream(<path to key store initialization file
(.kst)>), <ini file password>);
```

Note

To avoid confusion between key store initialization files and other configuration files used by the Toolkit, give the key store initialization file a `.kst` file name extension and a file name that associates the key store with its user.

Use the password of the credential store specified in the key store initialization file as the second argument in this method.

Toolkit procedure — logging in to a key store

The following procedure illustrates how to log in to a key store, how to retrieve

encryption and signing keys, and how to read and write public key certificates.

1 Instantiate a `User` and log in.

Note

Refer to the procedure entitled **Identifying a user** in the **User and credentials management section**.

2 Create an instance of the `java.security.KeyStore` class.

```
KeyStore keyStore = KeyStore.getInstance("Entrust");
```

Note

The `getInstance(String type)` method returns a `KeyStore` instance of type **Entrust**. If you have not installed the Entrust Cryptographic Service Provider (CSP), the method will throw a `KeyStoreException` indicating that the Entrust key store type is not available.

3 Load a key store.

```
keyStore.load(new FileInputStream(<path to key store ini file>),
password.getStringBuffer().toString().toCharArray());
```

Note

The `password` object is an instance of the `SecureStringBuffer` class.

4 Retrieve the encryption public key and encryption public certificate from the key store.

```
Key encryptionKey= keyStore.getKey("encryption", null);

Certificate[] encryptionCertChain = keyStore.getCertificateChain("encryption");
Certificate encryptionCert = keyStore.getCertificate("encryption");
```

Note

The alias, **encryption**, specifies that the `getKey` and `getCertificate` methods should retrieve the encryption public key and encryption public certificate respectively.

5 Retrieve the signing private key and verification public certificate from the key store.

```
Key signingKey = keyStore.getKey("signing", null);

Certificate[] signingCertChain = keyStore.getCertificateChain("signing");
Certificate signingCert = keyStore.getCertificate("signing");
```

Note

The alias, **signing**, specifies that the `getKey` and `getCertificate` methods should retrieve the signing private key and verification public certificate respectively.

6 Retrieve certificates from the certificate repository.

```
Certificate c1 = keyStore.getCertificate(<DN>);
Certificate c3 = keyStore.getCertificate(<DN>);
Certificate c6 = keyStore.getCertificate(<DN>);
Certificate c7 = keyStore.getCertificate(<DN>);
```

Note

If you have established a connection to a certificate repository, you can retrieve certificates using the `getCertificate` method specifying the distinguished name (DN) of the entity to whom the certificate belongs.

7 Write the encryption public certificate and the verification public certificates to a read/write certificate store.

```
keyStore.setCertificateEntry("encCert", encryptionCert);
keyStore.setCertificateEntry("sigCert", signingCert);
```

Note

The `setCertificateEntry(String alias, Certificate cert)` method assigns the given alias to the `Certificate` object.

8 Verify that the certificates were written to the certificate store.

```
if(keyStore.getCertificate("sigCert") != null &&
    keyStore.getCertificate("encCert") != null)
{
    System.out.println("All certificates could be read from the read/write certificate
store.");
}
else
{
    System.out.println("All certificates could not be read from the read/write
certificate store.");
}
```

9 Write the key store.

```
keyStore.store(null, null);
```

Note

The `store(OutputStream stream, char[] password)` method writes the key store to an `OutputStream` and protects it with a password. If both arguments are `null`, the method uses the path and password that were specified when the key store was loaded.

Working with key stores in memory

An Entrust key store contains a credential store and a certificate store. The Toolkit allows key stores to be held in memory for those situations where you would prefer not to save credentials and certificates in a file. Using generic stream input and output (I/O) operations, you can gain access to key stores in memory to retrieve, or to add, keys and certificates. Certificate stores that are held in memory are read/write stores.

Note

You should not combine the handling of key stores in memory with the handling of file-based key stores described in the previous subsections of this chapter.

Loading and storing key store information using generic I/O streams

The following code fragments show how to load and retrieve information from an

Entrust key store in memory. The Toolkit's `etjava\examples\keystoreInMemory` folder contains a sample application that demonstrates these tasks.

Create a key store and load credentials from a data structure in memory

A data structure in memory might be an instance of a class that is unique to your own application. The following code fragment uses a simple `ByteArrayInputStream`.

```
java.security.KeyStore.KeyStore keyStore = KeyStore.getInstance("Entrust");
keyStore.load(new ByteArrayInputStream(byteArray),
password.getStringBuffer().toString().toCharArray());
```

where `byteArray` represents a set of credentials in a byte array (`byte[]`).

Note

The `java.security.KeyStore.KeyStore.getInstance(String type)` method returns a `KeyStore` instance of type **Entrust**. If you have not installed the Entrust Cryptographic Service Provider (CSP), the method will throw a `KeyStoreException` indicating that the Entrust key store type is not available.

Add a trusted certificate to the key store

```
keyStore.setCertificate("Bob", cert);
```

where `Bob` is the alias of the certificate and `cert` is an instance of the `java.security.cert.Certificate` class containing a trusted certificate.

Retrieve certificates from the key store using their aliases

```
// CA certificate
X509Certificate caCert = (X509Certificate)keyStore.getCertificate("CA");

// Encryption certificate
X509Certificate encCert = (X509Certificate)keyStore.getCertificate("encryption");

// Signing certificate
X509Certificate signCert = (X509Certificate)keyStore.getCertificate("signing");

// Bob's certificate
X509Certificate signCert = (X509Certificate)keyStore.getCertificate("Bob");
```

Write the key store to another ByteArrayOutputStream

```
ByteArrayOutputStream anotherBaos = new ByteArrayOutputStream();
keyStore.store(anotherBaos, password.getStringBuffer().toString().toCharArray());
```

Using Single Login

Single Login is a feature of the Entrust Login Interface (ELI). Single Login allows users to log in to more than one application managed by Entrust/Entelligence and running in Microsoft Windows operating systems, without having to identify, or authenticate, themselves every time they want to use an application. ELI centralizes control of

inactivity time outs across the Entrust Authority Toolkit applications that are running on a computer. ELI is part of the runtime components, which provide the underlying cryptographic capabilities that the Entrust/PKI and Entrust client applications are based upon. To interact with the runtime components, the Toolkit uses the Java Native Interface (JNI).

The native application extension, or dynamic link library (DLL), for use with Single Login is `JNIELI.dll`. It is distributed in `etjava61lib.zip`, which you can download from the Entrust Customer Support Extranet by way of the Entrust Portal (https://www.entrust.com/index.htm). Unzipping this file puts `JNIELI.dll` into the Toolkit's `etjava\lib\win32` folder. Copy or move `JNIELI.dll` to the appropriate location on your computer — the current directory of your application, or a location identified by the `PATH` environment variable. Ensure that the PATH environment variable does not specify any other versions of `JNIELI.dll`.

Note

Users of your application should be aware that if their policy certificates have the attribute set that disables Single Login, the Toolkit will throw an exception if they try to use this feature. The `Disable single login` attribute is not set by default, so, unless the policy settings of users' certificates have been edited to set the attribute, users will be able to use the Single Login feature.

Entrust recommends that FipsMode is turned off (`FipsMode=0`) in the `[FIPS mode]` section of your `entrust.ini` file when you are using ELI with the Security Toolkit for Java. If FipsMode is turned on (`FipsMode=1`) and if your `entrust.ini` file does not contain an entry that specifies a string containing a MAC of an authorized application (for example, `myappnameAuth=DES-MAC,64,C7EAA250F7EA8013:CAST3-MAC,64,64,C4285C5`), the Toolkit will generate a `com.entrsut.toolkit.exceptions.UserFatalException` exception when you try to log into the Toolkit.

When you use a `SingleLoginReader`, you cannot set a `CredentialWriter`. Key updates, DN changes, and CA key updates are handled by the runtime components. Password changes, however, are still possible using the Toolkit, as they are passed on to the runtime components internally.

Toolkit procedure — single login

The following procedure illustrates how you can use the Toolkit to incorporate support for Single Login in your applications.

1 Instantiate a `SingleLoginReader` object.

```
SingleLoginReader singleLoginReader =
    new SingleLoginReader(applicationPath,
                          entrustIniFile,
                          etHardwareProfile);
```

Note

The `SingleLoginReader` class in the `com.entrust.toolkit.credentials` package is a subclass of `CredentialReader`. The `SingleLoginReader` constructor has three `String` arguments:

- `applicationPath` — a `String` argument that specifies the path and fully qualified name of the class in your application that contains the main method, or the path to the jar file that contains the class with the main method.
- `entrustIniFile` — a `String` argument that specifies the fully qualified path to the `entrust.ini` file. Without this, login using ELI is not possible.

  Note

  If you are using ELI with a cryptographic hardware token, the location of the PKCS #11 library (`JNIPKCS11.dll`) is read from either the `CryptokiV2LibraryNT` (on Microsoft Windows NT) or `CryptokiV2Library95` (on Microsoft® Windows® 95) entries in the `[Entrust Settings]` section of the `entrust.ini` file. The `entrust.ini` must be the same file used by Entrust/Entelligence (located by default in the `Windows` or `WINNT` directory on Microsoft Windows operating systems) otherwise ELI might not be able to locate token credentials properly. After your application has finished using ELI, close the PKCS #11 library before the application terminates using the following method in the `com.entrust.toolkit.credentials.SingleLoginReader` class:

  ```
  SingleLoginReader.closePKCS11LibraryConnection();
  ```

- `etHardwareProfile` — a `String` argument that specifies the fully qualified path to the `ethardware.ini` file.

  Note

  When you use this argument, Entrust Profiles on hardware tokens are searched using the PKCS #11 libraries (those libraries that end with `V2LibraryNT` or `V2Library95` on Windows NT and Windows 95 operating systems respectively) specified in the `[CryptokiLibraries]` section of the `ethardware.ini` file. If you set this argument to `null`, the path to the PKCS #11 shared library is taken from the `entrust.ini` file.

2 Create a `User` object and log in.

```
User user = new User();
user.login(singleLoginReader, null);
```

Note

No password is required in the `login` method when using a `SingleLoginReader` as the credential reader.

3 Prevent ELI from logging out the user when ELI times out.

```
user.blockELILogout();
```

Note

Use the `com.entrust.toolkit.User.blockELILogout()` method when

you do not want the user to be logged out at the same time as ELI logs out. This method is useful if you want your application to perform Toolkit operations unattended for a period longer than the ELI time out setting. The user remains logged in after ELI logs out until the `user.unblockELILogout()` is called.

4 Perform other Toolkit operations.

5 Remove the ELI log out restriction.

```
user.unblockELILogout();
```

6 Log out the user.

```
user.logout();
```

Note

The user can log out before ELI has timed out.

Remarks

The high-level API classes, `SingleLoginReader` and `TokenReader`, implement classes that are not part of the public API of the Toolkit. These classes in turn implement a method (`addLogoutListener(LogoutListener logoutListener)`) to add the `com.entrust.toolkit.credentials.ELILogoutListener` to the `User` instance associated with the high-level classes. When the `User` instance logs out, it calls the `logout` methods of all the `LogoutListener` objects (such as, `ELILogoutListener.logout()`) that have been added. The `ELILogoutListener.logout()` method informs the Single Login classes that the Toolkit has logged out. This process is automatic when you work directly with the `SingleLoginReader` and `TokenReader` classes.

Using the Server Login feature

Server Login addresses the requirement for an application to obtain access to Entrust credentials without the need for manual authentication. Server Login is designed for computers, usually servers, that run Entrust applications as services or as background applications. These computers, running 24 hours a day, seven days a week, do not have a user continuously present and are often in a physically secure area that has restricted access. Using Server Login, Entrust-Ready services or background processes can start without operator intervention.

Configuring your computer environment

The steps you should take to configure your computer to work with the Server Login feature depend on your computer's operating system. The following subsections describe how to enable Server Login support in Microsoft Windows and the AIX, HP-UX, and Solaris versions of the UNIX operating system.

Microsoft Windows operating systems

The `UALJNI.dll` file is the native application extension, or dynamic link library (DLL),

for use by applications built with the Toolkit when working with the Server Login feature. The DLL is distributed in `etjava61lib.zip`, which you can download from the Entrust Customer Support Extranet by way of the Entrust Portal (https://www.entrust.com/index.htm). Unzipping this file puts `UALJNI.dll` into the Toolkit's `etjava\lib\win32` folder. Copy or move `UALJNI.dll` into your computer's `$windir\system32` folder (`c:\winnt\system32`, in a Windows NT operating system, for example).

UNIX operating systems

The shared libraries for the Server Login feature are distributed in `etjava61lib.zip`, which you can download from the Entrust Customer Support Extranet by way of the Entrust Portal (https://www.entrust.com/index.htm). Unzip this file and use the shared library that matches your operating system.

- AIX — use `libualjni.so` in the Toolkit's `etjava\lib\aix` folder
- HP-UX — use `libualjni.sl` in the Toolkit's `etjava\lib\hpux` folder
- Solaris — use `libualjni.so` in the Toolkit's `etjava\lib\solaris` folder

To enable support for the Server Login feature, put the appropriate shared library file for your operating system into the shared library search path on your computer — `LD_LIBRARY_PATH` on Solaris, `SHLIB_PATH` on HP-UX, or `LIBPATH` on AIX.

Allowing Server Login

If you want your application to use the Server Login support that the Toolkit provides, the application must have access to a bound credentials file. The `com.entrust.toolkit.credentials.UALCreator` class implements a Server Login binder that binds a password to a set of machine information to create a credentials (`.ual`) file. To accomplish this, the class uses an Entrust initialization file (`entrust.ini`), an Entrust Profile (a `.epf` credentials file), and the Toolkit's native application extension or shared library for the operating system you are using.

You can also create a Server Login credentials file using the binder application that is part of the Entrust Authority Toolkits Server Login feature. When you run the binder application, the Entrust profile and the computer are cryptographically bound together to create the `.ual` credentials file. The binder application uses the Entrust user's password and Profile (`.epf` file) to log into Entrust. The application then verifies that the user's status is valid and checks the user's policy certificate to determine whether the administrative capabilities for the role of the user allow the Server Login feature to be used. If the user's policy certificate specifies that the Server Login feature can be used, the binder application creates a `.ual` file, otherwise the application displays an error message stating that the user is not authorized to use Server Login, and does not create a `.ual` file.

The `ServerLoginExample.java` sample application in the `etjava\examples` folder demonstrates how to create a set of credentials (`.ual` file) that binds a user's Entrust Profile to a computer, and how to perform a login operation using the newly created `.ual` file. The sample application also shows how to instantiate credential writers and readers to provide automatic key updates of the `.ual` file if they are

required.

Note

For more details on the Server Login feature, download the feature from the Entrust Customer Support Extranet by way of the Entrust Portal (https://www.entrust.com/index.htm), and refer to the release notes.

Toolkit procedure — logging in using .ual credentials

The following procedure illustrates how to log in using a `.ual` credentials file.

1 Create a `CredentialReader` depending on the kind of credentials you want your application to read.

```
FileInputStream fis = new FileInputStream(<path to .epf file>);
com.entrust.toolkit.credentials.CredentialReader credReader =
    new FilenameProfileReader(fis);
```

Note

The code fragment above instantiates a `FilenameProfileReader` to read an Entrust Profile.

2 Create a `User`.

```
com.entrust.toolkit.User user = new User();
```

3 Instantiate the `UALCredentialReader`.

```
com.entrust.toolkit.credentials.UALCredentialReader reader =
    new UALCredentialReader(credReader, new FileInputStream(<path to .ual file>));
```

4 Log in the user.

```
user.login(reader, null);
```

Note

No password is needed for the `login` method in this case — use `null` as the second argument.

Remarks

The Toolkit includes a credential reader, `UALCredentialReader`, that you must use to log in a user and read the password from a `.ual` credentials file. The `UALCredentialReader` relies on native code to determine the machine properties needed to read the `.ual` file. As a result, it requires that the Server Login DLL or shared library (`ualjni.dll` in a Microsoft Windows operating system, or `libualjni` in a UNIX operating system) be installed on the computer in the appropriate path — the current directory of your application, a location identified by the PATH environment variable in a Microsoft Windows operating system, or in the shared library search path on a computer running UNIX.

Using the Toolkit in FIPS mode

Release 6.1 of the Security Toolkit for Java has an operating mode designed to conform to the Federal Information Processing Standards (FIPS) PUB 140-2, May 2001 (http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf). The FIPS series of standards is published by the United States National Institute of Standards and Technology (NIST). FIPS PUB 140-2 is the standard that describes a set of requirements to which cryptographic software, used to secure unclassified data in computer and telecommunications systems, must conform. When release 6.1 of the Toolkit is operating in FIPS mode, it is operating in a mode designed to conform with security level 1 of the FIPS PUB 140-2 standard.

Toolkit preparation

FIPS requires that the source of cryptographic algorithms, the relevant Toolkit jar files, be authenticated using a message authentication code (MAC). To make this task easier, ensure that the following jar files are located in the same folder. These jar files comprise the cryptographic module of the Toolkit containing classes that implement cryptographic logic:

- `entbase.jar`
- `entp5.jar`
- `entp11.jar`
- `entp12.jar`
- `entroaming.jar`
- `entssl.jar`

Note

If you unpacked the Toolkit zip file into the default folder, `etjava`, all Toolkit jar files are located in the `etjava\lib` folder.

If you have not installed the Toolkit as an installed optional package (installed extension), ensure that the jar files listed here are specified in your classpath when you are using the Toolkit in FIPS mode.

Toolkit initialization

Follow the procedure entitled **Identifying a user** (earlier in this chapter) to log in to your Toolkit-based application. To initialize Toolkit in FIPS mode, call the `initialize` method in the `com.entrust.toolkit.security.fips.SecurityEngine` package:

```
com.entrust.toolkit.security.fips.SecurityEngine.initilaze(true);
```

Initialization registers the Toolkit's cryptographic providers in the correct order, which ensures that calls to the JCA use the certified FIPS algorithms. The `initialize` method registers providers in the following order:

1 SUNJCE — the cryptographic provider for the JCE

2 Entrust — the Entrust cryptographic provider (including FIPS algorithms)

3 IAIK — the IAIK cryptographic provider

You can use all of the algorithm implementations supplied by the Entrust and IAIK cryptographic service providers when working in FIPS mode. Using algorithms from CSPs other than these means that your Toolkit-based application will not be operating in FIPS mode.

Note

If you have reason to move the Entust provider from its position as the second registered provider, your application must specify the Entrust provider in a call to `javax.crypto.Cipher.getInstance` to use your chosen FIPS mode algorithm. For example:

```
javax.crypto.Cipher.getInstance("cipherName", "Entrust");
```

where `cipherName` is the chosen algorithm.

Security considerations

512-bit RSA

Using the `CredentialCreator` class, your application can generate 512-bit RSA keys. However, RSA keys of this length are not secure. 512-bit numbers can be factored within a limited time frame using a large, but feasible amount of computing resources. To maintain interoperability, however, the Toolkit supports 512-bit RSA.

PKCS #1 version 1.5

In certain environments, RSA encryption based on PKCS #1 version 1.5 is not secure. Entrust recommends that for RSA encryption you use PKCS #1 version 2.01, the most recent and secure edition of the standard.

Client's system time

To ensure that operations such as certificate validation, signature verification, and SSL/TLS communications work correctly, the system time on the computer that is running the Toolkit-based application must be reasonably accurate (+/- 10 minutes between client and PKI, and between SSL client and SSL server) .

Chapter 4

Certificate management

This section discusses various certificate management tasks (such as validating certificate extensions, adding trusted certificates, and handling certificate and authority revocation lists) that you can build into yourToolkit-based application.

Topics in this section:

- Certificate validation
- Extended certificate validation
- Hierarchical PKI
- Certificate validation and CRLs
- Adding trusted certificates
- Caching certificates, CRLs, and ARLs
- Exporting certificates
- Working with Microsoft Active Directory

Certificate validation

Much of the work involved with certificate validation in the Toolkit is accomplished automatically. For example, during a PKCS #7 encoding operation, recipient's certificates are validated automatically when your application calls the `com.entrust.toolkit.PKCS7EncodeStream.setRecipients` method. In a PKCS #7 decoding operation, the signers' certificates are validated automatically in the `com.entrust.toolkit.PKCS7DecodeStream.read` method once the entire message has been read.

The Certification Authority (CA) in a PKI is responsible for publishing certificate revocation lists (CRLs) for the entities it certifies. CAs usually post CRLs to the X.500 Directory server. Software applications, such as those built with the Security Toolkit for Java, commonly retrieve CRLs using the Lightweight Directory Access Protocol (LDAP) specified in RFC2253 (http://www.ietf.org/rfc/rfc2253.txt).

Earlier releases of the Toolkit recognized CRL distribution points (CDPs) in X.509 certificates in the X.500 distinguished name (DN) format. Release 6.1 of the Toolkit introduces support for CDPs expressed as LDAP and HTTP URLs. The former allows revocation lists stored in an LDAP directory to be checked, the latter allows for revocation checking on HTTP Web servers.

The Toolkit's classes that handle the retrieval of CRLs recognize CDPs (single and multiple CDPs) expressed in any of the following formats:

- As a DN (RFC2253 — http://www.ietf.org/rfc/rfc2253.txt) — for example, `cn=CRL1,o=yourCompany,c=CA`
- As an LDAP URL (RFC2253 — http://www.ietf.org/rfc/rfc2253.txt) — for example, `ldap://LDAPServer/o=yourCompany,c=CA/??sub?(cn=newCRL1)`
- As an HTTP URL — for example, `http://www.yourCompany.com/crlfile.crl`

Note

Upon retrieval of CRLs from CDPs, the Toolkit's classes search for CDPs in the order shown in the list to improve performance.

Two new classes in the Toolkit's `com.entrust.toolkit.util` package, `LdapURL` and `LdapUrlParser`, respectively represent and parse LDAP URLs.

When performing revocation checking in a Directory, the Toolkit parses any LDAP URLs it reads into DN format using the `com.entrust.toolkit.util.LdapUrlParser` class. The `HttpCRLRS` and `DirectoryCRLRS` classes extend the `X509CRLRS` class, all in the `com.entrust.toolkit.x509.revocation` package, to handle CRLs retrieved using CDPs expressed as HTTP and LDAP URLs respectively.

Note

The Toolkit's classes that deal with S/MIME do not provide for the automatic validation of encryption and verification public certificates. You must ensure that you explicitly include validation procedures for these certificates in your S/MIME applications. Refer to the sections of the Programmer's Guide that describe the Toolkit's S/MIME operations for more information.

If you want to validate a certificate explicitly, your application should call the `User.validate(X509Certificate certificate)` method in the `com.entrust.toolkit` package.

```
user.validate(cert);
```

Where,

- `user` is a `com.entrust.toolkit.User` object
- `cert` is an `iaik.x509.X509Certificate` object

Extended certificate validation

The Toolkit enables you to extend certificate validation. Your applications might have to handle users whose certificates are issued by different certification authorities (CA), each with different policies. This means that you must implement certificate validation for each user that your application encounters in a certificate chain. A certificate chain is constructed of one or more cross-certificates, which can be validated using a trusted CA verification certificate. A valid certificate chain contains the certificates needed to prove that a trusted CA has approved Cx's verification public key, where the trusted CA is at the start of the chain, and Cx is at the end of the chain.

Abstract base classes and interfaces in the Toolkit let you add certificate sources, Testlets, and revocation information so that you can process new X.509 certificate extensions, alternative certificate sources, and alternative revocation information. To extend certificate validation, use the following classes depending upon whether you are extending validation for certificates or for CRLs.

- Certificate sources — implement the `CertificateStore` abstract base class and attach a `CertificateStore` object as storage for certificate information.
- Testlets — implement one of the following Testlets:
  - `CertTestlet` — to validate certificate extensions
  - `CRLTestlet` — to validate certificate revocation lists (CRL) extensions
  - `CRLEntryTestlet` — to validate extensions in a CRL entry

  Note

  Not all extensions have to be tested — some extensions provide information only. Testlets are classes that test specific certificate extensions and must implement the methods `init()`, `notify()`, `validate()`, and `reset()`. Each Testlet deals with a single extension. Refer to the `com.entrust.toolkit.x509.testlet` package information in the Javadoc reference for a list of ready-made Testlets.

- Revocation information — implement the `RevocationStore` abstract base class

Note

If you deal with several users in your application, you must set the certificate validation extensions in each user. If you log out a user and subsequently log in, you must re-register your certificate validation extensions.

Certificate extensions are added as follows:

- CertificateStores — added with `User.getCertificateStore().attach(certStore)`
- All Testlets — added with `User.getExtensionTester().addTestlet(testlet, oid)`
- RevocationStores — added with `User.getRevocationStore().attach(revStore)`

Hierarchical PKI

The Toolkit supports a hierarchical PKI trust model. This is a trust model with a single root CA and one or more intermediate CAs between the root and end users.

Hierarchical PKI support in the Toolkit is transparent to the application developer. One method of interest, however, is the `getRootCACertificate` method in the `com.entrust.toolkit.User` class, which returns the root certificate in a PKI hierarchy.

Certificate validation and CRLs

You can control how the Toolkit treats CRLs during the certificate validation process. The Toolkit has two methods for this purpose depending upon whether you want to perform validation using a `com.entrust.toolkit.User` object or a `com.entrust.toolkit.CertVerifier` object (without a `User` object). The methods are:

- `User.requireCRL(boolean required)`
- `CertVerifier.getRevocationStore().requireCRL(boolean required)`

The `requireCRL(boolean)` methods specify whether or not the certificate validation process fails if no CRL is found. The default behavior of these methods is that when you are working online (connected to the Directory) CRL validation will fail if a CRL cannot be found. In offline mode (no connection to the Directory) certificates are considered to be valid (not revoked) if a CRL cannot be found.

When the `boolean` argument `required` is set to `true`, certificate validation fails if no CRL can be found for a certificate. When the argument is set to `false`, certificate validation succeeds regardless of whether or not a CRL can be found for a certificate.

Note

Entrust strongly recommends that you set the `boolean` argument `required` to `true` when working online with Entrust/PKI.

Adding trusted certificates

Adding an additional trusted certificate is a simple process. If you are working with a `com.entrust.toolkit.User` object, use the `User.addTrustedCertificate(cert)` method. If you are working only with a `com.entrust.toolkit.x509.CertVerifier`

object, use the `CertVerifier.getCertificateStore().addTrustedCertificate(trustedCert)` method.

For example, the `User.addTrustedCertificate(cert)` and `CertVerifier.getCertificateStore().addTrustedCertificate(trustedCert)` methods add a trusted certificate from an address book. To enable the addition of trusted certificates, CA certificates and end-user certificates, the `allow ca pab` and `allow end user pab` flags in your policy settings should be set to `true`. If the policy settings prohibit the addition of trusted certificates, the methods throw a `CertificationRootException` exception.

For further details, refer to the Javadoc reference documentation.

Caching certificates, CRLs, and ARLs

The Toolkit allows users to write to, and read from, cache archive files containing X.509 certificates, cross certificates, authority revocation lists (ARLs), and certificate revocation lists (CRLs). These cache archive files also provide sources of certificates, CRLs, and ARLs during the certificate validation process.

You can add certificates, ARLs, and CRLs to their respective cache archive files individually or as entire collections. Neither the certificate cache archive nor the CRL cache archive are enabled by default so you must initialize and add caches to each `User` object for which you intended to use a certificate, CRL, or ARL cache archive.

This feature gives the Toolkit caching capabilities similar to those of other Entrust Authority Toolkit products. The Toolkit recognizes the same file formats for cache archive files as those used by Entrust Entelligence™ Desktop Manager (formerly Entrust/Entelligence):

- Certificate cache file (`.cch` file name extension)
- Cross certificate cache file (`.xcc` file name extension)
- CRL cache file (`.crl` file name extension)
- ARL cache file (`.arl` file name extension)

The following Toolkit packages contain classes allowing you to work with certificate, CRL, and ARL cache archive files:

- `com.entrust.toolkit.x509.certstore.ArchiveCertCache`
- `com.entrust.toolkit.x509.certstore.CertificateGraph`
- `com.entrust.toolkit.x509.revocation.ArchiveCRLCache`
- `com.entrust.toolkit.x509.revocation.CachedCRLRS`
- `com.entrust.toolkit.x509.revocation.DistPointAndCRL`

Handling certificate cache archives

The following code fragments illustrate some of the important classes and methods you can use to handle cache archive files.

- If your application does not instantiate a `User` and log in, you must add Entrust and IAIK as security Providers. To do this, retrieve an instance of the `com.entrust.toolkit.security.provider.Initializer` class and invoke its method.

```
Initializer.getInstance().setProviders(Initializer.MODE_NORMAL);
```

Note

To determine whether the Toolkit is operating in the desired mode, call the `Initializer.getMode()` method.

You can initialize the Toolkit to use the Entrust and IAIK cryptographic service providers, to perform private cryptographic operations on a hardware token (PKCS #11), or so that none of the Toolkit's providers are available. Refer to the Javadoc reference documentation for the `com.entrust.toolkit.security.provider.Initializer` class for more information.

To initialize the Toolkit in FIPS mode, refer to the section entitled **Using the Toolkit in FIPS mode** in the chapter entitled **User and credentials management**.

- Instantiate an `ArchiveCertCache` object.

```
ArchiveCertCache acc = new ArchiveCertCache();
```

Note

The `ArchiveCertCache` class is a subclass of `com.entrust.toolkit.x509.certstore.CertificateStore` and has two constructors:

- `ArchiveCertCache()` — creates a new, empty certificate cache archive.
- `ArchiveCertCache(InputStream is)` — creates a certificate cache archive and adds certificates from an existing cache file specified by the `InputStream` argument.

- Add the contents of an existing cache archive file to the cache archive object.

```
acc.parse(new FileInputStream(<certificate cache archive file>));
acc.parse(new FileInputStream(<cross certificate cache archive file>));
```

Note

You can also add individual certificates, and arrays of certificates, to the cache archive object using:

- `addCertificate(X509Certificate cert)` — adds the certificate specified by the argument to the archive cache.
- `addCertificates(X509Certificate[] certs)` — adds the certificates contained in the `certs` array to the archive cache.

- Obtain information from the certificate cache archive. For example, obtain the certificate associated with a specific DN.

```
CertificateSet cs = acc.find(java.security.Principal dn);
cs = acc.find(java.security.Principal dn);
```

- Write the cache archives to a specified output stream.

```
acc.write(new FileOutputStream(<certificate cache archive file>),
ArchiveCertCache.USER_CERT_ONLY);
acc.write(new FileOutputStream(<cros certificate cache archive file>),
ArchiveCertCache.CROSS_CERT_ONLY);
```

Note

The `write(OutputStream os, int writeMode)` method writes the contents of the certificate cache archive to the `OutputStream` (usually a file) specified by the argument. Depending on the `writeMode` argument you choose, you can write the user certificate, the cross certificates, or both user and cross certificates to the output stream. The `writeMode` arguments are:

- `USER_CERT_ONLY` — specifies that only user certificates should be written to the output stream
- `CROSS_CERT_ONLY` — specifies that only cross certificates should be written to the output stream
- `USER_AND_CROSS_CERT` — specifies that both user certificates and cross certificates should be written to the output stream

Exchanging memory contents with cache archive file contents

The Toolkit has two methods in the `ArchiveCertCache` class that provide a convenient way for you to exchange the contents of the memory cache with the contents of a cache archive file:

- `addMemoryCache(CachedCRLRS cachedCRLRS)` — transfers the certificates in the memory cache (the current `CertificateGraph` object, see the **Handling the memory cache** section below) to the certificate cache.

```
acc.addMemoryCache();
```

- `initMemoryCache(CachedCRLRS cachedCRLRS)` — adds all certificates in the cache archive file (the calling cache object) to the memory cache (the `CertificateGraph` object) making them easily accessible to the certificate validation algorithm.

```
acc.initMemoryCache();
```

Note

To prevent a certificate cache archive file from becoming too large, use the `cleanup()` method to remove all certificates that are no longer valid from the cache archive file.

Handling the memory cache

The memory cache is a `CertificateGraph` object. The `CertificateGraph` class handles certificates and the subject-issuer relationship between them by modelling the mathematical concept of a simple graph. A simple graph is a finite set of nodes connected by links, or edges. In `CertificateGraph`, nodes are represented by certificates and edges are represented by the subject-issuer relationship between the certificates.

The following methods retrieve certificates from the `CertificateGraph`:

- `find(Principal dn)` — returns all certificates for the entity specified in the argument, `dn`.
- `getCertificates()` — returns all certificates in the `CertificateGraph`, the current memory cache.

Handling CRL and ARL cache archives

CRLs and ARLs are stored in an `ArchiveCRLCache` object as a hashtable, where the key for each CRL or ARL is its distribution point name. The distribution point, which can be expressed as a DN, an LDAP URL, or an HTTP URL, is placed in the public certificate. To prevent the CRL cache archive files from becoming too large, use the `cleanup()` method to remove all CRLs and ARLs that are no longer valid.

For the file-based archive, certificates are valid for a length of time after the `nextUpdate` period — specified by the grace period in the policy certificate (default is 2 hours) — or, if there is no `nextUpdate` period, for a week after they were issued. You can modify the one week period using the `setMaximumCRLLifetime(int lifetime)` method in the `com.entrust.toolkit.x509.revocation.CollectionRS` class, if the one week expiry of CRLs and ARLs is inappropriate. If the value is set to zero and if they do not have a `nextUpdate` time specified, CRLs and ARLs do not expire.

CRL and ARL cache archive files are most useful when users are working in offline mode (no connection to the Directory). The following code fragments illustrate some of the important classes and methods you can use in your application to handle CRL and ARL cache archive files.

- Log in.

```
com.entrust.toolkit.User user = new User();
com.entrust.toolkit.credentials.FilenameProfileReader reader =
    new FilenameProfileReader(epf);
user.login(reader, new com.entrust.toolkit.util.SecureStringBuffer(pw));
```

Note

CRLs are not available offline so, when you are working in offline mode, make sure there is a CRL cache archive available.

- Create an empty CRL and ARL cache archive file.

```
ArchiveCRLCache cache = new ArchiveCRLCache(user.getCertVerifier());
```

Note

The `ArchiveCRLCache` class is a subclass of `com.entrust.toolkit.x509.revocation.CachedCRLRS` and has two constructors:

- `ArchiveCRLCache(ValidationInfo validator)` — creates an empty CRL/ARL archive cache. The constructor's argument specifies a validator (retrieved from the `User` object using the `getCertVerifier` method) used to determine the validity of CRLs and ARLs subsequently added to the cache archive file.
- `ArchiveCRLCache(InputStream is, ValidationInfo validator)` — creates a CRL cache archive and reads the data in an existing CRL cache specified by the `InputStream` argument, performing CRL extension and signature validation as it does so.

- Add the contents of the CRL and ARL archive cache file specified by the `InputStream` argument.

```
cache.parse(new FileInputStream(crl));
cache.parse(new FileInputStream(arl));
```

Note

You can also add individual CRLs, ARLs, and arrays of CRLs or ARLs, to the cache archive object using:

- `addCRL(DistPointAndCRL dpcrl)` — adds the specified CRL, or ARL, to the CRL cache archive.
- `addCRLs(DistPointAndCRL[] dpcrls)` — adds the CRLs, or ARLs, contained in the `dpcrls` array to the CRL or ARL cache archive.

- Write the cache archive files to a specified output stream.

```
cache.write(new FileOutputStream(crl),ArchiveCRLCache.CRL_ONLY);
cache.write(new FileOutputStream(arl),ArchiveCRLCache.ARL_ONLY);
```

Note

`write(OutputStream os, int writeMode)` — writes the contents of the CRL or ARL cache archive files to the `OutputStream` specified in the argument. Depending on the `writeMode` argument you choose, you can write CRLs, ARLs, or both CRLs and ARLs, using Entrust `.crl` and `.arl` file formats, to the output stream. The `writeMode` arguments are:

- `CRL_ONLY` — specifies that only CRLs should be written to the output stream.
- `ARL_ONLY` — specifies that only ARLs should be written to the output stream.
- `CRL_AND_ARL` — specifies that both CRLS and ARLs should be written to the output stream.

- Two other methods in the `ArchiveCRLCache` class provide a convenient way for you to exchange the contents of the memory cache with the contents of a cache

archive file:

- `addMemoryCache()` — adds the CRLs, or ARLs, in the memory cache to the CRL archive cache.
- `initMemoryCache()` — adds all CRLs and ARLs that are currently in the archive cache file (the calling cache object) to the memory cache.

- To remove from the cache archive file a CRL, or an ARL, that corresponds to a specific distribution point, use the `removeCRL(Name distPoint)` method.

You can also retrieve a user's cached CRLs after the user has logged in online.

1 Using methods from the `com.entrust.toolkit.x509.revocation.CachedCRLRS` and `com.entrust.toolkit.x509.revocation.CollectionRS` classes, create a `CachedCRLRS` object and retrieve a certificate verifier, a revocation store, and a CRL cache containing cached revocation lists.

```
CachedCRLRS cachedCRLRS =
user.getCertVerifier().getRevocationStore().getMemoryCRLCache();
```

2 Create an `com.entrust.toolkit.x509.revocation.ArchiveCRLCache` object and add the CRLs and ARLs in the memory cache retrieved in the previous step to the new archive cache.

```
ArchiveCRLCache archiveCRLCache = new ArchiveCRLCache(user.getCertVerifier() );
archiveCRLCache.addMemoryCache(cachedCRLRS);
```

3 Write the contents of the archive cache to an output stream .

```
archiveCRLCache.write(System.out, ArchiveCRLCache.CRL_AND_ARL);
```

Note

The `ArchiveCRLCache.CRL_AND_ARL` field specifies that both CRLS and ARLs should be written to the output stream. You can specify that only ARLs should be written to the output stream using the `ARL_ONLY` field, or that only CRLs should be written to the output stream using `CRL_ONLY`.

The DistPointAndCRL and CachedCRLRS classes

The `DistPointAndCRL` class encapsulates in one object an X509 CRL and the distribution point from which it was retrieved. The constructor `DistPointAndCRL(Name distPoint, X509CRL crl)` takes a CRL and its distribution point as arguments. The class has the following two accessor methods:

- `getDistPoint()` — returns the distribution point from which the CRL was retrieved.
- `getCRL()` — returns the CRL.

The `CachedCRLRS` class is a revocation store for cached CRLs. The `getCRLs()` method of `CachedCRLRS` returns an array of `DistPointAndCRL` objects from which you can retrieve specific distribution points and CRLs using the `getDistPoint()` and `getCRL()` methods belonging to the `DistPointAndCRL` class.

Exporting certificates

The PKCS #7 specification permits the transmission of messages that contain certificates only, and do not contain secured data. Such messages are called certificates-only messages because, although they contain certificates, they have no other content. Certificates-only messages provide a simple method of transmitting certificates between users.

Create a `PKCS7EncodeStream` object using the `PKCS7EncodeStream(User, OutputStream, int)` constructor.

```
PKCS7EncodeStream encoder =
    new PKCS7EncodeStream(user,
                          new FileOutputStream(<output file>),
                          PKCS7EncodeStream.EXPORT_CERIFICATES);
```

Where,

- `user` is a logged in user
- `FileOutputStream(<output file>)` specifies a file as the output stream
- `PKCS7EncodeStream.EXPORT_CERIFICATES` is the operation constant for exporting certificates

Working with Microsoft Active Directory

Microsoft Active Directory™, a component of Microsoft® Windows® 2000, is a directory service designed to enable the users and resources of a computer network to work together . Active Directory enables interoperability between Microsoft Windows and other operating systems, centralizes the management of users on a network, and includes features that allow these tasks to be performed securely. For detailed information about Microsoft Active Directory, refer to the Microsoft Windows 2000 Web site (http://www.microsoft.com/windows2000/technologies/directory/ad/default.asp).

Microsoft Active Directory stores users' encryption public certificates, CA certificates, and CRLs in its Users area. The contents of the Users area cannot be read anonymously, so client applications must perform an authentication procedure before Active Directory allows them to read its contents. Active Directory supports three methods of authentication:

1 Kerberos — a network authentication protocol that provides strong authentication for client-server applications. Kerberos was developed by the Massachusetts Institute of Technology (MIT) (http://web.mit.edu/kerberos/www/).

2 NT LAN Manager (NTLM) — an undocumented authentication protocol used by Microsoft Internet Information Server (IIS), Microsoft Internet Explorer (IE), and Microsoft Active Directory.

3 Simple Authentication — a protocol that performs authentication using a user's Microsoft Windows name and Windows login password. The password is sent to

the server in clear text. Refer to RFC 2251 and RFC 2829 refer for more information.

The Kerberos authentication protocol is not implemented in this release of the Toolkit.

NTLM relies on user credentials that are acquired during a Microsoft Windows login operation. Client applications and applets written in Java do not have access to these login credentials, nor would they necessarily be running in a Microsoft Windows operating system. These limitations rule out NTLM as a viable authentication protocol for applications built with the Toolkit.

Simple Authentication is the only suitable method for applications built with the Security Toolkit for Java to use when performing the authentication step required for access to Microsoft Active Directory. However, simple authentication sends the user's Windows login password in the clear over the network to the Active Directory. This is not appropriate for most architectures, so the Toolkit provides a proxy server for the Active Directory. Remote clients authenticate to the proxy over an SSL tunnel, and only the proxy server itself uses simple authentication to gain access to the Active Directory.

Communicating with Microsoft Active Directory

Toolkit-based applications can communicate with Microsoft Active Directory using the Toolkit's proxy server (`ActiveDirectoryServer.java` in the `etjava\examples\activeDirectory` folder) and the `com.entrust.toolkit.util.HttpsDirectoryClient` class. The Toolkit's classes allow you to establish a tunnel to an LDAP Directory (in this case, Active Directory) by way of an SSL-enabled proxy server running on the same computer as Active Directory. The client application connects to the proxy server using strong SSL authentication and the proxy server connects to Active Directory using Simple Authentication.

The `etjava\examples\activeDirectory` folder contains sample applications that demonstrate how to use the Toolkit's proxy server to communicate with Active Directory. The properties file, `activeDirectoryProxy.properties`, specifies the configuration for the SSL-enabled server (`ActiveDirectoryServer.java`) that is part of the sample application, and that acts as a proxy for Microsoft Active Directory. Refer to the sample's readme document (`etjava\examples\activeDirectory\activeDirectory_readme.html`) for more information.

Note

If possible, configure Microsoft Active Directory so that it permits simple authentication only of the user who represents the proxy server. This ensures that all user requests are routed through the proxy server with client-authenticated SSL. If such a configuration is not possible, put Active Directory behind a firewall.

Toolkit procedure — logging in and reading certificates in Microsoft Active Directory

The procedure for logging in a user whose X.509 certificates are held in Active Directory is similar to that used for any LDAP Directory. To log in a user and read certificates in

the Active Directory, complete the following steps:

1 Obtain the user's password as a `SecureStringBuffer`.

```
SecureStringBuffer password =
    new SecureStringBuffer(new StringBuffer(<user's password>));
```

2 Instantiate a `User` object to represent the user.

```
User user = new User();
```

3 Instantiate a credential reader, in this case a `FilenameProfileReader`, passing the location of the credentials file as an argument.

```
FilenameProfileReader credReader =
    new FileInputStream(<path to Entrust Profile>);
```

Note

If you use `StreamProfileReader` and `StreamProfileWriter` to read from, and write to, a credentials file managed by instances of the `FileInputStream` and `FileOutputStream` classes, ensure that you specify different file names for the input and output streams. When you open a file with `FileOutputStream`, the file is always overwritten and when you try to read the same file using `FileInputStream`, the file will be empty. To prevent the destruction of the credentials file you are working with, ensure that you read from, and write to, files with different file names. If you use `StreamProfileReader` and `StreamProfileWriter`, use a `ByteArrayOutputStream` to write the credentials temporarily to memory as a precaution against the destruction of the credentials file.

The Toolkit classes called `FilenameProfileReader` and `FilenameProfileWriter` — used to read and write Entrust Profiles — exclude the possibility of overwriting or destroying credentials stored as files.

4 Instantiate a credential writer, in this case a `FilenameProfileWriter` object.

```
CredentialWriter credWriter =
    new FilenameProfileWriter(<.epf file - path and file name>, null, 0);
user.setCredentialWriter(credWriter);
```

Note

Use `null` and zero as the second and third arguments for algorithm and hash count respectively. This ensures that the constructor uses the default values for algorithm and hash count.

5 Create an `HttpsDirectoryClient` object to control the transmission of data to and from the Active Directory by way of a proxy server.

```
String directoryServerURL = "http://" + <AD proxy server IP address>
        + ":" + <AD proxy server port number>;
LdapDirectory dir = new HttpsDirectoryClient(directoryServerURL, 0);
```

6 Create a `ManagerTransport` object to control communication with the Registration Authority of the CA.

```
int authorityPortNumber = 829;
ManagerTransport emt = new ManagerTransport(managerIPAddress, authorityPortNumber);
```

Note

The key management server port number for PKIX-CMP communications is 829.

7 Set the connections prepared in the previous two steps.

```
user.setConnections(dir, emt);
```

8 Call the `User.login` method to complete the user's authentication.

```
int status = user.login(credReader, password);
```

9 Check the password status after logging in.

```
if((status & User.WARNING_PW_EXPIRED) != 0)
{
    System.out.println("WARNING_PW_EXPIRED");
    System.out.println("Password lifetime was " +
        user.getClientSettings().getPasswordRuleTester().getExpirationTimeInWeeks() + "
weeks");
}

if((status & User.WARNING_PW_NOT_VALID) != 0)
{
    System.out.println("WARNING_PW_NOT_VALID");
}
```

10 Read certificates in the Active Directory.

```
String caName = user.getCaCertificate().getSubjectDN().toString();
System.out.println("CA distinguished name: " + "\"" + caName + "\"");
byte[][] attr = user.getDirectory().getAttr(caName, "caCertificate");

if(attr.length > 0)
{
    FileOutputStream ostream = new FileOutputStream("caCertificate.cer");
    ostream.write(attr[0]);
    ostream.close();
}
```

Remarks

The `etjava\examples\activeDirectory` folder in the Toolkit package contains a proxy LDAP server application and sample applications that demonstrate the log in and credential creation operations. Refer to the **activeDirectory Readme** document (`etjava\examples\activeDirectory\activeDirectory_readme.html`) for a description of the sample applications and instructions for their use.

Chapter 5

Electronic message management

This section describes the Toolkit's support for PKCS #7 and S/MIME (version 2 and version 3) cryptographic operations for securing electronic messages using digital signatures and encryption.

Topics in this section:

- Concepts
- Using the PKCS7EncodeStream class
- Using the PKCS7DecodeStream class
- Using the ECDSA
- S/MIME version 2
- S/MIME version 3

Concepts

Public key cryptography standards

The Public Key Cryptography Standards (PKCS) define standards for cryptographic operations that use public key cryptography. PKCS #7 is the Cryptographic Message Syntax standard describing a syntax for expressing messages that have been cryptographically secured. PKCS #7 supports recursion so that you can use multiple levels of digital signatures and envelopes.

Digital signatures

A digital signature is an electronic signature — the result of applying a hash function to data and encrypting the resulting hash value with the user's signing private key. Anyone who has the corresponding verification public key can verify the digital signature and be sure of the identity of the sender of the data and that the original data remains intact. You can use a digital signature in your application to sign any kind of data, either encrypted or in plain text.

Streams

As part of the Toolkit's high-level API, the `PKCS7EncodeStream` and `PKCS7DecodeStream` classes provide you with the capability to encode and decode data of arbitrary length, and to verify signatures and validate signers', recipients', and user's certificates automatically. `PKCS7EncodeStream` and `PKCS7DecodeStream` extend respectively, `FilterOutputStream` and `FilterInputStream` in the `java.io` package, so data is processed as it is being read or written. Input and output streams can be of any type — files or sockets for example.

Using streams for PKCS #7 operations is tolerant of exceptions, such as a certificate chain validation failure, because recipients, having retrieved complete blocks of data, are given the option of dealing with the data as they see fit. Automatic signature verification and certificate validation means that there is no requirement for calling other methods to perform these checks.

Sets

Another concept important to understand when performing PKCS #7 operations with the Toolkit is that of sets — specifically certificate sets. The `CertificateSet` class extends the more generic `Set` class in the `com.entrust.toolkit` package, and represents a set of X.509 certificates. In an application using PKCS #7 operations, the public key certificates belonging to the recipients of your message are added to a `CertificateSet` object, which you can think of as a set of certificates containing only one instance of each certificate. Another `CertificateSet` object contains the set of rejected (untrusted) recipients, if there are any, once the certificates have been processed. If all recipients are trusted, a set of rejected certificates is not created.

Set concepts

The Toolkit uses the concept of sets in classes such as `iaik.security.ssl.CipherSuiteList` and `com.entrust.toolkit.CertificateSet`. To understand how these classes handle groups of related objects (or sets) you should be familiar with the concepts of basic set theory. The `CertificateSet` class is a subclass of the `com.entrust.toolkit.Set` class, which has the following methods:

- `union()`
- `intersection()`
- `subtract()`

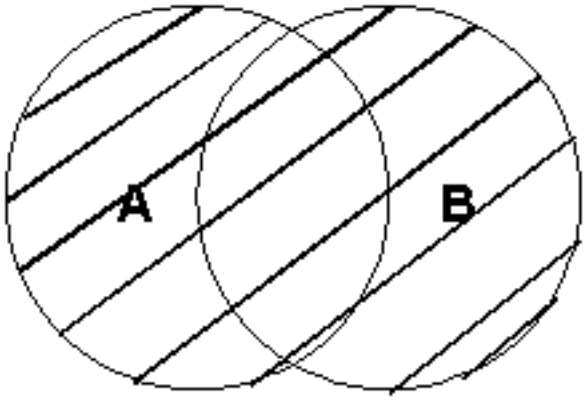
Each of these methods takes a `Set` object as its only argument. The methods perform the mathematical set operations of union, intersection, and difference taking the calling `Set` and the method's argument as operands and returning a `Set` object as the result of the operation.

In the following diagrams, which represent the mathematical set operations discussed above, the term, element, refers to an individual member of a set. The sets in the Toolkit are ordered sets — meaning that each member, or element, of the set is unique and appears only once in the set.

Union

The **union** of two sets **A** and **B**, is the set of elements obtained by combining all the elements of **A** and **B**.

The union of two sets, A and B

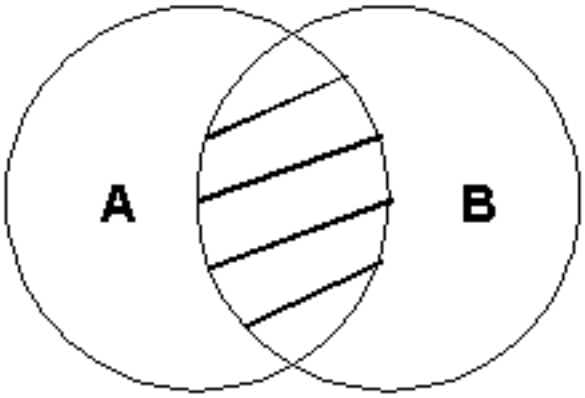


A union B

Intersection

The **intersection** of two sets **A** and **B** is the set of all elements common to both A and B.

The intersection of two sets, A and B

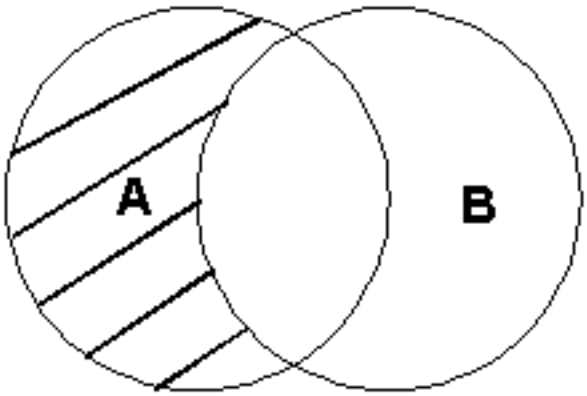


A intersection B

Difference

The difference set of two sets **A** and **B** (**A/B**), is the set of elements that belong only to A — not to B nor to the intersection of A and B.

The difference set A\B



Set difference A\B

Using the PKCS7EncodeStream class

The `PKCS7EncodeStream` class signs and encrypts, encrypts, or signs data in accordance with PKCS #7. The class has two constructors:

- `PKCS7EncodeStream(User, OutputStream, int)`
  - used to encrypt and sign (or just to sign) data and write it to the specified output stream

- `PKCS7EncodeStream(User, OutputStream, OutputStream)`
  - used to clear sign data and write the data and digital signature to two individually specified output streams

You can perform the following operations on your data and messages using the `PKCS7EncodeStream` class:

- Encrypt and sign
- Encrypt only
- Sign only
- Clear sign

The `PKCS7EncodeStream` class also allows you to export certificates.

When you are building applications or applets with PKCS #7 capabilities, ensure that you include the `entp7.jar` file in your operating system's `CLASSPATH` environment variable or in the `-classpath` option of your compiler. This jar file is in addition to the `entbase.jar` and `entuser.jar` files that should be in your classpath whenever you are working with the Toolkit.

The important methods in the `PKCS7EncodeStream` class include:

- `flush()`— to force buffered output to be written to the output stream.
- `setBlockSize()` — to specify the size of the data part of the underlying ASN.1 octet string (specifying a block size of 0 writes the data in one octet string).
- `write()` — to write the data to the output stream.
- `close()` — to close the output stream and release any associated resources.

Message recipients for encrypted messages include those whose public key certificates you specify in the certificate set. Always included in the certificate set as the default recipient is the certificate of the `User` object encrypting the data. The `setRecipients()` method validates the set of certificates and those that fail are removed from the set and returned as a set of rejected certificates. If no certificates fail, `setRecipients()` returns `null`.

Algorithms

For encryption and signing operations the encryption and message digest algorithms are set to CAST and SHA1 by default. You can specify other algorithms by calling the `setEncryptionAlgorithm()` and `setDigestAlgorithm()` methods in the `PKCS7EncodeStream` class. These methods take `AlgorithmID` objects as arguments.

The `AlgorithmID` class in the `iaik.asn1.structures` package statically registers a number of public key, key exchange, symmetric, signature, and message digest algorithms. For example, `RC4` is the AlgorithmID for the RC4 stream cipher algorithm. It has the following properties:

- ObjectID = "1.2.840.113549.3.4"
- Name = "RC4"
- ImplementationName = "RC4/ECB/NoPadding"

Refer to the Javadoc reference for a list of the algorithms registered by `iaik.asn1.structures.AlgorithmID` and for an explanation of how to register object identifiers other than those specified.

Encoding a PKCS #7 message

The following list outlines the steps to take to encode a PKCS #7 message:

- Instantiate, and log in, a user.
- Create a `PKCS7EncodeStream` object.
- Select the encryption algorithm you want to use — optional.
- Select the recipients of your message — optional.
- Write the data to an output stream.
- Close the output stream to complete the operation.

Your application should import classes from the following Toolkit packages:

- `com.entrust.toolkit`
- `com.entrust.toolkit.credentials`
- `com.entrust.toolkit.util`
- `com.entrust.toolkit.x509.directory`
- `com.entrust.toolkit.exceptions`
- `iaik.x509`
- `iaik.asn1.structures` (optional)
- `java.io`

Toolkit procedure — encrypting and signing data

To encrypt and sign a PKCS #7 message with the Toolkit, your application needs the following information:

- User's credentials.
- User's password.
- Location of the data to be encrypted.
- Output stream.
- IP address of the Directory (to retrieve recipients' public key certificates).

- Recipient, or recipients, of the message.

The following procedure demonstrates the encrypting and signing process and passes information to the application directly from the command line.

1 Instantiate a User, set the connection to the Directory (if the user provides the IP address), and log in.

```
FileInputStream credentials =
    new FileInputStream (<location of user credentials>);
SecureStringBuffer password =
    new SecureStringBuffer(new String(<user's password>));

User user = new User();

if (<IP address> != null)
{
  JNDIDirectory dir = new JNDIDirectory (<ip address>, <port number>);
  user.setConnections(dir, null);
}

CredentialReader credReader = new StreamProfileReader(credentials);
user.login(credReader, password>);
```

Note

Refer to the section entitled **Identifying a user** for more details on logging in a user.

2 Create a `PKCS7EncodeStream` object using the `PKCS7EncodeStream(User, OutputStream, int)` constructor.

```
PKCS7EncodeStream encoder =

    new PKCS7EncodeStream(user,
                          new FileOutputStream(<output file>),
                          PKCS7EncodeStream.SIGN_AND_ENCRYPT);
```

Where,

- `user` is a logged in user
- `FileOutputStream(<output file>)` specifies a file as the output stream
- `PKCS7EncodeStream.SIGN_AND_ENCRYPT` is the operation constant for signing and encrypting data

Note

The additional operation constants are:

- `ENCRYPT_ONLY`
- `SIGN_ONLY`
- `EXPORT_CERTIFICATES`
- `CLEAR_SIGN`

3 Create a `CertificateSet` object and load it with the recipients' public key certificates.

```
X509Certificate[] certs = new X509Certificate[1];
certs[0] =
    new X509Certificate(new FileInputStream(<path to recipient's certificate>));
```

```
CertificateSet certSet = new CertificateSet(certs);

while (<command line list of certificates is not empty>)
{
  certs[0] =
      new X509Certificate(new FileInputStream(args[<next recipient>]));
  certSet.addElement(certs[0]);
}
```

Note

The code fragment illustrates the case where more than one recipient has been specified at the command line `args[]` and the `CertificateSet` is constructed using an array of X.509 certificates.

The `encode.java` sample application in the `etjava\examples\pkcs7` folder contains a specific example of this step.

4 Create a `CertificateSet` object to hold rejected certificates and call `CertificateSet.setRecipients()` to validate the recipient's certificates.

```
CertificateSet rejectedCerts = encoder.setRecipients(certSet);
```

5 Specify the digest and encryption algorithms you want to use.

```
encoder.setDigestAlgorithm(AlgorithmID.sha);

encoder.setEncryptionAlgorithm(AlgorithmID.rsaEncryption);
```

6 If the size of the data you are sending is large, use `PKCS7EncodeStream.setBlockSize()` to specify a block size in bytes.

```
encoder.setBlockSize(1024);
```

Note

By default, the digest and encryption algorithms are SHA1 and CAST respectively. You can specify other algorithms by calling `PKCS7EncodeStream.setDigestAlgorithm()` and `PKCS7EncodeStream.setEncryptionAlgorithm()`.

7 Specify the location of the input data and write the encrypted and signed data to the output stream.

```
FileInputStream input_data =
    new FileInputStream(<location of input data>);

byte[] b = new byte[128];
int i = input_data.read(b);

while (i >= 0)
{
  encoder.write(b, 0, i);
  i = input_data.read(b);
}
```

8 Close the output stream when the write operation is complete.

```
encoder.close();
```

Remarks

Using a digital signature assures the recipient of a message that the data he or she has received has not been modified in any way, as well as verifying the identity of the sender. Encrypting the data, to prevent unauthorized viewing, and applying a digital signature to your message provides for maximum security.

Toolkit procedure — encrypting data

You need not apply a digital signature to your message before sending it electronically, but you might want to encrypt the data to prevent unauthorized viewing or unauthorized use. To encrypt data, a user must be logged in. If you need to, refer to the section entitled **Identifying a user** to review the log in procedure. The encryption process is similar to that used to both encrypt and sign data except that you must specify the `ENCRYPT_ONLY` operation in the constructor of the `PKCS7EncodeStream` object as follows:

Create a `PKCS7EncodeStream` object using the `PKCS7EncodeStream(User, OutputStream, int)` constructor.

```
PKCS7EncodeStream encoder =
    new PKCS7EncodeStream(user,
                          new FileOutputStream(<output file>),
                          PKCS7EncodeStream.ENCRYPT_ONLY);
```

Where,

- `user` is a logged in user
- `FileOutputStream(<output file>)` specifies a file as the output stream
- `PKCS7EncodeStream.ENCRYPT_ONLY` is the operation constant for encrypting data

Toolkit procedure — signing data

A digital signature allows a message's recipient to determine whether or not a message has been modified since it left the sender and to authenticate its origin. The following code fragment demonstrates how to use the Toolkit to sign a message without encrypting it. Users must be logged in before they can perform cryptographic operations. To sign data without encrypting it, specify the `SIGN_ONLY` operation in the `PKCS7EncodeStream` constructor.

Create a `PKCS7EncodeStream` object using the `PKCS7EncodeStream(User, OutputStream, int)` constructor.

```
PKCS7EncodeStream encoder =
    new PKCS7EncodeStream(user,
                          new FileOutputStream(<output file>),
                          PKCS7EncodeStream.SIGN_ONLY);
```

Where,

- `user` is a logged in user
- `FileOutputStream(<output file>)` specifies a file as the output stream

- `PKCS7EncodeStream.SIGN_ONLY` is the operation constant for digitally signing data

Toolkit procedure — clear signing a message

The Toolkit supports the technique of clear signing a message, or data, to authenticate its origin. Clear signing allows you to attach a digital signature to an unencrypted message, or to unencrypted data, as a means of proving that it was you who sent it.

You can clear sign a message for recipients who have mixed or unknown mail readers. Those who have S/MIME compatible software have the benefit of being able to verify a digital signature, and those without S/MIME compatible software are still able to read the message. To clear sign a message, ensure the user is logged in and use either `PKCS7EncodeStream` class constructor.

```
PKCS7EncodeStream encoder =
    new PKCS7EncodeStream(user,
                          new FileOutputStream(<output file one>),
                          new FileOutputStream(<output file two>));
```

Where,

- `user` is a logged in user
- `FileOutputStream(<output file one>)` writes the message to the specified file

  Note

  Set this parameter to `null` if you do not want to write the data you are clear signing to an output stream.

- `FileOutputStream(<output file two>)` writes the signature to the specified file

```
PKCS7EncodeStream encoder =
    new PKCS7EncodeStream(user,
                          new FileOutputStream(<output file>),
                          PKCS7EncodeStream.CLEAR_SIGN);
```

Where,

- `user` is a logged in user
- `FileOutputStream(<output file>)` specifies a file as the output stream
- `PKCS7EncodeStream.CLEAR_SIGN` is the operation constant for clear signing data

Using the PKCS7DecodeStream class

The `PKCS7DecodeStream` class decrypts data that has been encrypted according to the PKCS #7 standard, and verifies digital signatures when the end of the input stream is reached. The class has two constructors:

- `PKCS7DecodeStream(User, InputStream)`
  - decrypts and verifies the data in an input stream object, and also retrieves

exported certificates

- `PKCS7DecodeStream(User, InputStream, InputStream)`
  - decrypts and verifies data that has been clear signed

The Toolkit can read messages that are signed with more than one key. Verification of the digital signatures takes place when the end of the input stream is reached.

When you are building applications or applets with PKCS #7 capabilities, ensure that you include the `entp7.jar` file in your operating system's `CLASSPATH` environment variable or in the `-classpath` option of your compiler. This jar file is in addition to the `entbase.jar` and `entuser.jar` files that should be in your classpath whenever you are working with the Toolkit.

Important methods

The important methods in the `PKCS7DecodeStream` class include:

- `getOperation` — returns the security operation (sign and encrypt, sign only, clear sign, encrypt only, and export certificates) that was performed on the input stream being read.
- `getEncryptionAlgorithm`— returns the encryption algorithm (as an `AlgorithmID` object) used to encrypt the data — `null` if the data are not encrypted
- `getDigestAlgorithm` and `getSignerCertificate` — return the signing information for the specified signature
- `getIncludedCertificates` — returns the certificates that are included in the PKCS #7 message
- `read()` and `read(byte[] b, int off, int len)` — validate the signatures and certificates that were used to sign the data when the end of the input stream is reached.
- `close()` (inherited form the `FilterInputStream` class in the `java.io` package) — closes the underlying stream

Decoding a PKCS #7 message

The following list outlines the steps to take to decode a PKCS #7 message:

- Instantiate a user and log in.
- Create a `PKCS7DecodeStream` object and read data from the specified input stream.
- Determine the encoding security operation (sign and encrypt, sign only, clear sign, encrypt only, or export certificates) used by the sender, the algorithm used to encrypt the data, and information about the entity, or entities, that signed the message — optional.
- Close the input stream to complete the operation.

Your application should import classes from the following Toolkit packages:

- `com.entrust.toolkit`
- `com.entrust.toolkit.credentials`
- `com.entrust.toolkit.util`
- `com.entrust.toolkit.x509.directory`
- `com.entrust.toolkit.exceptions`
- `iaik.x509`
- `java.io`

Toolkit procedure — decoding encrypted and signed data

To decrypt a signed and encrypted PKCS #7 message with the Toolkit, your application needs the following information:

- User's credentials
- User's password
- Location of the data to be decrypted (input stream)
- IP address of the PKI Directory (to retrieve public key certificates)

The following procedure demonstrates the decryption process and passes information to the application directly from the command line.

1 Instantiate a `User`, set the connection to the Directory (if the user provides the IP address), and log in. (See the section entitled **Identifying a user** for more details on logging in a user).

```
FileInputStream credentials =
    new FileInputStream (<location of user credentials>);
SecureStringBuffer password =
    new SecureStringBuffer(new String(<user's password>));

User user = new User();

if (<IP address> != null)
{
  JNDIDirectory dir = new JNDIDirectory (<IP address>, <port number>);
  user.setConnections(dir, null);
}

CredentialReader credReader = new StreamProfileReader(credentials);
user.login(credReader, password>);
```

Note

The IP address in the `JNDIDirectory` constructor is optional. If the message is not signed, the IP address is not required. If the message is signed, the IP address is used only for CRLs and cross certification.

2 Create a `PKCS7DecodeStream` object using the `PKCS7DecodeStream(User, InputStream)` constructor.

```
PKCS7DecodeStream decoder =
    new PKCS7DecodeStream(user,
```

```
                                new FileInputStream(<input file>));
```

Where,

- `user` is a logged in user
- `FileInputStream(<input file>)` specifies the input stream (the source of the encoded data)

3 Read the decrypted and signed data.

```
byte[] b = new byte[128];
int i = decoder.read(b);

while (i >= 0)
{
   i = decoder.read(b);
}
```

4 Optionally, obtain information about the encode operation or digest algorithm used by the sender of the message.

```
int no_of_signatures = decoder.getNumberOfSignatures();
for(int n = 0; n &lt; no_of_signatures; ++n;)
{
  System.out.println("Signature: " + n);
  System.out.println("Digest algorithm: " +
      decoder.getDigestAlgorithm(n));
  System.out.println("Signer certificate: " +
      decoder.getSignerCertificate(n).getSubjectDN().getName());
  System.out.println(decoder.getSignerInfo(n).toString());
}
```

5 Close the input stream to the encrypted data and the output stream to the newly decoded data.

```
decoder.close();
output_data.close();
```

If the operation is clear sign, you can use this `PKCS7DecodeStream` as the second input stream in the constructor for clear signed data, `PKCS7DecodeStream(User, InputStream, InputStream)`.

Use a similar procedure to decode messages encoded using any security operations except clear sign.

Toolkit procedure — decoding clear signed data

To decode a message that has been clear signed, use the `PKCS7DecodeStream(User, InputStream, InputStream)` constructor to instantiate a `PKCS7DecodeStream` object.

```
PKCS7DecodeStream decoder =
    new PKCS7DecodeStream(user,
                          new FileInputStream(<unencrypted data>),
                          new FileInputStream(<digital signature>));
```

Create an output stream to store the data and use the `PKCS7DecodeStream.read()` method to read the data.

```
byte[] b = new byte[128];
int i = decoder.read(b);

while (i >= 0)
{
   i = decoder.read(b);
}
```

The following code fragment shows how you can obtain information about the digital signature (or signatures) in the clear signed message. Here, the information is written to the standard output stream for the system.

```
int no_of_signatures = decoder.getNumberOfSignatures();
for(int n = 0; n &lt; no_of_signatures; ++n;)
{
  System.out.println("Signature: " + n);
  System.out.println("Digest algorithm: " +
      decoder.getDigestAlgorithm(n));
  System.out.println("Signer certificate: " +
      decoder.getSignerCertificate(n).getSubjectDN().getName());
  System.out.println(decoder.getSignerInfo(n).toString());
}
```

Using the ECDSA

The Toolkit provides support for elliptic curve cryptography (ECC) in the form of the elliptic curve digital signature algorithm (ECDSA) — providing digital signature capabilities using the Java Cryptography Extension (JCE) for both stand-alone applications and applications that work with an Entrust PKI. ECC is a relatively new family of public-key algorithms that can provide shorter key lengths and, depending upon the environment and application in which the ECC algorithms are used, improved performance over systems based on integer factorization and discrete logarithms. The ECDSA is the elliptic curve counterpart of the digital signature algorithm (DSA) and uses arithmetic based on elliptic curves. The Toolkit's implementation of ECDSA supports all named curves in ANSI X9.62-10.Oct.1999, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). These curves are pre-registered in the Toolkit.

Release 6.1 of the Toolkit does not integrate support for the ECDSA into much of its high-level API. This means that the methods and operation constants of many high-level classes do not recognize the ECDSA. Exceptions to this are the high-level PKCS #7 and PKCS #12 classes that have been modified to work with the ECDSA in this release. To use the ECDSA, you must work directly with the low-level classes and methods of the JCE. For background information about ECC refer to the technical document entitled **Elliptic Curve Cryptography Support in Entrust** at the Resource Centre on the Entrust Web site (http://www.entrust.com/resources/whitepapers.htm).

Note

Refer to the **Release notes** for information about the elliptic curve requirements for building applications with the Toolkit that interoperate with applications based on the EntrustFile Toolkit. The default named curve used by the Toolkit is `x962p192r1` (OID 1.2.840.10045.3.1).

Example

The com.entrust.toolkit.security.provider package contains the Toolkit classes required to work with curves that are registered in the Toolkit.

The com.entrust.toolkit.security.arithmetic.groups.ellipticCurve and com.entrust.toolkit.security.arithmetic.fields packages contain classes that allow you to work directly with groups and fields — the mathematical structures that underlie elliptic curves.

The following code fragments illustrate how to use the Toolkit's ECDSA classes and low-level JCE classes to generate a public and private key pair, and how to use the key pair to sign a message and verify the signature.

- Specify a registered elliptic curve.

```
ECParameters newParam =
    NamedCurveFactory.getInstance(new ObjectID("1.3.132.0.33"));
```

Note

The com.entrust.toolkit.security.provider.NamedCurveFactory class provides methods that allow you to retrieve named elliptic curves as instances of the ECParamater class. Using the getInstance(ObjectID oid) method, you can retrieve a registered curve by specifying its object identifier. The Javadoc reference material for the NamedCurveFactory class shows the correspondence between the Toolkit's registered curves and their object identifiers (OIDs).

Using the getInstance(int length) method, you can retrieve a registered curve by specifying its field size. The following line of code retrieves a curve with a field size of 224 bits:

```
ECParameters newParam = NamedCurveFactory.getInstance(224);
```

- Create an ECDSA key pair generator.

```
ECDSAKeyPairGenerator kpg =
    (ECDSAKeyPairGenerator) KeyPairGenerator.getInstance("ECDSA");
```

Note

This line of code generates an instance of the java.security.KeyPairGenerator class that implements the ECDSA algorithm, and then uses an explicit cast to create an ECDSAKeyPairGenerator object. It is important to use an explicit cast to ensure that the ECDSAKeyPairGenerator object behaves as expected. The getInstance(String algorithm) method returns an appropriate KeyPairGenerator object only if the requested algorithm is available in the installed provider packages.

- Inititialize the key pair generator.

```
kpg.initialize(newParam);
```

- Generate the key pair.

```
java.security.KeyPair kp = kpg.generateKeyPair();
```

- Retrieve references to the public key and the signing private key.

```
ECDSAPublicKey publicKey = (ECDSAPublicKey) kp.getPublic();
PrivateKey privateKey = kp.getPrivate();
```

- Create a `java.security.Signature` object and sign a message.

```
Signature sig = Signature.getInstance("ECDSA");
sig.initSign(privateKey);
sig.update(new String("Test message to be signed.").getBytes());
byte[] signature = sig.sign();
```

- Verify the signature.

```
Signature ver = Signature.getInstance("ECDSA");
ver.initVerify(publicKey);
ver.update(new String("Test message to be signed.").getBytes());
if (ver.verify(signature))
{
    System.out.println("Signature was verified: It's valid.");
}
else
{
    System.out.println("Signature was verified: It's NOT valid.");
}
```

S/MIME version 2

Secure multipurpose Internet mail extensions (S/MIME) version 2 is a widely used application of PKCS #7 for exchanging messages and data by transport protocols capable of conveying MIME data, such as email and http. S/MIME offers authentication, using digital signatures to validate a sender's identity, and privacy, using encryption to protect a message against unauthorized access. Following the syntax described in PKCS #7, S/MIME specifies how to include encryption information and a digital certificate as part of an email message.

Note

It is important to understand the distinction between S/MIME and PKCS #7. PKCS #7 is a generic specification for secure messaging that can be used with a variety of security mechanisms. S/MIME is an application of PKCS #7, specifically designed for MIME messaging.

To assist your development of Java email and messaging applications, the Security Toolkit Toolkit for Java includes classes that implement the S/MIME version 2 protocol and extend the capabilities of the JavaMail™ API (http://java.sun.com/products/javamail/index.html) and the JavaBeans™ Activation Framework (http://java.sun.com/products/javabeans/glasgow/jaf.html) extensions.

The Toolkit's `etjava\lib` folder includes two mailcap files, `mailcap` and `mailcap.default`. These files are consulted by email applications to determine how to display messages that have an unfamiliar content type. Copy these files into the `lib` folder of your J2SDK installation.

Note

You can also specify a path to the `mailcap` and `mailcap.default` files using the `MailcapCommandMap(java.lang.String fileName)` constructor in the `javax.activation` package. Refer to the Javadoc reference documentation (http://java.sun.com/products/javabeans/glasgow/javadocs/index.html) for the JavaBeans™ Activation Framework (http://java.sun.com/products/javabeans/glasgow/jaf.html).

For detailed information about S/MIME refer to:

- RFC2311 — S/MIME Version 2 Message Specification (http://www.imc.org/rfc2311)
- RFC2312 — S/MIME Version 2 Certificate Handling (http://www.imc.org/rfc2312)

Handling S/MIME messages

The Toolkit's jar file, `entsmime.jar`, contains the `iaik.security.smime` package. S/MIME uses the PKCS #7 format to deliver secure MIME-encoded data objects putting encrypted data in a digital envelope. PKCS #7 supports the encryption and signing of data as well the transfer of certificates, CRLs, and certificate chains. Include `entp7.jar`, `entp10.jar`, `entp12.jar`, `mail.jar` (the JavaMail jar file), and `activation.jar` (the JavaBeans Activation Framework jar file) in your classpath.

The Toolkit supports a complete range of MIME types:

- application/pkcs7-mime
- application/pkcs7-signature
- application/pkcs10
- multipart/signed

An S/MIME message uses a symmetric algorithm — DES, Triple-DES, or RC2 — for encryption, and a public key, or asymmetic, algorithm — RSA — for key exchange and for digital signatures.

The S/MIME applications you write require access to certain environment properties, such as the mail server host name, for example, as well as information such as the email addresses of the sender and the recipient of a message. You can handle these properties easily and efficiently using the methods in the `com.entrust.toolkit.utility.IniFile` class, which let you manipulate the information in an `ini` file. Such a properties file, or `ini` file, lists properties using key-value pairs under section headings as illustrated in the following example.

```
[FirstSectionName]
searchbase=<searchbase X500>
searchexpr=<searchexperession CN, for example>
to=<recipient's email address>
from=<sender's email address>
host=<mailhost>
Profile=<sender's Profile, .epf file>
```

```
[SecondSectionName]
recip=<recipient>
```

```
recip_password=<recipient's mailbox password>
mbox=<mailbox name>
host=<mailhost>
protocol=<mail server's protocol>
Profile=<recipient's Profile, .epf file>
```

Note

Refer to the section in this document entitled **Parameterization** for more information about using the `com.entrust.toolkit.utility.IniFile` class to create initialization and configuration files.

Certificate validation

When you are using the Toolkit's S/MIME API, your application must explicitly provide for the validation of encryption and verification certificates. Refer to the subsection entitled Certificate validation in the Managing certificates section for more information.

Creating, sending, and receiving S/MIME messages

The procedures in the following sections demonstrate how to create, send, and receive S/MIME messages. The code fragments deal with the following kinds of messages between a sender and one recipient.

- Plain
- Signed
- Encrypted
- Signed and encrypted
- Certificates only
- Certificate request

In the `etjava\examples` folder of the Toolkit you will find detailed sample programs that demonstrate these procedures. The documentation that accompanies the sample programs includes instructions about how to set up your Java environment to run the samples.

Toolkit procedure — preparation

The following procedure demonstrates the preparations you need to make in your application before sending an S/MIME message.

1 Create a `User` object to represent the sender of the message.

```
com.entrust.toolkit.User sender = new User();
```

2 Connect to the Directory and to the CA's key management server.

```
com.entrust.toolkit.x509.directory.JNDIDirectory dir =
     new JNDIDirectory(<IP address>, <port number>);
com.entrust.toolkit.util.ManagerTransport emt =
    new ManagerTransport(<IP address>, <port number>);
sender.setConnections(dir, emt);
```

3 Log in.

```
sender.login(fis, <user's password>);
```

4 Retrieve and validate the recipient's CA certificate.

```
byte[][] result =
    dir.Search(<searchbase X500>, <searchexpression, CN for example>, userCertificate);
recipientCertificate = new X509Certificate(result[0]);
sender.validate(recipientCertificate);
```

5 Retrieve the sender's CA and verification certificates.

```
iaik.x509.X509Certificate[] signerCertificates = new X509Certificate[2];
signerCertificates[0] = sender.getVerificationCertificate();
signerCertificates[1] = sender.getCaCertificate();
signerEncryptionCertificate = sender.getEncryptionCertificate();
```

6 Retrieve the sender's signing private key.

```
java.security.PrivateKey signerPrivateKey = sender.getSigningKey();
```

Toolkit procedure — constructing a multipart message

This procedure extends the preparation procedure and demonstrates how to construct a multipart message.

1 Create a `MimeBodyPart` using the `iaik.security.smime.SMimeBodyPart` class.

```
javax.mail.internet.MimeBodyPart mbp1 = new SMimeBodyPart();
mbp1.setText("This is a Test of the Entrust/Toolkit for Java
              S/MIME implementation!\n\n");
```

2 Create an attachment to the message — in this case, the Entrust home page.

```
MimeBodyPart attachment = new SMimeBodyPart();
URL url = new URL("http://www.entrust.com/");
attachment.setDataHandler(new DataHandler(new URLDataSource(url)));
attachment.setFileName("index.html");
```

3 Combine the message parts — the text and the attachment - using the `iaik.security.smime.SMimeMultipart` class

```
javax.mail.internet.Multipart mp = new SMimeMultipart();
mp.addBodyPart(mbp1);
mp.addBodyPart(attachment);
```

4 Wrap the multipart message in a `DataHander` object.

```
javax.activation.DataHandler multipart =
    new DataHandler(mp, mp.getContentType());
```

Toolkit procedures — creating and sending messages

This set of procedures demonstrates how to use the Toolkit to create and send S/MIME messages.

Sending a plain message

1 Perform **Toolkit procedure — preparation** and **Toolkit procedure — constructing a multipart message**.

2 Establish a session with the mail server.

```
java.util.Properties props = new Properties();
props.put("mail.smtp.host", <mailhost>);
javax.mail.Session session = Session.getDefaultInstance(props, null);
```

Note

Refer to the discussion of properties and `ini` files in the section entitled **Handling S/MIME messages**.

3 Create a MIME message.

```
javax.mail.internet.MimeMessage mimeMsg = new MimeMessage(session);
```

4 Set the sender, recipient, and date of the message.

```
mimeMsg.setFrom(new javax.mail.internet.InternetAddress(<sender's email address>));
mimeMsg.setRecipients(Message.RecipientType.TO, InternetAddress.parse(<recipient's email
address>, false));
mimeMsg.setSentDate(new Date());
```

5 Set the message subject and the `DataHandler`.

```
javax.mail.Message messageMsg = mimeMsg;
messageMsg.setSubject("S/MIME: Plain");
if (dataHandler != null)
    messageMsg.setDataHandler(dataHandler);
else
    messageMsg.setText("This is a plain message.\n
                  It is neither signed nor encrypted.\n");
```

6 Send the message.

```
javax.mail.Transport.send(messageMsg);
```

Sending an implicitly signed message.

1 Perform steps 1, 2, 3, and 4 of the procedure **Sending a plain message**.

2 Instantiate a `SignedContent` object.

```
iaik.security.smime.SignedContent sc =
    new SignedContent(true);
```

Note

Use `true` as the argument of the `SignedContent` constructor if you intend to include the message in the signature. Use `false` otherwise.

3 Set the message subject and the `DataHandler`.

```
javax.mail.Message messageMsg = mimeMsg;
messageMsg.setSubject("S/MIME: Implicitly Signed");
if (dataHandler != null)
    sc.setDataHandler(dataHandler);
else
    sc.setText("This message is implicitly signed!\n
                You need an S/MIME-aware mail client to view this message.\n");
```

4 Set the signer's certificates to be included in the S/MIME message.

```
sc.setCertificates(signerCertificates);
```

Note

`signerCertificates` is an array of `X509Certificates` representing the certifcate chain of the signer.

5 Sign the message content.

```
sc.setSigner(signerPrivateKey, signerCertificates[0]);
```

6 Set the message's content type.

```
messageMsg.setContent(sc, sc.getContentType());
```

7 Send the message.

```
javax.mail.Transport.send(messageMsg);
```

Remarks

Implicitly signed messages (those of MIME type application/pkcs7-mime) can be read only with a mail client that supports S/MIME because the content is included in the S/MIME signature (DER encoded ASN.1).

Explicitly signed messages consist of a multipart/signed content where the first part contains the human readable content and the second part contains the signature.

Sending an encrypted message

1 Perform steps 1, 2, 3, and 4 of the procedure **Sending a plain message**.

2 Set the message subject.

```
javax.mail.Message messageMsg = mimeMsg;
messageMsg.setSubject("S/MIME: Encrypted");
```

3 Instantiate an encrypted content object.

```
iaik.security.smime.EncryptedContent ec =
    new EncryptedContent();
```

4 Set the message content.

```
ec.setText("This is the encrypted content!\n");
```

Note

`setText` is a convenience method of the `EncryptedContent` class that sets the given `String` as the object's content with a MIME type of text/plain.

5 Specify recipients (in this case one recipient and the sender) and the key encryption algorithm to use.

```
ec.addRecipient(recipientCertificate, AlgorithmID.rsaEncryption);
ec.addRecipient(signerCertificates[0], AlgorithmID.rsaEncryption);
```

Notes

Check that the `recipientCertificate` is a trusted certificate using the

`com.entrust.toolkit.User.validate` method.

The key encryption algorithm must be the RSA encryption algorithm, `AlgorithmID.rsaEncryption`.

6 Set the symmetric algorithm for encrypting the message.

```
ec.setEncryptionAlgorithm(<algorithm>, <keyLength>);
```

Note

Possible encryption algorithms and key lengths are:

- AlgorithmID.rc2_CBC with 40, 64, or 128 bits
- AlgorithmID.des_CBC
- AlgorithmID.des_EDE3_CBC

7 Set the message's content type.

```
messageMsg.setContent(ec, ec.getContentType());
```

8 Send the message.

```
javax.mail.Transport.send(messageMsg);
```

Sending a signed and encrypted message

1 Perform steps 1, 2, 3, and 4 of the procedure **Sending a plain message**.

2 Set the message subject.

```
javax.mail.Message messageMsg = mimeMsg;
messageMsg.setSubject("S/MIME: Implicitly signed and encrypted");
```

3 Instantiate a `SignedContent` object.

```
iaik.security.smime.SignedContent sc =
    new SignedContent(true);
```

Note

Use `true` as the argument of the `SignedContent` constructor if you intend to include the message in the signature. Use `false` otherwise.

4 Set the `DataHandler`.

```
if (dataHandler != null)
    sc.setDataHandler(dataHandler);
else
    sc.setText("This message is implicitly signed and encrypted.\n"


                You need an S/MIME-aware mail client to view this message.\n");
```

5 Set the signer's certificates to be included in the S/MIME message.

```
sc.setCertificates(signerCertificates);
```

Note

`signerCertificates` is an array of `X509Certificates` representing the certificate chain of the signer.

6 Instantiate an S/MIME encrypted and signed content object.

```
iaik.security.smime.EncryptedContent ec =
    new EncryptedContent(sc);
```

7 Specify recipients (in this case one recipient and the sender) and the key encryption algorithm to use.

```
ec.addRecipient(recipientCertificate, AlgorithmID.rsaEncryption);
ec.addRecipient(signerEncryptionCertificate, AlgorithmID.rsaEncryption);
```

Note

The key encryption algorithm must be the RSA encryption algorithm, `AlgorithmID.rsaEncryption`.

8 Set the symmetric algorithm for encrypting the message.

```
ec.setEncryptionAlgorithm(<algorithm>, <keyLength>);
```

Note

Possible encryption algorithms and key lengths are:

- AlgorithmID.rc2_CBC with 40, 64 or 128 bits
- AlgorithmID.des_CBC
- AlgorithmID.des_EDE3_CBC

9 Set the message's content type.

```
messageMsg.setContent(ec, ec.getContentType());
```

10 Send the message.

```
javax.mail.Transport.send(messageMsg);
```

Note

If you use the `iaik.security.smime.EncryptedContent.writeTo()` method to encrypt content that you want to sign using the `iaik.security.smime.SignedContent.writeTo()` method, ensure that you call the `EncryptedContent.writeTo()` method just once. Calling `EncryptedContent.writeTo()` a second time will cause the signature verification to fail because every time you call the `EncryptedContent.writeTo()` method, the content you are writing is encrypted with a newly generated symmetric key. The `SignedContent.writeTo()` method, however, calculates the signature only once, even if it is called several times, as it assumes that the encrypted content it contains remains unchanged.

Sending a certificates-only message

1 Perform steps 1, 2, 3, and 4 of the procedure **Sending a plain message**.

2 Set the message subject.

```
javax.mail.Message messageMsg = mimeMsg;
messageMsg.setSubject("S/MIME: Certs-only message");
```

3 Determine whether to use old or new S/MIME content types.

```
iaik.security.smime.SMimeParameters.useNewContentTypes(true);
```

Note

Set the argument to `true` to use new S/MIME content types. Set the argument to `false` to use old S/MIME content types.

4 Instantiate a `SignedContents` object specifying that this is to be a certificates-only message.

```
iaik.security.smime.SignedContent sc =
    new SignedContent(true, SignedContent.CERTS_ONLY);
```

Note

Use `true` as the first argument of the `SignedContent` constructor if you intend to include the message in the signature. Use `false` otherwise.

Set the second argument to `SignedContent.CERTS_ONLY` for a certificates-only message.

5 Specify the signer's certificates to be included in the message.

```
sc.setCertificates(signerCertificates);
```

6 Set the message's content type.

```
messageMsg.setContent(sc, sc.getContentType());
```

7 Set the message's file name and attachment parameters.

```
sc.setHeaders(messageMsg);
```

8 Send the message.

```
javax.mail.Transport.send(messageMsg);
```

Sending a certificate request message

This procedure describes how to send a certificate request to a Certification Authority.

1 Perform steps 1, 2, 3, and 4 of the procedure **Sending a plain message**.

2 Set the message subject.

```
javax.mail.Message messageMsg = mimeMsg;
messageMsg.setSubject("S/MIME: Certificate request");
```

3 Instantiate a `PKCS10Content` object.

```
iaik.security.smime.PKCS10Content pc = new PKCS10Content();
```

Note

S/MIME specifies the `application/pkcs10` type as the format for sending a PKCS #10 certification requests to a CA.

4 Create an `iaik.pkcs.pkcs10.CertificateRequest` object.

```
Name subject = new Name();
subject.addRDN(ObjectID.commonName, firstName + " " + lastName);
subject.addRDN(ObjectID.emailAddress, from);
CertificateRequest certRequest;
```

```
certRequest =
    new CertificateRequest(signerCertificate.getPublicKey(), subject);
certRequest.sign(AlgorithmID.sha1WithRSAEncryption, signerPrivateKey);
certRequest.verify();
```

This step supplies the subject's distinguished name and public encryption key to the CA, and signs the request. After verifying the signature, the CA responds to the request by sending a message containing an X.509 public key certificate, or a PKCS #6 extended certificate created from the data it received in the certificate request.

Note

Refer to the `iaik.asn1.structures.Name` and `iaik.pkcs.pkcs10.CertificateRequest` classes in the Javadoc reference documentation for details about this step.

5 Set the PKCS #10 request.

```
pc.setCertRequest(request);
```

6 Set the message's content type.

```
messageMsg.setContent(pc, pc.getContentType());
```

7 Update the message headers.

```
pc.setHeaders(messageMsg);
```

8 Send the message.

```
Transport.send(messageMsg);
```

Toolkit procedures — receiving messages

This set of procedures demonstrates how to use the Toolkit to receive S/MIME messages.

Connecting to the mail server

1 Create a `User` object to represent the recipient of the message.

```
com.entrust.toolkit.User recipient = new User();
```

2 Retrieve the recipient's Profile.

```
FileInputStream fis =
    new FileInputStream(<path to recipient's Profile>);
```

3 Instantiate a credential reader (in this case a `StreamProfileReader`) to read the Profile.

```
StreamProfileReader spf = new StreamProfileReader(fis);
```

4 Log in.

```
recipient.login(spf, <user's password>);
```

5 Retrieve the recipient's decryption private key.

```
java.security.PrivateKey privateKey = recipient.getDecryptionKey();
```

6 Retrieve recipient, recipient's mail box password, mail server, protocol, and mail box information.

```
String recip =
    properties.getString(<section name>, <key name for recipient>);
String mbPassword =
    properties.getString(<section name>, <key name for mailbox password>);
String host =
    properties.getString(<section name>, <key name for mailhost>);
String protocol =
    properties.getString(<section name>, <key name for protocol>);
String mbox =
    properties.getString(<section name>, <key name for mailbox>);
```

Note

This code fragment assumes you are using an `ini` file, `properties`, as described in the section entitled **Handling S/MIME messages**.

7 Create a `Properties` object and get the default session.

```
java.util.Properties props = System.getProperties();
javax.mail.Session session = Session.getDefaultInstance(props, null);
```

8 Create a `Store` object to represent a message store — for storing and retrieving messages.

```
javax.mail.Store store = null;
if (url != null) {
    URLName urln = new URLName(url);
    store = session.getStore(urln);
    store.connect();
}
else if (protocol != null)
{
    store = session.getStore(protocol);
}
else
{
    store = session.getStore();
}
```

Note

`url` refers to the message store.

9 Connect to the mail server.

```
if (host != null || recip != null || recipPassword != null)
{
    store.connect(host, recip, recipPassword);
}
else
{
    store.connect();
}
```

Retrieving messages from the mail box

1 Open the recipient's mail box.

```
javax.mail.Folder folder = store.getDefaultFolder();
folder = folder.getFolder(mbox);
folder.open(Folder.READ_ONLY);
```

Note

You can open a folder in either the `READ_ONLY` or the `READ_WRITE` mode.

2 Retrieve messages from the mailbox.

```
javax.mail.Message[] msgs = folder.getMessages();
```

3 Create a `FetchProfile` object to customize message attributes.

```
javax.mail.FetchProfile fp = new FetchProfile();
fp.add(FetchProfile.Item.ENVELOPE);
fp.add(FetchProfile.Item.FLAGS);
```

Note

For detailed information about the `FetchProfile` class and the `FetchProfile.Items` (ENVELOPE, FLAGS) you can specify, refer to the Javadoc reference documentation for the JavaMail API.

4 For each `message` in the mail box, read the envelope information.

```
javax.mail.Address[] address;
address = message.getFrom());
address = message.getRecipients(Message.RecipientType.TO))

for (int j = 0; j < address.length; j++)
{
    System.out.println("FROM: " + address[j].toString());
    System.out.println("TO: " + address[j].toString());
}

System.out.println("SUBJECT: " + message.getSubject());
Date date = message.getSentDate();
System.out.println("SendDate: "+ date.toString());
System.out.println("Size: " + message.getSize());
```

5 Decrypt messages that have encrypted content.

```
iaik.security.smime.EncryptedContent ec = (EncryptedContent)message;
ec.decryptSymmetricKey(privateKey, 0);
```

Note

`privateKey` is the recipient's decryption private key. Refer to step 5 in the procedure entitled **Connecting to the mail server**.

6 Display certificates in a certificates only message.

```
iaik.security.smime.SignedContent sc = (SignedContent)message;

if (sc.getSMimeType().equals("certs-only")) {
    System.out.println("This message contains certificates only.");
    X509Certificate[] certs = sc.getCertificates();
    for (int i = 0; i < certs.length; ++i) {
        System.out.println(certs[i].toString(true));
    }
```

```
}
```

7 Verify the S/MIME signed content and display the signer's certificate.

```
iaik.x509.X509Certificate signer = null;
signer = sc.verify();
System.out.println("This message is signed by: " +
     signer.getSubjectDN());
System.out.println(signer.toString(true));
```

8 Validate the signer's certificate.

```
recipient.validate(signer);
```

Determining the message's content type and displaying the message

1 Determine the message's content type.

```
if (message instanceof javax.mail.Part) {
    System.out.println("CONTENT-TYPE: " +
        ((Part)message).getContentType());
    message = ((Part)message).getContent();
}
```

2 Display the content of a certificate request.

```
System.out.println("This message contains a certificate request.");
iaik.security.smime.PKCS10Content pkcs10 = (PKCS10Content)message;
iaik.pkcs.pkcs10.CertificateRequest request = pkcs10.getCertRequest();
System.out.println(request.toString());
```

3 Display `String` content.

```
System.out.println("Content is a String");
System.out.println((String)message);
```

4 Display multipart content.

```
System.out.println("Content is a multipart message");
javax.mail.Multipart mp = (Multipart)message;
int count = mp.getCount();
for (int i = 0; i < count; i++) {
    System.out.println("Multipart: " + (i+1));
    //  Code to display message body part.
}
```

5 Close the `Folder` and the `Store` opened in the procedures entitled **Connecting to the mail server** (step 8) and **Retrieving messages from the mail box** (step 1).

```
folder.close(false);
store.close();
```

Note

The `Folder.close` method takes a boolean argument. If the argument is set to `true`, all messages in the mailbox that are flagged for deletion are deleted. If the argument is `false`, the flagged messages are untouched, but remain flagged for deletion.

S/MIME version 3

Secure multipurpose Internet mail extensions (S/MIME) version 3 is built upon, and includes, the Cryptographic Message Syntax (CMS) — RFC 2630 — which describes a standard syntax for protecting electronic messages and data. Applications that integrate the capabilities of S/MIME version 3 therefore, can send and receive secure MIME content without concern for the transport protocols that underlie the data transfer, as long as they can convey MIME data.

S/MIME version 3 is fully defined in the following specifications:

- RFC 2630 — the Cryptographic Message Syntax (http://www.imc.org/rfc2630)
- RFC 2632 — S/MIME version 3 Certificate Handling (http://www.imc.org/rfc2632)
- RFC 2633 — S/MIME version 3 Message Specification (http://www.imc.org/rfc2633)

The Toolkit's support for S/MIME version 3 allows you to develop applications that provide electronic messages with cryptographic services such as authentication, message integrity, non-repudiation, and privacy. Such messages must be created according to to the S/MIME version 3 Message Specification (RFC 2633) and must conform to the Cryptographic Message Syntax. The CMS specification defines the syntax for multiple content types (data structures) and S/MIME version 3 uses three of these:

- Data — used within the SignedData and EnvelopedData types
- SignedData — provides integrity, authentication, and non-repudiation using digital signatures
- EnvelopedData — provides privacy by encrypting message content

As required by the CMS, the Toolkit supports the use of both the DSA and RSA algorithms for the verification of digital signatures in S/MIME version 3 (S/MIME version 2 had to use only the RSA algorithm for this purpose). An EnvelopedData content type comprises encrypted content and at least one content-encryption key to form a digital envelope. Content-encryption keys are encrypted for each recipient of a message using one of the following three general techniques described in the CMS specification:

- Key transport — encrypts the content-encryption key with the recipient's public key (this is similar to S/MIME version 2)
- Key agreement — encrypts the content-encryption key using a shared secret (a symmetric key generated using the sender's private key and the recipient's public key)

  Note

  Key agreement, and in particular the Ephemeral-Static variant of the Diffie-Hellman key agreement algorithm, is not supported in Release 6.1 of the Toolkit.

- Symmetric key-encryption keys — encrypts the content-encryption key with a

symmetric key that has already been exchanged on a secure channel

Toolkit classes

The Toolkit's S/MIME version 3 capabilities are divided among the following jar files:

- `entcms.jar`
- `entsmimev3.jar`

Note

To develop applications using the S/MIME version 3 capabilities offered by the Toolkit, you require a J2SDK development environment.

The Toolkit contains classes that represent the S/MIME version 3 content types — Data, SignedData, and EnvelopedData — as well as classes that represent the signer and recipient of a message. These classes are in the `entcms.jar` file in the `etjava\lib` folder.

This subsection introduces the main classes in the Toolkit's `iaik.cms` and `iaik.smime` packages. Refer to the Javadoc reference documentation for more detailed information about the packages and classes mentioned here.

Classes in iaik.cms

Data and DataStream

The `Data` and `DataStream` classes represent the standard and stream implementations respectively of the CMS Data content type.

Note

If an application is handling large amounts of data, use the `iaik.cms.DataStream` class.

SignedData and SignedDataStream

The `SignedData` and `SignedDataStream` classes represent the standard and stream representations of the CMS SignedData content type, which provides the syntax for creating digital signatures. If there are no signers for the content, you can use SignedData content type to distribute certificates and certificate revocation lists.

EnvelopedData and EnvelopedDataStream

The `EnvelopedData` and `EnvelopedDataStream` classes represent the standard and stream implementations of the CMS EnvelopedData content type, which provides a syntax for building digital envelopes. Using this class, you can create a digital envelope for content of any type and for any number of recipients simultaneously.

SignerInfo

The `SignerInfo` class represents the CMS SignerInfo type. Using this class, you can gather the information required to create a digital signature for the content of a CMS SignedData object.

RecipientInfo

The `RecipientInfo` class represents the CMS RecipientInfo type. With this class, you can collect information about a particular recipient in a CMS EnvelopedData object.

Classes in iaik.smime

SignedContent

Use the `SignedContent` class in combination with classes in the `javax.mail` package to create signed S/MIME messages. The CMS defines two formats for signed S/MIME messages:

- `application/pkcs7-mime` with SignedData — created by processing the prepared MIME entity to be signed into a CMS object of type SignedData, and then inserted it into an `application/pkcs7-mime` MIME entity.
- `multipart/signed` — a clear signing format. Recipients who do not have an S/MIME application can read messages signed using this format.

EncryptedContent

Use the `EncryptedContent` class in combination with classes in the `javax.mail` package to create S/MIME encrypted (enveloped) messages.

The S/MIME version 3 Message Specification stipulates the `application/pkcs7-mime` type for enveloping data. To create this format, the prepared MIME entity to be enveloped is processed to create a CMS object of type EnvelopedData, which is then inserted into an `application/pkcs7-mime` MIME entity.

Dependencies

The Toolkit's `etjava\lib` folder includes two mailcap files, `mailcap` and `mailcap.default`. These files are consulted by email applications to determine how to display messages that have an unfamiliar content type. Copy these files into the `lib` folder of your J2SDK installation.

Note

You can also specify a path to the `mailcap` and `mailcap.default` files using the `MailcapCommandMap(java.lang.String fileName)` constructor in the `javax.activation` package. Refer to the Javadoc reference documentation (http://java.sun.com/products/javabeans/glasgow/javadocs/index.html) for the JavaBeans™ Activation Framework (http://java.sun.com/products/javabeans/glasgow/jaf.html).

The Toolkit's support for S/MIME version 3 relies upon two jar files from Sun Microsystems:

- `mail.jar` (http://java.sun.com/products/javamail/) — contains the JavaMail™ API, which provides classes that form the framework of Java mail and messaging applications.

- `activation.jar` (http://java.sun.com/products/javabeans/glasgow/jaf.html) — contains the JavaBeans Activation Framework, which provides services to interrogate arbitrary data to determine its type, to provide access to it, to find out the operations available on it, and to create the software component to implement those operations.

The packages in `mail.jar` (`javax.mail`) and `activation.jar` (`javax.activation`) are standard extensions to the Java 2 Platform. You can place them directly in the `lib\ext` folder of the Java Runtime Environment (which corresponds to the `jre\lib\ext` folder in J2SDK directory structure) to create installed extensions, or you can specify them in the classpath option when you compile and run your S/MIME messaging applications.

Sample applications

The `etjava\example\smimev3` folder contains sample applications that demonstrate how to use the Toolkit's API to create applications that support S/MIME version 3. The examples in this subsection are intended to illustrate the important steps required in applications that create signed and encrypted S/MIME version 3 messages — they are not complete procedures.

Creating a signed message

Complete the following steps to create a signed message using the Toolkit's's S/MIME version 3 API:

1 Create a `javax.mail.MimeMessage` object.

```
MimeMessage msg = createMessage(session);
```

where `createMessage` is a method in your application that returns a `MimeMessage` object given a `javax.mail.Session` object.

2 Specify whether to create an implicitly or explicitly signed message.

```
boolean implicit = true;
iaik.smime.SignedContent sc = new SignedContent(implicit);
```

Note

Implicitly signed messages can be read only with mail clients that support S/MIME (because the content is included in the S/MIME signature). Explicitly signed messages consist of multipart/signed content, where the first part contains human readable content and the second part contains the signature — clear signed messages.

3 Construct the message.

```
StringBuffer buf = new StringBuffer();
msg.setSubject("S/MIME implicitly signed message.");
buf.append("This message is implicitly signed.\n");
buf.append("You must have an S/MIME-aware client to read this message.");
```

4 Set the content of the `SignedContent` object.

```
sc.setText(buf.toString());
```

The `setText` method wraps the content in a `javax.activation.DataHandler`

object internally, specifying the MIME type as `text/plain`.

5 Add the signer's certificates to the signed content.

```
sc.setCertificates(signerCertificates);
```

6 Add one signer to the signed content.

```
sc.addSigner(signerPrivateKey, signerCertificate);
```

7 Set the message's content, specifying its MIME type.

```
msg.setContent(sc, sc.getContetntType());
```

8 Send the message.

```
javax.mail.Transport.send(msg);
```

Creating an encrypted message

Complete the following steps to create an encrypted message using the Toolkit's S/MIME version 3 API:

1 Create a `javax.mail.MimeMessage` object.

```
MimeMessage msg = createMessage(session);
```

where `createMessage` is a method in your application that returns a `MimeMessage` object given a `javax.mail.Session` object.

2 Construct the content to be encrypted.

```
iaik.smime.EncryptedContent ec = new EncryptedContent();

StringBuffer buf = new StringBuffer();
msg.setSubject("S/MIME encrypted message.");
buf.append("This is encrypted content.\n");
buf.append("Content encryption algorithm is: " + algorithm.getName());
```

where `algorithm` is an instance of the `iaik.asn1.structures.AlgorithmID` class.

3 Set the content of the `EncryptedContent` object.

```
ec.setText(buf.toString());
```

Note

The `setText` method sets the given `String` as the `EncryptedObject`'s content and with a MIME type of `text/plain`.

4 Add one recipient with `rsaEncyrption` as the key transport algorithm.

```
ec.addRecipient(recipientCertificate, AlgorithmID.rsaEncryption);
```

5 Add the signer as a recipient and specify the encryption algorithm.

```
ec.addRecipient(encryptionCertOfSigner, AlgorithmID.rsaEncryption);
ec.setEncryptionAlgorithm(algorithm, keyLength);
```

6 Set the message's encrypted content, specifying its MIME type.

```
msg.setContent(ec, ec.getContetntType());
```

7 Update message headers.

```
ec.setHeaders(msg);
```

8 Send the message.

```
javax.mail.Transport.send(msg);
```

Chapter 6

XML Encryption

This section describes the Toolkit's support for XML encryption as described in the XML Encryption Syntax and Processing candidate recommendation of 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-core-20020802/).

Topics in this section:

- Introduction to XML encryption
- XML encrypted data structure
- Encrypting and decrypting XML

Introduction to XML encryption

The Toolkit's implementation of XML encryption is based upon the proposals under review by the World Wide Web Consortium's (W3C) XML Encryption Working Group (http://www.w3.org/Encryption/2001/) and specifically on the XML Encryption Syntax and Processing candidate recommendation (http://www.w3.org/TR/2002/CR-xmlenc-core-20020802/). With the implementation of XML encryption in Release 6.1 of the Toolkit, you can encrypt and decrypt entire XML documents, fragments of XML documents that are complete and well-formed elements, and arbitrary binary data.

XML encryption is important when there is a need to encrypt only selected parts of XML documents, leaving the rest of the document in plain view. This capability integrates well with XML digital signatures, which allow for the signing of a specific section, or fragment, within an XML document.

Refer to the section entitled **Recommended reading** in the **Introduction**, for a list of related Web addresses and W3C Recommendations covering XML related technologies.

XML encrypted data structure

The **XML Encryption Syntax and Processing** candidate recommendation refers to the need to encapsulate all the information required to handle encrypted data efficiently. Such information includes details about the encryption method, and a reference to the key used to encrypt the data.

Encrypting XML elements results in the replacement of those elements with `<EncryptedData>` elements. This section briefly describes the structure of an `<EncryptedData>` element within an XML document. Refer to sections 3 to 7 of the **XML Encryption Syntax and Processing** candidate recommendation for the definitive description of XML encryption syntax, processing rules, and algorithms.

An XML document contains encrypted data within one or more `<EncryptedData>` elements, which, together with most of the element's descendents, belongs to the XML encryption namespace — `xmlns="http://www.w3c.org/2001/04/xmlenc#"`. An exception is `<KeyInfo>`, an optional child element of `<EncryptedData>`, from the XML digital signatures namespace, `xmlns="http://www.w3.org/2000/09/xmldigsig#"`. Other elements, such as the node list within the `<EncryptionMethod>` element, can belong to any namespace. The XML Encryption Syntax and Processing candidate recommendation discusses the interaction of XML encryption and XML digital signatures applied to XML fragments in the same document.

The following XML code fragment illustrates the basic structure of an `<EncryptedData>` element.

```
<?xml version="1.0"?>

<xenc:EncryptedData Id="ED0"
                    Type="http://www/w3.org/2002/04/xmlenc#Element"
                    xmlns:xenc="http://www.w3c.org/2001/04/xmlenc#">

   <xenc:EncryptionMethod xenc:Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc">
      <IV>JYZ86ne83lU4obBxANCFYA==</IV>
```

```
   </xenc:EncryptionMethod>

   <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
      <xenc:EncryptedKey Id="EK0"
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
         <xenc:EncryptionMethod
               xenc:Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p" />
         <xenc:ReferenceList>
            <xenc:DataReference URI="#ED0" />
         </xenc:ReferenceList>
         <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
            <dsig:KeyName>cn=user A,o=Autobots,c=CA</dsig:KeyName>
            <dsig:X509Data>
               <dsig:X509IssuerSerial>
                  <dsig:X509IssuerName>o=Autobots,c=CA</dsig:X509IssuerName>
                  <dsig:X509SerialNumber>989942410</dsig:X509SerialNumber>
               </dsig:X509IssuerSerial>
            </dsig:X509Data>
         </dsig:KeyInfo>
         <xenc:CipherData>
            <xenc:CipherValue>AcBhYvM+KC4dSz6cYpVtXC93ju0Ex0xBa/aE1tWA=</xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedKey>
      <xenc:RetrievalMethod Type="" URI="#EK0" />
   </dsig:KeyInfo>

   <xenc:CipherData>
      <xenc:CipherValue>YNTlQEqTtBOYz8OODPyy/9Zhb78bSBj4wg9yHFYqkUCNanFVfmdzV4w1bgv
       FCYNh A+I1wzmD0EppajILKAiUKoTOUx/nPblmGfoh53rt3jAaqzR+qMJbQNanFVfmdzV4w1bgvR
       YNTlQEqTtBOYz8OODPyy/9Zhb78bSBj4wg9yHFYqkUCNanFVfmdzV4w1bgvNanFVfmdzV4w1bgvb
       FCYNh A+I1wzmD0EppajILKAiUKoTOUx/nPblmGfoh53rt3jAaqzR+qMJbQNanFVfmdzV4w1bgv=
      </xenc:CipherValue>
   </xenc:CipherData>

</xenc:EncryptedData>
```

An XML document can contain any number of distinct <EncryptedData> elements. <EncryptedData> elements cannot be nested within one another. The Toolkit's implementation of XML encryption incorporates one or more <EncryptedKey> elements for each recipient of the encrypted data. The <EncryptedKey> elements, which in turn contain key, reference, and encryption method information, are siblings of each other. <EncryptedKey> elements can be children of the <EncryptedData> element, but they can also be located elsewhere in an encrypted XML document as a child of a <dsig:KeyInfo> element. When you use the Toolkit's implementation of XML encryption, <EncryptedKey> elements are always children of <KeyInfo> elements.

<EncryptedData>

The <EncryptedData> element is the central element in the XML encryption syntax. In an XML document that represents the result of encrypting XML elements or content in accordance with the **XML Encryption Syntax and Processing** specification, the <EncryptedData> element distinguishes those sections of the document that contain encrypted information. Encrypting binary data also results in an <EncryptedData> element, which is returned to the application to be inserted into another XML document or to be used as the root element in a new document. Refer to section 4.1 of the **XML Encryption Syntax and Processing** for more information.

An <EncryptedData> element can act as the document root of a self-contained XML

document. Its<CipherData> child element contains either the encrypted data as a base64-encoded octet sequence in a child <CipherValue> element, or a <CipherReference> element. A <CipherReference> element identifies an external source, with a universal resource indicator (URI) reference, and the transforms required to retrieve the encrypted data as a base64-encoded octet sequence.

The <EncryptedData> element can contain an empty<EncryptionMethod> element whose attributes describe the encryption algorithm used, and tell applications how to decrypt the cipher data. Refer to the XML Encryption schema in the the **XML Syntax and Processing** candidate recommendation for a detailed description of the content model of the <EncryptedData> element.

The Type attribute specifies the type of data (as a URI) contained in an <EncryptedData> element. Section 3.1 of the **XML Encryption Syntax and Processing** candidate recommendation advises the use of the Type attribute to allow the decryption process to recover an encrypted XML document in its original form. The Javadoc reference documentation for the com.entrust.toolkit.xencrypt.init.XMLEConstants class specifies the URIs for encrypted data of types element and content. Use these URIs as values for the Type attribute.

<EncryptedKey>

An <EncryptedKey> element is a means of transporting encryption keys between the encrypting and decrypting entities of a transaction. It can be a self-contained XML document, or it can appear within an <EncryptedData> element as a child of the <KeyInfo> element. The element contains a base64-encoded encrypted symmetric key in the <CipherValue> element under its mandatory <CipherData> element. If an <EncryptedKey> element is not inside a <KeyInfo> element, it can contain references to data and keys (using <DataReference> and <KeyReference> elements) that have been encrypted using its symmetric key.

The key in the <EncryptedKey> element is always encrypted for the recipient, or recipients, of the data. Decrypting the contents of <EncryptedKey> makes the key immediately available to the <EncryptionMethod> element.

<KeyInfo>

The <KeyInfo> element, an optional child element of the <EncryptedData> or <EncryptedKey> elements, belongs to the XML digital signatures namespace, xmlns="http://www.w3.org/2000/09/xmldigsig#". It can identify the symmetric key, and it can identify recipients of the encrypted data (the entities for whom the data has been encrypted), in which case it contains information related to the recipients' public encryption keys. Keying material need not accompany an encrypted XML document, but the recipient of the encrypted data must have access, by some other means, to the keys and algorithms used for encryption to be able to decrypt the data.

The <KeyName> element is a child of KeyInfo and contains a string value that communicates a key identifier to the recipient of the data. The content of the KeyName element is typically an identifier that is related to the key pair used to encrypt the symmetric key, but the element can contain other protocol-related information that

indirectly identifies a key pair. Common uses of the <KeyName> element include simple string names for keys, a key index, a distinguished name (DN), or an email address.

<EncryptionMethod>

The <EncryptionMethod> element is an optional child element of both the <EncryptedData> and <EncryptedKey> elements, and identifies the encryption algorithm applied to the data in the <CipherValue> element. The recipient must have access, by some other means, to the encryption algorithm used to encrypt the data, if the <EncryptionMethod> element is not used.

Algorithms

The Toolkit supports the algorithms required by XML Encryption Syntax and Processing candidate recommendation. They are listed in the following table.

XML encryption algorithms

Block encryption algorithms	**Identifying URI**
AES with 128-bit key in CBC mode with PKCS #5 padding	http://www.w3.org/2001/04/xmlenc#aes128-cbc
AES with 192-bit key in CBC mode with PKCS #5 padding	http://www.w3.org/2001/04/xmlenc#aes192-cbc
AES with 256-bit key in CBC mode with PKCS #5 padding	http://www.w3.org/2001/04/xmlenc#aes256-cbc
Triple DES encrypt-decrypt-encrypt (EDE) in CBC mode	http://www.w3.org/2001/04/xmlenc#tripledes-cbc
Key transport algorithms	**Identifying URI**
RSA using PKCS #1 v1.5 algorithm identifier	http://www.w3.org/2001/04/xmlenc#rsa-1_5
RSA using PKCS #1 version 2.0 (OAEP) padding	http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
Symmetric Key wrap algorithms	**Identifying URI**
AES with 128-bit key	http://www.w3.org/2001/04/xmlenc#kw-aes128
AES with 192-bit key	http://www.w3.org/2001/04/xmlenc#kw-aes192
AES with 256-bit key	http://www.w3.org/2001/04/xmlenc#kw-aes-256
Triple DES	http://www.w3.org/2001/04/xmlenc#kw-tripledes
Message digest algorithms	**Identifying URI**
SHA1	http://www.w3.org/2001/04/xmlenc#sha1
Encoding algorithms	**Identifying URI**
Base64	http://www.w3.org/2001/04/xmlenc#base64

Note

Refer to the `com.entrust.toolkit.xencrypt.init.XMLEConstants` class in the Javadoc reference documentation for information about the algorithms for XML encryption supported by the Toolkit.

<CipherData>

The `<CipherData>` element is a mandatory child of the `<EncryptedData>` element or of the `<EncryptedKey>` element. This element has one of two child elements:

- `<CipherValue>` — containing actual encrypted data as a base64-encoded octet sequence.
- `<CipherReference>` — provides a reference to an external location containing data to be encrypted as a base64-encoded octet sequence.

Encrypting and decrypting XML

The `init.properties` file, and other associated properties files, contain various attributes and mappings required by the Toolkit's XML libraries. The following extract from the `init.properties` file shows its various sections and their properties. The file contains extensive comments and instructions on how to edit the properties for your particular computing environment.

The properties in section 1 specify ISO language and country codes used to select the appropriate exception message resource bundle.

```
# Initialization properties for XML Encryption and IXSIL
#
#=====================================================================
# Section 1: Properties for localizing exception messages
#=====================================================================

IXSILException.ISOLanguageCode = "en"
IXSILException.ISOCountryCode  = "US"
```

The two properties in section 2 contain URIs specifying the location of the IXSIL algorithm properties and `keyManager` properties files.

```
#=====================================================================
# Section 2: Edit this section as required to reflect property file
#            and schema locations
#=====================================================================

location.algorithmsProperties = file:/C:../init/properties/algorithms.properties
location.keyManagerProperties = file:/C:../init/properties/keyManager.properties

DOMUtils.SignatureSchema = file:/C:../init/schemas/Signature.xsd

#---------------------------------------------------------------------
# OPTIONAL: This property contains a URI specifying the (virtual)
# location of the IXSIL init properties file (i. e. this file). The
# URI MUST be absolute.

#location.initProperties = file:/C:/etjava/examples/xml/init/properties/init.properties
```

Section 3 contains properties that allow you to specify the XML namespace prefixes you want to use for XML Signature and XML Schema.

```
#====================================================================
# Section 3: NameSpace prefixes for XML signature and XML schema
#====================================================================

namespacePrefix.XMLSignature = dsig:
namespacePrefix.XMLSchemaInstance = xsi:
```

Section 4 contains properties that you do not need to edit.

```
#====================================================================
# Section 4: Nothing in this section needs to be edited.
#====================================================================

# DOMUtils properties

 ... contents not shown ...

#--------------------------------------------------------------------
# AlgorithmFactory properties

... contents not shown ...

#--------------------------------------------------------------------
# VerifierKeyManager properties

... contents not shown ...

#--------------------------------------------------------------------
# XPathUtils properties

... contents not shown ...

#--------------------------------------------------------------------
# CanonicalizationUtils properties

... contents not shown ...

#--------------------------------------------------------------------
# Key Wrap AlgorithmFactory properties

... contents not shown ...

#--------------------------------------------------------------------
# Key Wrap algorithm namespace identifiers

... contents not shown ...

#--------------------------------------------------------------------
# Encryption AlgorithmFactory properties

... contents not shown ...

#--------------------------------------------------------------------
# Encryption algorithm namespace identifiers

... contents not shown ...
```

Refer to the **Readme** file for the XML encryption sample application in the `etjava\examples\xml\encrypt` folder for information about editing the `init.properties` file.

The Toolkit provides high-level classes in four packages for use when working with XML encryption. The packages are distributed in the `entxml.jar` file.

com.entrust.toolkit.xencrypt.core

This package contains the `Encryptor` and `Decryptor` classes used to parse XML documents and to create Document Object Model (DOM) trees. The nodes of the DOM trees represent the XML elements in the documents you are encrypting and decrypting. The most frequently used methods in these classes allow you to perform the following operations:

- Initialization of the Toolkit
- Selection of users and recipients
- Selection of particular elements in the document you want to encrypt
- Selection of encryption algorithms
- Encryption and decryption of XML documents

You can use the other classes in the package to encrypt DOM elements for a group of recipients (`EncryptedElementSet`) and for working with the Toolkit's reference implementation of transaction counting (`XMLEncCounter`).

com.entrust.toolkit.xencrypt.init

This package contains two classes:

- `XMLEInit` — used to initialize the Toolkit in preparation for performing XML encryption and XML decryption operations
- `XMLEConstants` — containing constants, such as algorithm identifiers, used by the Toolkit for XML encryption

com.entrust.toolkit.xencrypt.algorithms

This package contains classes that define the behaviour of the encryption algorithms used by the Toolkit's XML encryption classes.

com.entrust.toolkit.xencrypt.exceptions

This package contains classes that handle exceptions thrown by the Toolkit's XML encryption classes.

The samples folder, `etjava/examples/xml`, and its subfolders, contain sample applications that demonstrate how to use the Toolkit's XML encryption capabilities to encrypt and decrypt information for one or more recipients. From the **StartHere** document, click on the links to the documents entitled, **Running the sample applications** and **Using XML Digital Signatures and XML Encryption** for information about how to run the sample applications.

Toolkit procedure — encrypting XML fragments

The following steps summarize how to encrypt a fragment of an XML document for a single recipient:

- Log in to the Toolkit
- Retrieve recipient's public key certificate

- Retrieve properties file (`init.properties`)
- Initialize IXSIL — the package containing classes that provide the Toolkit's XML signature handling capabilities
- Initialize XMLE — the classes that provide the the Toolkit's XML encryption capabilities
- Encrypt the specified XML fragments

To encrypt XML fragments within an XML document:

1 Instantiate a `User` and log in.

Note

Refer to the procedure entitled **Identifying a user** in the **User and credentials management section**.

2 Retrieve the recipient's public key certificate.

```
X509Certificate certificate =
    new X509Certificate(new FileInputStream(<path to recipient's certificate>));
```

3 Retrieve the initialization file (`init.properties`) and initialize the IXSIL library using one of the `init()` methods of the `iaik.ixsil.init.IXSILInit` class.

```
URI initProps = new URI(<URL of an initialization properties file>);
IXSILInit.init(initProps);
```

Note

Refer to the XML samples **Readme** file (see `etjava\StartHere.html`, or go directly to `etjava\examples\xml\xml_readme.html`) for information about editing the `init.properties` file. For more information about the properties files, refer to the `iaik.ixsil.init.IXSILInit` class in the Javadoc reference documentation.

4 Initilaize the Toolkit to prepare for XML encryption.

```
com.entrust.toolkit.xencrypt.init.XMLEInit initializer =
    new XMLInit(new FileInputStream(<path to init.properties file>));
```

5 Create a `com.entrust.toolkit.xencrypt.core.Encryptor` instance.

```
Encryptor encryptor =
    new Encryptor(initializer, new FileInputStream(<path to document to encrypt>));
```

Note

The `Encryptor` constructor initializes the Toolkit and parses the document creating a DOM tree that includes the elements to be encrypted.

6 Initialize the `Encryptor` instance, created in the previous step, with a trust manager.

```
encryptor.setTrustmanager(sender);
```

Note

`sender` represents the user logged into the Toolkit in step 1.

The trust manager provides the means to validate the recipient's public key

certificate.

7 Set the symmetric encryption algorithm.

```
encryptor.setSymmetricAlgorithm(XMLEConstants.ALGORITHM_AES128);
```

Note

The `XMLEConstants` class is in the `com.entrust.toolkit.xencrypt.init` package.

Note

This line of code sets the 128-bit AES algorithm. To use the default algorithm (256-bit AES), omit this line from your code. Refer to the `com.entrust.toolkit.xencrypt.init.XMLEConstants` class in the Javadoc reference documentation for algorithm identifiers of other symmetric encryption algorithms you can use with the Toolkit.

8 Specify a value for the ID attribute of the `<EncryptedKey>` elements in the encrypted document.

```
encryptor.setEncryptedKeyBaseID("KB");
```

Note

The `String` KB is the base ID for the `<EncryptedKey>` elements in the encrypted document. `Encryptor` assigns values sequentially to the IDs beginning at 0. This means that the base ID for the first `<EncryptedKey>` element is KB0, for the second it is KB1, and so on up to KBn. Refer to the `com.entrust.toolkit.xencrypt.core.Encryptor.setEncryptedKeyBaseID` method in the Javadoc reference documentation for more detailed information.

If you omit this line from your code, The Toolkit sets the ID attribute to default values (EK0, EK1, ... , EKn).

9 Specify a value for the ID attribute of the `<EncryptedData>` elements in the encrypted document.

```
encryptor.setEncryptedDataBaseID("DB");
```

Note

The `String` DB is the base ID for the `<EncryptedData>` elements in the encrypted document. `Encryptor` assigns values sequentially to the IDs beginning at 0. This means that the base ID for the first `<EncryptedData>` element is `DB0`, for the second it is `DB1`, and so on up to `DBn`. Refer to the `com.entrust.toolkit.xencrypt.core.Encryptor.setEncryptedDataBaseID` method in the Javadoc reference documentation for more detailed information.

If you omit this line from your code, the Toolkit sets the ID attribute to default values (`ED0`, `ED1`, ... , `EDn`).

10 Specify the elements in the XML document that are to be encrypted.

```
org.w3c.dom.NodeList elements =
```

```
    encryptor.getDocument().getElementsByTagName(<a String representing the element
name>);

for(int i = 0; i < elements.getLength(); i++)
    {
        encryptor.setRecipient(element, certificate);
        encryptor.setContentOnly(element, true);
    }
```

Note

The `setContentOnly(org.w3c.dom.Element element, boolean contentOnly)` method uses the value of the `contentOnly` argument to determine whether or not to encrypt an entire DOM element (`contentOnly` set to `false`), including opening and closing tags, or to encrypt only the element content (`contentOnly` set to `true`).

As an alternative, the `com.entrust.toolkit.xencrypt.core.Encryptor` class has an `encryptContent(org.w3c.dom.Element element)` method that directly encrypts the contents of a DOM element.

```
boolean status =
    true ? encryptor.encryptContent(element) : encryptor.encryptElement(element);
```

Call the `updateElement(org.w3c.dom.Element element)` to replace the unencrypted element or content with an `<EncryptedData>` element.

11 Encrypt the elements.

```
encryptor.encrypt();
```

The `encrypt()` method replaces the element (or elements) to be encrypted with an `<EncryptedData>` element whose `Type attribute is set to Element ot to Content`.

Note

If you have not called the `setContentOnly` method (to specify whether an entire DOM element, including opening and closing tags, or just the content of a DOM element is to be encrypted) before this step, the `encrypt()` method will throw an exception.

12 Write the encrypted file.

```
encryptor.toOutputStream(new FileOutputStream(<path to encrypted file>));
```

The `toOutputStream(java.io.OutputStream outputStream)` method, inherited from the `com.entrust.toolkit.xencrypt.core.EncryptionHandler` class, serializes the DOM document to the specified output stream.

Toolkit procedure — decrypting XML fragments

The following steps summarize how to decrypt an encrypted XML document:

- Log in to the Toolkit
- Retrieve properties file for XMLE

- Initialize IXSIL — the package containing classes that provide the Toolkit's XML signature handling capabilities
- Initialize XMLE — the classes that provide the the Toolkit's XML encryption capabilities
- Specify the recipients for whom to decrypt information
- Decrypt the `<EncryptedData>` elements

To decrypt XML fragments within an XML document:

1 Instantiate a `User` and log in.

Note

Refer to the procedure entitled **Identifying a user** in the **User and credentials management section**.

2 Retrieve the initialization file (`init.properties`) and initialize the IXSIL library using one of the `init()` methods of the `iaik.ixsil.init.IXSILInit` class.

```
File initProps = new File(<path to init.properties file>);
IXSILInit.init(initProps);
```

Note

Refer to the XML samples **Readme** file (see `etjava\StartHere.html`, or go directly to `etjava\examples\xml\xml_readme.html`) for information about editing the `init.properties` file. For more information about the properties files, refer to the `iaik.ixsil.init.IXSILInit` class in the Javadoc reference documentation.

3 Initialize the Toolkit to prepare for XML decryption.

```
com.entrust.toolkit.xencrypt.init.XMLEInit initializer =
    new XMLInit(new FileInputStream(<path to init.properties file>));
```

4 Create a `com.entrust.toolkit.xencrypt.core.Decryptor` instance.

```
Decryptor decryptor =
    new Decryptor(initializer, new FileInputStream(<path to document to decrypt>));
```

Note

The `Decryptor` constructor requires an `XMLEInit` object and an `InputStream` object — the document you want to decrypt — as arguments. It initializes the Toolkit and parses the document creating a DOM tree that includes the elements to be decrypted.

5 Attach a logged in user (usually the same user instantiated in step 1) who can decrypt the document.

```
decryptor.addUser(recipient);
```

Note

To allow the `Decryptor` to decrypt those XML fragments encrypted for specific recipients, you must invoke the `addUser(User user)` method for each recipient.

6 Decrypt elements that have been encrypted for the user, or users, added in step 5 of this procedure.

```
decryptor.decrypt();
```

Note

The `decrypt()` method decrypts all the `<EncryptedData>` elements, in the encrypted document,that have been encrypted for the user, or users, added to the `Decryptor` instance in step 5 .

7 Write the decrypted file.

```
decryptor.toOutputStream(new FileOutputStream(<path to decrypted file>));
```

Note

The `toOutputStream(java.io.OutputStream outputStream)` method, inherited from the `com.entrust.toolkit.xencrypt.core.EncryptionHandler` class, serializes the DOM document to the specified output stream.

Toolkit procedure — encrypting arbitrary data using XML

The following steps summarize how to encrypt external binary data:

- Log in to the Toolkit.
- Retrieve the initialization properties for XMLE file.
- Initialize IXSIL — the package containing classes that provide the Toolkit's XML signature handling capabilities.

  Note

  The Toolkit invokes some of the utility methods in IXSIL.

- Initialize XMLE — the classes that provide the the Toolkit's XML encryption capabilities.
- Retrieve the recipient's encryption public certificate (and the sender's encryption public certificate if the application is to encrypt for the sender).
- Encrypt the binary data as an octet sequence.

In more detail, the procedure to encrypt binary data is as follows:

1 Instantiate users to encrypt (`sender`) and decrypt (`recipient`) the data and log into the Toolkit.

Note

Refer to the procedure entitled **Identifying a user** in the **User and credentials management section** for inforamtaion about how to perform this step

2 Retrieve the XMLE initialization properties file (`init.properties`).

```
iaik.ixsil.util.URI initProps = new URI(<URL of the init.properties file>);
```

3 Initialize the IXSIL library using one of the `init()` methods of the `iaik.ixsil.init.IXSILInit` class.

```
IXSILInit.init(initProps);
```

4 Initialize the Toolkit's XML encryption classes, XMLE.

```
iaik.ixsil.util.ExternalReferenceResolverImpl res =
    new ExternalReferenceResolverImpl(initProps);
com.entrust.toolkit.xencrypt.init.XMLEInit initializer =
    new XMLEInit(res.resolve(initProps));
```

5 Retrieve the recipient's and, if required, the sender's, encryption public certificate.

```
X509Certificate recipientCertificate = recipient.getEncryptionCertificate();
X509Certificate senderCertificate = sender.getEncryptionCertificate();
```

Note

In this procedure, `sender` represents the `User` who encrypts the data, and `recipient` represents the user who decrypts the data — both created in step 1.

6 Create an XML document to contain the encrypted data.

```
com.entrust.toolkit.xencrypt.core.Encryptor encryptor = new Encryptor(initializer);
```

Note

The `Encryptor` class encrypts Document Object Model (DOM) elements in an XML document.

7 Create an `EncryptedElementSet` object and add to it all the elements (XML DOM elements) to be encrypted.

```
com.entrust.toolkit.xencrypt.core.EncryptedElementSet set =
    new EncryptedElementSet(encryptor);
set.addElement(<URL of binary data to be encrypted>);
```

Note

The `EncryptedElementSet` class allows an application to define a set of DOM elements to be encrypted for a group of recipients. This procedure encrypts for a single recipient.

If you want to store the encrypted data remotely, you can specify a location using an absolute URI.

```
encryptor.setCipherURI(<URL of binary data to be encrypted>, <URL of encrypted binary
data);
```

8 Set a trust manager to validate certificates and add the recipient (and optionally the sender) to the set.

```
encryptor.setTrustmanager(sender);

set.addRecipient(recipientCertificate);
set.addRecipient(senderCertificate);
```

9 Encrypt the data.

```
encryptor.encrypt();
```

Once your application has called the `Encryptor.encrypt` method, the data has been encrypted and you can handle it at your discretion. For example, you can retrieve a single `<EncryptedData>` element:

```
org.w3c.dom.Element = encryptor.getEncryptedDataElement(<URL of binary data to be
encrypted>);
```

You can also choose to write the cipher text to a file :

```
encryptor.toOutputStream(new FileOutputStream(<path to encrypted XML file>));
```

If the encrypted data was written to a URL, you can retieve it and store it locally. For example:

```
FileOutputStream fos = new FileOutputStream(new File(<local file name>));
fos.write(encryptor.getCipherText(<URL of the encrypted binary data>));
fos.close();
```

Toolkit procedure — decrypting binary data using XML

The following steps summarize how to decrypt encrypted binary data:

- Log in to the Toolkit.
- Retrieve the initialization properties for XMLE file.
- Initialize IXSIL — the package containing classes that provide the Toolkit's XML signature handling capabilities.

  Note

  The Toolkit invokes some of the utility methods in IXSIL.

- Initialize XMLE — the classes that provide the the Toolkit's XML encryption capabilities.
- Decrypt the binary data.

In more detail, the procedure to decrypt encrypted binary data is as follows:

1 Perform steps 1 to 4 of the procedure entitiled **Encrypting arbitrary data using XML**

2 Instantiate a `com.entrust.toolkit.xencrypt.core.Decryptor`.

```
Decryptor decryptor =
    new Decryptor(initializer, new FileInputStream(<path to encrypted XML file>));
```

Note

The `Decryptor` class decrypts encrypted elements in an XML document.

3 Add a user to decrypt the data.

```
decryptor.addUser(recipient);
```

Note

`recipient` represents the user who decrypts the data — created in step 1.

4 Decrypt the data.

```
decryptor.decrypt();
```

Your application can handle the decrypted data at your discretion. For example, you can retrieve the decrypted octet sequence as a `byte` array and write it to local file:

```
byte[] decryptedByte = (byte[]) decryptor.getDecryptedBinary().elementAt(0);

FileOutputStream fos = new FileOutputStream(<path to local file>);
fos.write(<path to local file>);
fos.close();
```

Chapter 7

XML digital signatures

This section describes the Toolkit's support for the XML-encoded digital signature format as described in the XML-Signature Syntax and Processing Recommendation 12 February 2002 (http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/).

Topics in this section:

- Introduction to XML digital signatures
- Types of XML digital signatures
- Structure of an XML digital signature
- Creating an XML digital signature
- Verifying an XML digital signature
- Using the Decryption Transform for XML SIgnature

Introduction to XML digital signatures

Encryption and digital signatures are well-recognized means for ensuring the authenticity, integrity, and non-repudiation of data exchanged over computer networks. Many standards and protocols exist to govern and secure this interchange of information. With the increasing interest in the Extensible Markup Language (XML) as the basis for e-commerce over the World Wide Web, a number of proposals for representing digital signatures in XML have begun to appear.

The World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) have created a common XML-encoded digital signature format, which is now a W3C recommendation — XML-Signature Syntax and Processing (http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/). This section of the Programmer's Guide describes how you can use the Security Toolkit for Java to secure and sign XML-encoded data.

Refer to the section entitled **Recommended reading** in the **Introduction**, for a list of related Web addresses and W3C Recommendations covering XML related technologies.

Note

The Toolkit's support for XML digital signature operations now includes the following enhancements:

- Support for the recent Apache XML Project releases of the Xerces-J XML processor (version 1.4.4) and the Xalan-J XSLT processor (version 2.3.1)
- Support for the recent XML Signature Schema (refer to http://www.w3.org/TR/xmldsig-core/#sec-Schema)
- Faster enveloped signatures
- Support for operations involving crptographic tokens

Structure of an XML digital signature

This section outlines the basic structure of an XML digital signature. Sections 4 and 5 of the XML-Signature Syntax and Processing recommendation describe the XML digital signature syntax in detail.

XML digital signatures are represented by a `<Signature>` element and its contents. The following XML code fragment, without the attributes that are attached to some elements, illustrates the basic structure.

```
<Signature Id="001"
                  xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod> ... </CanonicalizationMethod>
    <SignatureMethod> ... </SignatureMethod>
    <Reference URI= ... >
      <DigestMethod> ... </DigestMethod>
      <DigestValue> ... </DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
  <KeyInfo>
```

```
    <KeyValue>
      <DSAKeyValue>
        <P> ... </P>
        <Q> ... </Q>
        <G> ... </G>
        <Y> ... </Y>
      </DSAKeyValue>
    <KeyValue>
    <X509Data>
      <X509SubjectName> ... </X509SubjectName>
      <X509Certificate> ... </X509Certificate>
    </X509Data>
  </KeyInfo>
  <Object>
    <SignatureProperties>
      <SignatureProperty Target="#001">
      ...
      </SignatureProperty>
    </SignatureProperties>
  </Object>
  <Object>
    <Manifest>
      <Reference>
      ...
      </Reference>
    </Manifest>
  </Object>
  <Object>
     ...
  </Object>
</Signature>
```

The <Signature> element is the root element of the XML digital signature. The <SignedInfo> element is a child of <Signature>, and contains the signed information, the canonicalization and signature algorithms, and one or more <Reference> elements that point to the resources signed by the XML digital signature.

The canonicalization algorithm, referred to by the <CanonicalizationMethod> element, is applied to the <SignedInfo> element during signing and verification to produce the element's canonical, or standard, form. There is no difference in the byte-by-byte comparison of the canonical form of two logically equivalent XML documents, or elements, so this step ensures that the signing application and the verifying application obtain the same hash value for the <SignedInfo> element. For example, the following XML elements are logically equivalent, but applying a hash function to them would not produce the same hash value. The elements differ in the order of their attributes.

```
<Reference URI="http://www.company.com/document.html"
           Id="ref001">
    ... Some code not shown ...
</Reference>
```

```
<Reference Id="ref001"
           URI="http://www.company.com/document.html">
    ... Some code not shown ...
</Reference>
```

To obtain the same hash value, the Toolkit applies the canonicalization algorithm to each of the above elements to produce their canonical form before applying a hash function. The Canonical XMLVersion 1.0 W3C Recommendation

(http://www.w3.org/TR/xml-c14n) describes XML canonicalization in detail.

The optional <SignatureProperty> elements are additional statements about the XML signature itself. If the <SignatureProperty> elements are referred to from within the <SignedInfo> element, they too are signed by the XML signature.

If your application has special requirements to handle referenced resources, you can include such references in an optional <Manifest> element. Your application controls how, or even whether, the references in a <Manifest> element are verified.

The <KeyInfo> element, containing public key management information (<KeyValue> and <X509Data> elements, for example) and any number of optional <Object> elements, which can contain any data, are siblings of the <SignedInfo> and <SignatureValue> elements. The <KeyInfo> element is specified in the XML-Signature Syntax and Processing recommendation as an optional element, but, to ensure interoperability, the Toolkit's implementation of XML digital signatures includes the <KeyInfo> element and its contents. The base64 encoded value of the <SignatureValue> element is the digital signature itself. If the signer wishes to bind the keying information to the signature, a <Reference> can easily identify and include the <KeyInfo> as part of the signature.

The complete <Signature> element, containing the <SignedInfo>, <SignatureValue> and <KeyInfo> elements, comprises the XML digital signature.

Types of XML digital signatures

An XML digital signature, designed primarily for use in XML Web transactions, signs digital content adding authentication (identification) and data-integrity to your data and allowing precise specification of the data to be signed. You can use an XML digital signature to add authentication to one or more data resources — using a uniform resource indicator (URI) to refer to each of them — and to associate them with a cryptographic key.

Although the XML digital signatures themselves are created using XML syntax, an XML digital signature is not limited to signing only XML documents. If the XML document is accompanied by other resources such as style sheets or graphics, you might need to include these secondary resources when digitally signing your XML document.

One of the most important features of XML digital signatures is the ability to sign one or more parts of an XML document. When a number of authors have contributed to the contents of a document, or as new content is being added, each section can be digitally signed by an individual author.

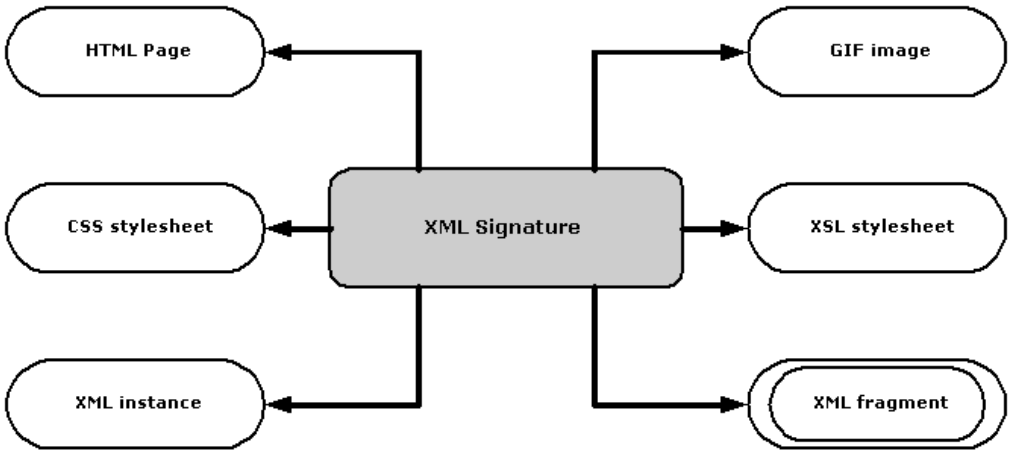
Such flexibility can be very important in situations where some parts of a document are digitally signed, and other parts are not. This gives the receiver of a document the ability to edit the unsigned portion of the document without invalidating any of the digitally signed information. For example, a signed XML document is delivered to a person for completion. If the entire XML document were signed, any change to the form's original content would invalidate the original signature.

A single XML Signature may encompass

- Character-encoded data (such as an HTML page).

- Binary-encoded data (such as a GIF image).
- XML-encoded data.
- A specific section, or XML fragment, within an XML file.

Resources



There are three possible types of XML signatures:

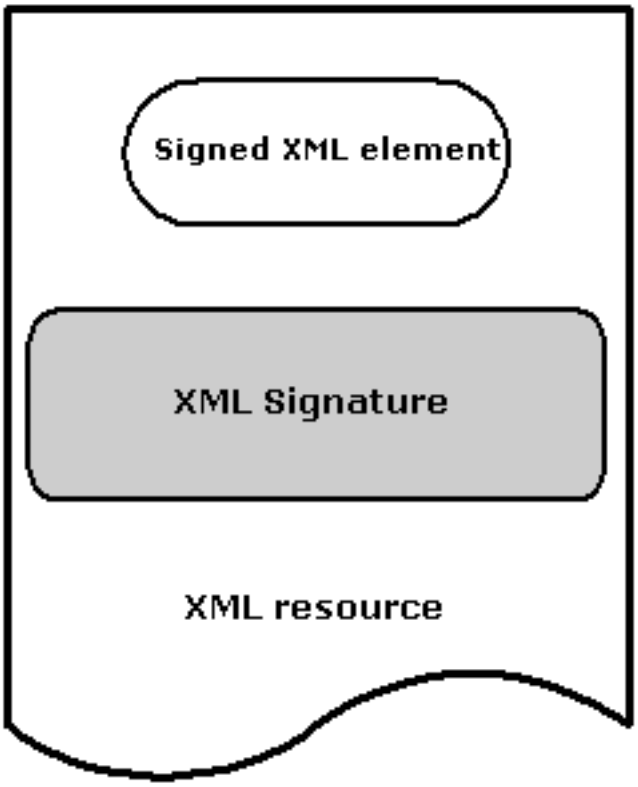
- Detached signatures
- Enveloping signatures
- Enveloped signatures

Detached XML signature

Detached digital signatures are signatures that can be separate from the signed resource, and that leave the resource in its original state. Detached XML digital signatures are commonly used to sign resources that are not XML documents. The resource and the signature are separate and both the resource and the digital signature must be transferred together.

An XML detached signature refers to, and signs, either a resource that is a sibling element within the same document as the signature, or a resource that is not part of the same XML document at all.

Detached XML signature



Practical example

A government agency posts a Request For Proposal (RFP) on their main Web site, and digitally signs the Web page containing the RFP, with a detached XML signature. The digital signature is made available for public download, which serves to confirm that the entity that created the Web page is the same entity that signed the page. Users can be confident that the Web page is trusted, and that the RFP has not been changed.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dsig:Signature Id="Signature001" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:SignedInfo>
      <dsig:CanonicalizationMethod Algorithm="http://www.w3.org/TR/2000/WD-
xml-c14n-20001011" />

      <dsig:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />

      <dsig:Reference URI="http://host/signedData.xml">
         <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />

         <dsig:DigestValue>TzxBrkGelVFeAU8JPqe6K07VTPU=</dsig:DigestValue>
      </dsig:Reference>
   </dsig:SignedInfo>

   <dsig:SignatureValue>OPU6zWU9l+36NjKHN9 .... NcBSufciuI=</dsig:SignatureValue>

   <dsig:KeyInfo>
```

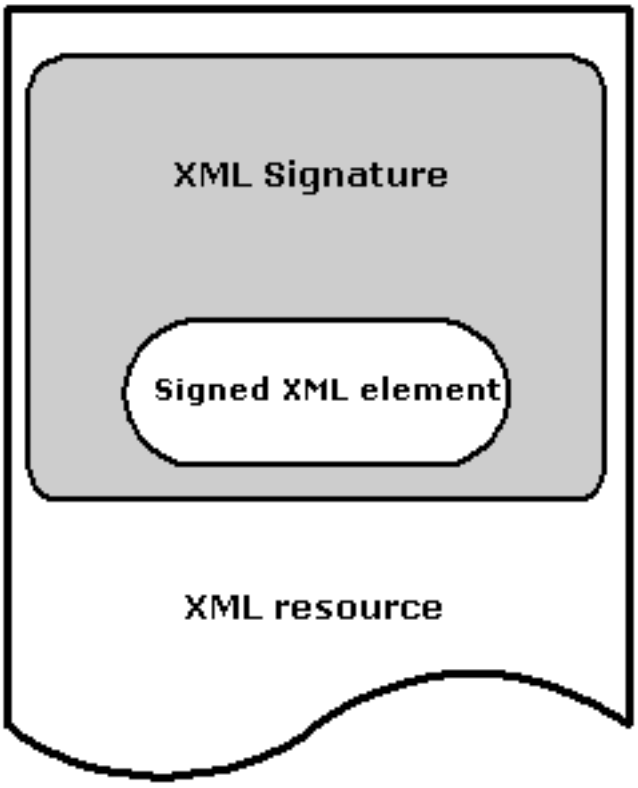
```
      <dsig:KeyValue>
         <dsig:RSAKeyValue>
            <dsig:Modulus>3PV+BoAm9hmXLkTSViVm ... Ysq2smyqgGok=</dsig:Modulus>

            <dsig:Exponent>AQAB</dsig:Exponent>
         </dsig:RSAKeyValue>
      </dsig:KeyValue>

      <dsig:X509Data>
         <dsig:X509SubjectName>cn=RFPsigner Test4,o=GovAgency,c=CA</dsig:X509SubjectName>

         <dsig:X509Certificate>MIIC1jCCAj+gAwI ... z3VRdkpDqjBNA==</dsig:X509Certificate>
      </dsig:X509Data>
   </dsig:KeyInfo>
</dsig:Signature>
```

Enveloping XML signature

XML enveloping signatures embed signed content into the XML `<Signature>` element. The main XML document contains the signature itself, and encloses the XML document.

Within an XML document, an XML enveloping signature refers to, and signs, a resource that is a child element of the XML signature.

Enveloping XML signature



Practical example

A person is being treated in hospital for a chronic illness. When the patient is released from the hospital, the doctor writes a medical report and uses an enveloping digital signature to sign the information. The patient returns to the hospital several weeks later and is treated by a different doctor. When the patient is released, the doctor adds new information to the patient's medical history and again uses an enveloping digital signature to sign only the new information. By using an enveloping digital signature, new information can be written at different times by different medical professionals, each signing only the medical records relevant to the time the patient was under their care.

```
<?xml version="1.0" encoding="UTF-8"?>
<dsig:Signature Id="Signature001" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:SignedInfo>
      <dsig:CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2000/WD-xml-c14n-20001011" />

      <dsig:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />

      <dsig:Reference URI="#Resource1">
         <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />

         <dsig:DigestValue>eFvCUOLuDDLjzxhNj5VKyikMHxY=</dsig:DigestValue>
      </dsig:Reference>
   </dsig:SignedInfo>

   <dsig:SignatureValue>xig/yjr0niDzhEH ... T3jo84oRvk=</dsig:SignatureValue>

   <dsig:KeyInfo>
      <dsig:KeyValue>
         <dsig:RSAKeyValue>
            <dsig:Modulus>3PV+BoAm9hmXLkTS ... Ysq2smyqgGok=</dsig:Modulus>

            <dsig:Exponent>AQAB</dsig:Exponent>
         </dsig:RSAKeyValue>
      </dsig:KeyValue>

      <dsig:X509Data>
         <dsig:X509SubjectName>cn=Henry Jekyll,c=UK</dsig:X509SubjectName>

         <dsig:X509Certificate>MIIC1jCCAj+gAwIBA ... 3VRdkpDqjBNA==</dsig:X509Certificate>
      </dsig:X509Data>
   </dsig:KeyInfo>

   <dsig:Object Id="Resource1">
      <document_element>
         <text>
            <child_of_text>Medical history report by Dr. Jekyll.</child_of_text>
         </text>
      </document_element>
   </dsig:Object>
</dsig:Signature>

<dsig:Signature Id="Signature002" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:SignedInfo>
      <dsig:CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2000/WD-xml-c14n-20001011" />

      <dsig:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />

      <dsig:Reference URI="#Resource1">
         <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
```

```
         <dsig:DigestValue>UwBLevgw378Jyt6noa+HNW6aD8A=</dsig:DigestValue>
      </dsig:Reference>
   </dsig:SignedInfo>

   <dsig:SignatureValue>hZ9a7YlH93Jr3 ... Rb6pNZfJMIU=</dsig:SignatureValue>

   <dsig:KeyInfo>
      <dsig:KeyValue>
         <dsig:RSAKeyValue>
            <dsig:Modulus>3PV+BoAm9hmXLk ... sq2smyqgGok=</dsig:Modulus>

            <dsig:Exponent>AQAB</dsig:Exponent>
         </dsig:RSAKeyValue>
      </dsig:KeyValue>

      <dsig:X509Data>
         <dsig:X509SubjectName>cn=Edward Hyde, c=UK</dsig:X509SubjectName>

         <dsig:X509Certificate>MIIC1jCCAj+gAwI ... gz3VRdkpDqjBNA==</dsig:X509Certificate>
      </dsig:X509Data>
   </dsig:KeyInfo>

   <dsig:Object Id="Resource1">
      <document_element>
         <text>
            <child_of_text>Medical history report by Dr. Hyde.</child_of_text>
         </text>
      </document_element>
   </dsig:Object>
</dsig:Signature>
```

Enveloped XML signature

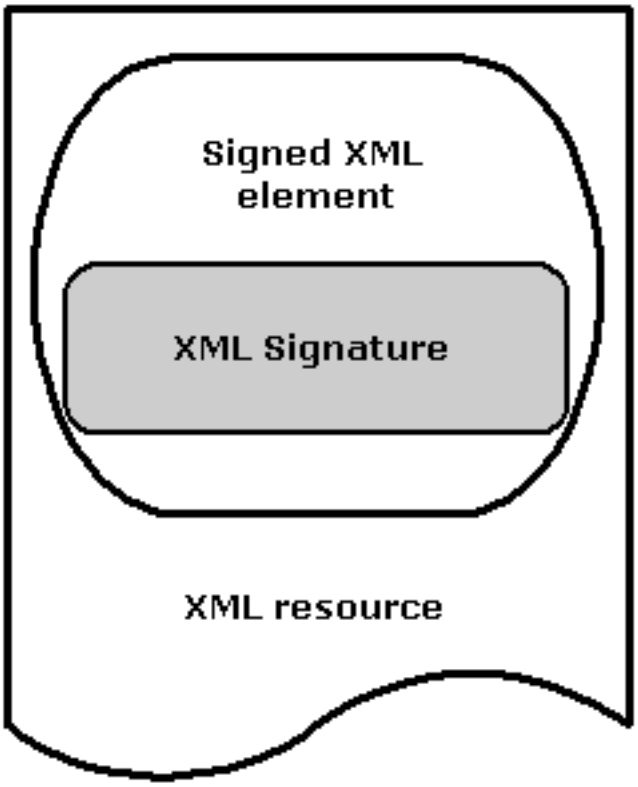
An XML enveloped signature is a digital signature that is embedded in an XML document, within the signed content. Enveloped digital signatures are commonly used when there are multiple resources within an XML document to be signed by one or more entities.

Note

You cannot use enveloped XML digital signatures to sign existing signatures.

Within an XML document, an XML enveloped signature refers to, and signs, a resource that contains the signature as an element.

Enveloped XML signature



Practical example

An employee is purchasing office supplies for their company and must complete an XML purchase order approval form. The form must be approved by the office manager and the finance department. The office manager must digitally sign the `<quantity>` element, which gives approval for the quantity of office supplies purchased. The finance department must digitally sign the `<creditApproval>` element, giving approval for the purchase of office supplies in a specific dollar amount.

```
<?xml version="1.0" encoding="UTF-8"?>
<RootElement>
... unsigned XML content ...
<quantity Id=Q1>40 boxes of printer paper
   <dsig:Signature Id="Signature001" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
      <dsig:SignedInfo>
         <dsig:CanonicalizationMethod
Algorithm="http://www.w3.org/TR/2000/WD-xml-c14n-20000907" />

         <dsig:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />

         <dsig:Reference URI=#Q1>
            <dsig:Transforms>
               <dsig:Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
            </dsig:Transforms>

            <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
```

```
            <dsig:DigestValue>Uwefn6Mkm8NjELWGQ7pNxqQsilk=</dsig:DigestValue>
         </dsig:Reference>
      </dsig:SignedInfo>

      <dsig:SignatureValue>B1FPUvo6GgjSLZ0 ... M+ebcNsNC8t1Ebng=</dsig:SignatureValue>

      <dsig:KeyInfo>
         <dsig:KeyValue>
            <dsig:RSAKeyValue>
               <dsig:Modulus>3PV+BoAm9hmX ... F11TfW04ocV9xmuN</dsig:Modulus>

               <dsig:Exponent>AQAB</dsig:Exponent>
            </dsig:RSAKeyValue>
         </dsig:KeyValue>

         <dsig:X509Data>
            <dsig:X509SubjectName>cn=OfficeManager o=ACME,c=CA</dsig:X509SubjectName>

            <dsig:X509Certificate>MIIC1jCCAj+gA ... QXV0b2JvdHMxF</dsig:X509Certificate>
         </dsig:X509Data>
      </dsig:KeyInfo>
   </dsig:Signature>
</quantity>
<creditApproval Id=A1>Finance department approves $1000.00 for purchase of required quantity.
   <dsig:Signature Id="Signature002" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
      <dsig:SignedInfo>
         <dsig:CanonicalizationMethod
Algorithm="http://www.w3.org/TR/2000/WD-xml-c14n-20000907" />

         <dsig:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />

         <dsig:Reference URI=#Q1>
            <dsig:Transforms>
               <dsig:Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
            </dsig:Transforms>

            <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />

            <dsig:DigestValue>Uwefn6Mkm8NjELWGQ7pNxqQsilk=</dsig:DigestValue>
         </dsig:Reference>
      </dsig:SignedInfo>

      <dsig:SignatureValue>B1FPUvo6GgjS ... M+ebcNsNC8t1Ebng=</dsig:SignatureValue>

      <dsig:KeyInfo>
         <dsig:KeyValue>
            <dsig:RSAKeyValue>
               <dsig:Modulus>3PV+BoAm9hmXLkTS ... F11TfW04ocV9xmuN</dsig:Modulus>

               <dsig:Exponent>AQAB</dsig:Exponent>
            </dsig:RSAKeyValue>
         </dsig:KeyValue>

         <dsig:X509Data>
            <dsig:X509SubjectName>cn=DepartmentofFinance
Test4,o=ACME,c=CA</dsig:X509SubjectName>

            <dsig:X509Certificate>MIIC1jCCAj+ ... QXV0b2JvdHMxF</dsig:X509Certificate>
         </dsig:X509Data>
      </dsig:KeyInfo>
   </dsig:Signature>
</creditApproval>

... unsigned XML content ...
```

```
</RootElement>
```

Creating an XML digital signature

Determine the resource, or resources, you want to sign. For example,

- http://www.entrust.com/index.html — refers to an HTML page on the Web.
- http://www.entrust.com/entrust_logo.gif — refers to a GIF image on the Web.
- http://www.entrust.com/xml/xml_doc.xml — refers to an XML document on the Web.
- http://www.entrust.com/xml/xml_doc.xml#sender1 — refers to a specific element in an XML document on the Web.
- #reference1 — refers to an element with the identifier `reference1` located in the same XML document as the XML signature. This element might be a child element of the signature's top element, an ancestor element of the signature element, or located on another branch of the XML instance.

The following list is an outline of the steps the Toolkit performs to create an XML digital signature:

- Calculate a digest of each resource, placing both the result and the digest method in a `<Reference>` element.
- Place the `<Reference>` elements inside a `<SignedInfo>` element.
- Calculate the digest of the `<SignedInfo>` element, sign the digest and put the signature value in a `<SignatureValue>` element.
- Place keying information in the `<KeyInfo>` element.
- Place the `<SignedInfo>`, `<SignatureValue>`, and `<KeyInfo>` elements into a `<Signature>` element. The `<Signature>` element is the XML Signature.

Toolkit procedure — creating a detached XML digital signature

To create a detached XML digital signature using the Toolkit, your application should perform the following steps:

1 Instantiate a `User` and log in.
Refer to the procedure entitled **Identifying a user** in the **User and credentials management section** of this guide, for further instructions on instantiating a user and logging in.

2 Initialize the IXSIL library using one of the `init()` methods of the `iaik.ixsil.init.IXSILInit` class.

```
URI initProps = new URI(<"URL of init.properties file">);
IXSILInit.init(initProps);
```

Note

The `init.properties` file, and other properties files, contain various

attributes and mappings required by the Toolkit's XML libraries, and set the kinds of key information that the Toolkit's libraries will process, or ignore, when verifying an XML signature. Specify the location of the `init.properties` file using a URI. You can use just the name of the file in the URI constructor, `URI initProps = new URI("init.properties");` for example, but in that case, the folder containing `init.properties` must be in the classpath.

Refer to the **Readme** file (`xml_sign_readme.html`) in the `etjava\examples\xml\sign` folder, and to the comments in the `init.properties` file for more information.

3 Create a `Signer` object to represent an XML document.

```
Signer signer = new Signer(<Absolute URI of the data to be signed>);
```

4 Create a `SignerSignature` object to represent the `<Signature>` element of the XML document and set the ID attribute of the `<Signature>` element.

a Create the `SignerSignature` object.

```
SignerSignature signature = signer.getSignature();
```

b Set the ID attribute of the `<Signature>` element.

```
signature.setId("Signature001");
```

Note

The `SignerSignature` interface (`iaik.ixsil.core.SignerSignature`) has a number of methods that allow you to specify certain characteristics of the `<Signature>` element. For example, accessor methods to handle the `<Signature>` element's ID attribute, and the `SignerSignature.sign()` method used to calculate the value of the XML signature's `<SignatureValue>` element.

5 Create an object to represent the `<SignedInfo>` element of the XML signature.

```
SignerSignedInfo signedInfo = signature.getSignerSignedInfo();
```

6 Specify the canonicalization algorithm to apply to the signature using the `iaik.ixsil.algorithms.CanonicalizationAlgorithmImplCanonicalXML` class.

a Create a new instance.

```
CanonicalizationAlgorithmImplCanonicalXML c14nAlg = new
CanonicalizationAlgorithmImplCanonicalXML();
```

b Set algorithm URI.

```
c14nAlg.setURI(new URI("http://www.w3.org/TR/2000/WD-xml-c14n-20000907"));
```

c Set canonicalization algorithm.

```
signedInfo.setCanonicalizationAlgorithm(c14nAlg);
```

7 Specify and set the signature algorithm using the

`iaik.ixsil.algorithms.SignatureAlgorithmImplDSA` and `iaik.ixsil.algorithms.SignatureAlgorithmImplRSA` classes, and the `iaik.ixsil.algorithms.SignatureAlgorithm` interface.

a Instantiate a signature algorithm of the appropriate kind, and set the corresponding URI.

```
SignatureAlgorithm signatureAlg = null ;
PrivateKey privatekey = m_user.getSigningKey();

if(privatekey.getAlgorithm().equals("DSA"))
{
  signatureAlg = new SignatureAlgorithmImplDSA();
  signatureAlg.setURI(new URI("http://www.w3.org/2000/09/xmldsig#dsa-sha1"));
}
else
{
  signatureAlg = new SignatureAlgorithmImplRSA();
  signatureAlg.setURI(new URI("http://www.w3.org/2000/09/xmldsig#rsa-sha1"));
}
```

b Set a private key.

```
signatureAlg.setSignerKey(privatekey);
```

c Set a signature algorithm.

```
signedInfo.setSignatureAlgorithm(signatureAlg);
```

8 Configure a reference to the resource you want to sign and add it to the `<SignedInfo>` element.

a Create a new reference.

```
SignerReference firstRef = signedInfo.createReference();
```

b Set reference URI.

```
firstRef.setURI(new URI(<internal or external reference to be signed>));
```

c Create a new instance of digest algorithm.

```
DigestAlgorithmImplSHA1 digestAlg1 = new DigestAlgorithmImplSHA1();
```

d Set digest algorithm URI.

```
digestAlg1.setURI(new URI("http://www.w3.org/2000/09/xmldsig#sha1"));
```

e Set digest algorithm.

```
firstRef.setDigestAlgorithm(digestAlg1);
```

f Add the resource reference to the signature.

```
signedInfo.addReference(firstRef);
```

Note

The `<SignedInfo>` element contains one or more references to the resources signed by the XML signature. This step configures and adds one reference to the XML signature. To add more references, repeat this step.

The `iaik.ixsil.core.SignerReference.setURI(URI uri)` method

points to the reference to be signed. This might be an external reference such as, `"file:///c:/test/tobesigned.txt"`, or an internal reference such as, `"#SignedProperty"`, or `"#Manifest"`.

9 Add a `<KeyInfo>` element that incorporates the keying information you want to include in the XML signature.

Note

The `<KeyInfo>` element specifies the particular keying information to include in an XML signature. The following steps show how to create a `<KeyInfo>` element that specifies the verification public key, the verification public certificate, and the signer's X .509 distinguished name (DN). In a real-world application, creating all three pieces of information within a `<KeyInfo>` element is unnecessary and increases the size of the XML digital signature slowing the verification operation. It is usually sufficient to include only the verification public certificate, which contains the signer's DN and verification public key, but you should determine which of the three pieces of keying information to include depending on the requirements of your PKI.

a Instantiate a key manager (`KeyManagerImpl`).

```
Document signatureDOMDoc = signer.toDocument();
    KeyManagerImpl keyManager = new KeyManagerImpl(signatureDOMDoc);
```

b Create and configure a KeyInfo provider (`KeyProviderImplX509Data`) for the X509Data clause.

```
KeyProviderImplX509Data keyProviderX509Data =
new KeyProviderImplX509Data(signatureDOMDoc);
```

c Include the user's verification certificate in the `<KeyInfo>` element.

```
X509Data x509 = new X509Data();

x509.insertHintAt(certificate, 0)  ;
```

d Include the user's distinguished name in the `<KeyInfo>` element.

```
X509SubjectName x509Name = new X509SubjectName((Name) certificate.getSubjectDN()) ;
x509.insertHintAt(x509Name, 0)  ;
keyProviderX509Data.insertX509DataAt(x509, 0);
```

e Create and configure a KeyInfo provider (`KeyProviderImplKeyValue`) for the KeyValue clause.

```
KeyProviderImplKeyValue keyProviderKeyValue = new
KeyProviderImplKeyValue(signatureDOMDoc);
    keyProviderKeyValue.setVerifierKey(certificate.getPublicKey());
```

f Add any (or all) of these KeyInfo providers.

```
keyManager.addKeyProvider(keyProviderKeyValue);
    keyManager.addKeyProvider(keyProviderX509Data);
```

g Set the signature's key manager.

```
signer.getSignature().setKeyManager( (SignerKeyManager) keyManager);
```

Note

The <KeyInfo> element is an optional child element of <Signature>.

10 Sign the <KeyInfo> element — optional.

a Create a reference for the <KeyInfo> element.

```
SignerReference ref = signedInfo.createReference();
```

b Identify the <KeyInfo> element reference with a URI.

```
String keyInfoId = keyManager.getId();
ref.setURI(new URI(null, null, null, null, keyInfoId));
```

c Create a new instance of the digest algorithm.

```
DigestAlgorithmImplSHA1 digestAlg1 = new DigestAlgorithmImplSHA1();
```

d Set the digest algorithm.

```
ref.setDigestAlgorithm(digestAlg1);
```

e Add the <KeyInfo> element reference to the resources to be signed.

```
signedInfo.addReference(ref);
```

Note

Your application should sign the <KeyInfo> element if the application supports the non-repudiation of XML digital signatures.

When you identify the the verification public key used to validate the XML digital signature in a <KeyValue> element, binding the <KeyInfo> element to the signature by signing it does not provide any more security than if the <KeyInfo> element were not signed.

The Toolkit's sample application source files use methods collected and defined in the Utils.java class in the etjava\examples\xml\utils folder. One of these methods, Utils.processTransforms(SignerSignature, boolean), used by all of the XML Digital Signature sample applications, demonstrates how to use the IAIK IXSIL API to ensure that you are not signing empty XML content.

11 Calculate the signature value.

```
signer.getSignature().sign();
```

12 Return the signed document to complete the procedure.

```
return signer.toDocument();
```

You can now handle the complete XML digital signature in a variety of ways. For example, you might want to write the signature to a file, or to send it to someone in an email message.

Remarks

To create an XML digital signature, use classes from the following Toolkit packages (found in entxml.jar) in your application:

- `com.entrust.toolkit`
- `iaik.ixsil.algorithms`
- `iaik.ixsil.core`
- `iaik.ixsil.init`
- `iaik.ixsil.keyinfo`
- `iaik.ixsil.util`
- `iaik.ixsil.exceptions`

Toolkit procedure — creating an enveloping XML digital signature

To create an enveloping XML digital signature using the Toolkit, your application should perform the following steps:

1 Follow steps 1 to 7 of the **Toolkit procedure — creating a detached XML digital signature**, and return to this procedure.

2 Configure references to the resources you want to sign and add them to the `<SignedInfo>` element.

a Create a reference to the resource.

```
SignerReference firstRef = signedInfo.createReference();
```

b Configure the reference for the enveloping digital signature - an internal reference to an XML digital signature Object that embeds the document.

```
URI baseURI = new URI(uriTobeSigned);
ExternalReferenceResolverImpl res = new ExternalReferenceResolverImpl(baseURI);
InputStream istrURI = res.resolve(baseURI);
```

c Retrieve a DOM representation of the document to be signed.

```
Document doc = DOMUtils.createDocumentFromXMLInstance(istrURI, baseURI,
DOMUtils.VALIDATION_NO_);
Element elemObject = doc.getDocumentElement();
```

Note

There is no base URI if your application reads the XML document directly from an `InputStream`. In this case, set the `baseURI` argument to `null`.

d Import the DOM representation into the document that has the `<Signature>` element as the root element - the document specified in the `Signer` instance.

```
DOMUtilsImpl utils = new DOMUtilsImpl();
utils.serializeElement( elemObject, new java.io.ByteArrayOutputStream() );
((DocumentImpl)signer.toDocument()).adoptNode(elemObject);
```

e Create a digital signature Object (`iaik.ixsil.core.Object`) that will contain the imported document.

```
iaik.ixsil.core.Object objectURI = signature.createObject( elemObject );
objectURI.setId( "Resource1");
```

```
signature.addObject( objectURI );
reference.setURI(new URI( "#Resource1") );
```

f Create a digest algorithm.

```
DigestAlgorithmImplSHA1 digestAlg1 = new DigestAlgorithmImplSHA1();
```

g Set digest algorithm URI.

```
digestAlg1.setURI(new URI("http://www.w3.org/2000/09/xmldsig#sha1"));
```

h Set digest algorithm.

```
firstRef.setDigestAlgorithm(digestAlg1);
```

i Add the resource reference to the signature.

```
signedInfo.addReference(firstRef);
```

Note

The <SignedInfo> element contains one or more references to the resources signed by the XML signature. This step configures and adds one reference to the XML signature. To add more references, repeat the step.

3 Follow steps 9, 10, and 11 of the **Toolkit procedure — creating a detached XML digital signature**, and return to this procedure.

You can now handle the complete XML digital signature in a variety of ways. For example, you might want to write the signature to a file, or to send it to someone in an email message.

Remarks

To create an XML digital signature, use classes from the following Toolkit packages (found in `entxml.jar`) in your application:

- `com.entrust.toolkit`
- `iaik.ixsil.algorithms`
- `iaik.ixsil.core`
- `iaik.ixsil.init`
- `iaik.ixsil.keyinfo`
- `iaik.ixsil.util`
- `iaik.ixsil.exceptions`

Toolkit procedure — creating an enveloped XML digital signature

To create an enveloped XML digital signature using the Toolkit, your application should perform the following steps:

1 Follow steps 1 and 2 of the **Toolkit procedure — creating a detached XML digital signature**, and return to this procedure.

2 Specify the XML element in your document that is to be the parent of the <Signature> element.

```
String insertSignaturePositionHere = <XML element name> ;
```

3 Create the XPath expression that specifies the position of the <Signature> element.

```
String xpath = null;

    xpath = "(//" + insertSignatureHere + ")[1]";

    Position signaturePosition = new Position(xpath, "", 0) ;

    URI baseURI = new URI(uriTobeSigned);
```

4 Create a Signer object, using the constructor that inserts a <Signature> element into an existing document.

```
Signer signer = new Signer(
     (new ExternalReferenceResolverImpl(baseURI)).resolve(baseURI),
        baseURI,
        signaturePosition);
```

Note

There is no base URI if your application reads the XML document directly from an InputStream. In this case, set the baseURI argument to null.

5 Follow steps 4 to 7 of the **Toolkit procedure — creating a detached XML digital signature**, and return to this procedure.

6 Configure references to the resources you want to sign and add them to the <SignedInfo> element.

a Create a reference.

```
SignerReference reference = signedInfo.createReference();
```

b Set the URI to null, so that the enveloped signature is over the entire XML document.

```
reference.setURI(new URI("") );
```

c Create an enveloped signature transform.

```
TransformImplEnvelopedSignature transform = new TransformImplEnvelopedSignature();
```

d Set the transform's algorithm URI.

```
transform.setURI(new URI("http://www.w3.org/2000/09/xmldsig#enveloped-signature"));
```

e Set the enveloped signature transform in the reference.

```
reference.insertTransformAt(transform, 0);
```

Note

This prevents the signing operation from signing the <Signature> element itself.

f Create a digest algorithm.

```
DigestAlgorithmImplSHA1 digestAlg1 = new DigestAlgorithmImplSHA1();
```

g Set the digest algorithm's URI.

```
digestAlg1.setURI(new URI("http://www.w3.org/2000/09/xmldsig#sha1"));
```

h Set the digest algorithm.

```
reference.setDigestAlgorithm(digestAlg1);
```

i Add the resource reference to the signature.

```
signedInfo.addReference(reference);
```

Notes

The `<SignedInfo>` element contains one or more references to the resources signed by the XML signature. This step configures and adds one reference to the XML signature. To add more references, repeat the step.

7 Follow steps 9, 10, and 11 of the **Toolkit procedure — creating a detached XML digital signature**, and return to this procedure.

You can now handle the complete XML digital signature in a variety of ways. For example, you might want to write the signature to a file, or to send it to someone in an email message.

Remarks

To create an XML digital signature, use classes from the following Toolkit packages (found in `entxml.jar`) in your application:

- `com.entrust.toolkit`
- `iaik.ixsil.algorithms`
- `iaik.ixsil.core`
- `iaik.ixsil.init`
- `iaik.ixsil.keyinfo`
- `iaik.ixsil.util`
- `iaik.ixsil.exceptions`

Handling the <SignatureProperties> and the <Manifest>elements

Within the `<Signature>`, you have the option to include statements about the XML signature itself. Collect the statements as `<SignedProperty>` child elements within a parent `<SignatureProperties>` element. By attaching an ID attribute to a `<SignatureProperty>`, you can refer to it in a `<Reference>` from within the `<SignedInfo>` element, making it one of the resources signed by the XML signature. Use the `iaik.ixsil.core.SignerSignatureProperites` interface and the `iaik.ixsil.core.SignatureProperty` class to accomplish this step. The following code fragment creates two signature properties.

1 Create a signature property container.

```
SignerSignatureProperties properties = signer.createSignatureProperties();
```

2 Create and add first signature property.

```
SignatureProperty firstProp =
    properties.createSignatureProperty("Content of  first (unsigned) signature
property");
firstProp.setTarget("#001");
properties.addSignatureProperty(firstProp);
```

3 Create and add second signature property.

```
SignatureProperty secondProp =
    properties.createSignatureProperty("Content of second (signed) signature property");
secondProp.setTarget("#001");
secondProp.setId("SignedProperty");
properties.addSignatureProperty(secondProp);
```

4 Create object container for the signature properties and incorporate it into the XML signature

```
Element propertiesDOMElem = properties.toElement();
iaik.ixsil.core.Object propertiesObject = signature.createObject(propertiesDOMElem);
signature.addObject(propertiesObject);
```

If you want your application to handle references in a `<Manifest>` element, use the `iaik.ixsil.core.SignerManifest` and `iaik.ixsil.core.SignerReference` interfaces. Create and configure the `<Reference>` elements and list them within the `<Manifest>` element.

1 Create an empty Manifest.

```
SignerManifest manifest = signer.createManifest();
manifest.setId("Manifest");
```

2 Create a reference.

```
SignerReference manifestRef = manifest.createReference();
```

3 Set reference URI.

```
manifestRef.setURI(new URI(<reference>));
```

4 Create a digest algorithm.

```
DigestAlgorithmImplSHA1 manifestDigestAlg = new DigestAlgorithmImplSHA1();
```

5 Set digest algorithm URI.

```
manifestDigestAlg.setURI(new URI("http://www.w3.org/2000/09/xmldsig#sha1"));
```

6 Set digest algorithm.

```
manifestRef.setDigestAlgorithm(manifestDigestAlg);
```

7 Add a reference to the manifest.

```
manifest.addReference(manifestRef);
```

8 Create an object container for the manifest and incorporate it into the signature.

```
Element manifestDOMElem = manifest.toElement();
iaik.ixsil.core.Object manifestObject = signature.createObject(manifestDOMElem);
signature.addObject(manifestObject);
```

Note

The `iaik.ixsil.core.SignerReference.setURI(URI uri)` method points to the reference.

Verifying an XML digital signature

To verify an XML detached, enveloping, and enveloped digital signature, incorporate the following steps into your application:

- Verify the signature of the `<SignedInfo>` element. To do this, re-calculate the digest of the `<SignedInfo>` element and use the public verification key to verify that the value of the `<SignatureValue>` element is correct for the digest of the `<SignedInfo>` element.
- If the last step passes, re-calculate the digests of the references contained within the `<SignedInfo>` element and compare them to the digest values expressed in each `<Reference>` element's corresponding `<DigestValue>` element.

Toolkit procedure — verifying an XML digital signature

To verify an XML digital signature stored as an XML instance your application should use the Toolkit to perform the following steps:

1 Initialize a `User` and log in.
Refer to the procedure entitled **Identifying a user** in the **User and credentials management section** of this guide, for further instructions on instantiating a user and logging in.

2 Initialize the IXSIL library using one of the `init()` methods of the `iaik.ixsil.init.IXSILInit` class.

```
URI initProps = new URI(<"URL of init.properties file">);
IXSILInit.init(initProps);
```

Note

The `init.properties` file, and other properties files, contain various attributes and mappings required by the Toolkit's XML libraries, and set the kinds of key information that the Toolkit's libraries will process, or ignore, when verifying an XML signature. Specify the location of the `init.properties` file using a URI. You can use just the name of the file in the URI constructor, `URI initProps = new URI("init.properties");` for example, but in that case, the folder containing `init.properties` must be in the classpath.

Refer to the **Readme** file (`xml_sign_readme.html`) in the `etjava\examples\xml\sign` folder, and to the comments in the `init.properties` file for more information.

3 Identify the location of the XML digital signature.

a Specify a URI that points to the XML document containing the XML signature.

```
URI baseURI = new URI(<URL of resource containing XML signature>);
```

For example,

The URL can be in any one of the following formats:

```
"file:/c:/etjava/examples/xml/signedData.xml"
"file:signedData.xml"
"http://host/signedData.xml"
```

b Open an input stream to the XML document containing the signature.

```
ExternalReferenceResolverImpl res = new ExternalReferenceResolverImpl(baseURI);
InputStream istream = res.resolve(baseURI);
```

Note

Optionally, you can choose to retrieve the signed document by resolving a URI. If your application reads the XML document directly from an `InputStream`, set the `baseURI` argument in the `Verifier` constructor (step 5) to `null`.

4 Specify where in the XML document to find the `<Signature>` element using an XPath expression.

```
String signatureSelector = "<XPath expression>";
```

Note

The Toolkit's XML digital signature samples (in `etjava\examples\xml`) create an ID attribute (Signature001) for the `<Signature>` element. The XPath expression that finds the `<Signature>` element with this ID is:

```
String signatureSelector = "//*[@ID=\"Signature001\"]";
```

5 Create an instance of `iaik.ixsil.core.Verifier`.

```
Verifier verifier = new Verifier(istream, baseURI, signatureSelector, null);
```

Note

There is no base URI if your application reads the XML document directly from an `InputStream`. In this case, set the `baseURI` argument to `null`.

6 Set a trust manager (`com.entrust.toolkit.TrustManager`) to validate the X.509 certificates contained in the `<dsig:X509Data>` elements, if these are part of the signature.

```
boolean verifiedByCertificate = false;

    X509TrustManagerInterface trustManager = new Trustmanager(user);

    KeyProviderInterface[] providers =
((KeyManagerImpl)verifier.getSignature().getKeyManager()).getKeyProviders();
    for(int i=0; i<providers.length ; i++)
    {
        if (providers[i] instanceof KeyProviderImplX509Data)
        {
```

```
verifiedByCertificate = true;
            ((KeyProviderImplX509Data)providers[i].setTrustManager(trustManager);
        }
    }
```

7 Validate the XML signature.

```
VerifierSignature signature = verifier.getSignature();
signature.verify();
```

Note

If this does not throw an exception, the signature has been verified, but the verification public key might have come from an explicit <KeyValue> element in the XML signature rather than from a valid certificate. This would not detect an attack in which a person modified the signed XML document, and replaced the <KeyValue>, and <SignatureValue> elements with a public key, and signature key that they have created.

To ensure that the verification public key comes from a valid certificate, disable the use (by IXSIL) of information contained in the <KeyValue> element, by removing the <KeyValue> subelement (`Subelement.02`) from the `keyManager.properties` file in the `examples\xml\init\properties` folder. If you initialize IXSIL properties from a URL rather than from the local file system, ensure that the channel to the URL is secure. This should prevent attacks that could otherwise modify the properties files before IXSIL initializes.

If you choose to leave the <KeyValue> subelement in place in the `keyManager.properties` file, you can use the `iaik.ixsil.keyinfo.VerifierKeyManager.getKeyProvider` method, which returns a reference to the key provider that provided the verification public key, or `null`, if there is no such key provider.

Two methods in `Utils.java` (in the `examples\xml\utils` folder), `validatePublicKey` and `isValidKeyValueInfo` demonstrate how to use the `getKeyProvider` method for determining whether a verification public key came from a valid certificate, and whether or not it is a trusted key.

Remarks

To verify an XML digital signature, use classes from the following Toolkit packages (found in `entxml.jar`) in your application:

- `com.entrust.toolkit`
- `iaik.ixsil.core`
- `iaik.ixsil.init`
- `iaik.ixsil.keyinfo`
- `iaik.ixsil.keyinfo.x509`
- `iaik.ixsil.util`
- `iaik.ixsil.exceptions`

Using the Decryption Transform for XML Signature

The `com.entrust.toolkit.xencrypt.core.TransformImplDecryption` class in the Security Toolkit for Java is an implementation of the Decryption Transform for XML Signature (http://www.w3.org/TR/2002/CR-xmlenc-decrypt-20020802). The Decryption Transform for XML Signature specifies the order of encryption and signature operations within a signed XML document so that applications can differentiate between those XML elements that were encrypted before being signed, and those that were encrypted after being signed.

Signed content in an XML document that is subsequently encrypted, must be decrypted before the signature can be verified. Inserting the W3C decryption transform into an XML digital signature identifies those portions of the data that had been encrypted before the signature was applied. The verification application can then inspect the `<Signature>` element, that contains the decryption transform, to find out which elements it must decrypt, and those it must not, before verifying the signature. For example, a time-stamp service might sign any XML documents it receives, including XML documents that have encrypted content.

Note

The term **transform** is used here in its mathematical sense to mean the result of a transformation.

Decryption transform structure

An XML digital signature, without the attributes that are attached to some elements, that contains a decryption transform has the following structure. Refer to the the Decryption Transform for XML Signature (http://www.w3.org/TR/2002/CR-xmlenc-decrypt-20020802) for more detailed information.

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod ... />
    <SignatureMethod ... />
    <Reference URI= ... >
      <Transforms>
        <dsig:Transform Algorithm="http://www.w3.org/2001/04/decrypt#">
          <dcrpt:Except URI="#ED0" xmlns:dcrpt="http://www.w3.org/2001/04/decrypt#"/>
        </dsig:Transform>
      </Transforms>
      <DigestMethod ... />
      <DigestValue ... />
    </Reference>
  </SignedInfo>
  <SignatureValue ... />
  <KeyInfo>
    <X509Data>
      <X509SubjectName ... />
    </X509Data>
    <X509Data>
      <X509Certificate ... />
    </X509Data>
```

```
  </KeyInfo>
</Signature>
```

The XML Signature Syntax and Processing specification (http://www.w3.org/TR/xmldsig-core/) describes the `<dsig:Transforms>` element. It is an optional child of the `<dsig:Reference>` element, and belongs to the XML digital signature namespace, `xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"`. The `<dsig:Transforms>` element contains an ordered list of `<dsig:Transform>` elements, the first of which takes as input, the dereferenced URI attribute of the `<dsig:Reference>` element. The input of each subsequent `<Transform>` element is the output of its predecessor. One such transform is the **Decryption Transform for XML Signature**. The output of the last `<Transform>` element in the list is the input to the `<dsig:DigestMethod>` element.

The decryption transform has one or more child elements in the decrypt namespace (`xmlns:dcrpt="http://www.w3.org/2001/04/decrypt#"`) called `<dcrpt:Except>`, whose required `URI` attribute is a reference to an `<enc:EncryptedData>` element within the same XML document as the signed content. `<EncryptedData>` elements referred to in this way are those that had been encrypted before the XML digital signature was applied. Verification applications should inspect XML digital signatures for transforms having `<dcrpt:Except>` elements as children, so that they can decrypt all `<enc:EncryptedDate>` elements except those referred to by the URI attribute in `<dcrpt:Except>` elements.

The decryption operation can expose additional `<EncryptedData>` elements, which the verifier must decrypt, again omitting those elements specified in the decryption transform. The Toolkit repeats this procedure until the XML content is restored to the form it had when the signatures was applied. The Toolkit does this internally whenever it verifies an XML digital signature.

Example

The following image shows a simple XML document containing elements that Alice encrypts for herself and for Bob, the recipient of the document. Alice signs the XML content of the `<SignThis>` element, excluding the content of the child element, `<DoNotSignThis>`. She then encrypts an element `<ForBob>`, which is inside the signed content, and finally wraps the element for Bob in another encryption. The image identifies the following elements:

- Alice encrypts two elements for herself
  - `ED0` — an element (`<ForAlice>`) inside the content she will subsequently sign
  - `ED1` — an element (`<ForAlice>`) in the `<DoNotSignThis>` element, outside the signed content
- Alice signs the content of the `<SignThis>` element, excluding the content of its child element, `<DoNotSignThis>`
- Alice encrypts for Bob an element inside the signed content — `ED2`, which Bob must decrypt before he can verify Alice's signature

- Alice wraps the encrypted and signed element <ForBob> in another encryption — ED3

XML document showing elements encrypted by Alice

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <!--Decryption Transform Sample-->
   <Header>This public header will be neither signed nor encrypted.</Header>
   <SignThis>
      <x>
         <!--The following statement is clear signed-->
         <AliceTestimonial>I like to combine XML digital signature and XML
         encryption !</AliceTestimonial>
      </x>
      <Super>
         <x>
            <!--The following secret will be signed, then encrypted, so the
            verifier decrypts it.-->
            <ForBob>Here is a secret that Alice signs, then encrypts
            for Bob.</ForBob>                                        ED2
         </x>
      </Super>                                                       ED3
      <x>
         <!--The following secret will be encrypted, then signed, so the
          verifier does not decrypt it.-->
         <ForAlice>Here is a secret that Alice encrypts for herself,
         then signs.</ForAlice>                                      ED0
      </x>
      <DoNotSignThis>
         <x>
            <!--The following secret is not part of the signed content, so
            the verifier does not decrypt it.-->
            <ForAlice>Here is a secret that Alice encrypts for
            herself.</ForAlice>                                      ED1
         </x>
      </DoNotSignThis>
      <PutSignatureHere/>
   </SignThis>
</Root>
```

When it signs XML content that might be encrypted later, your application should insert a decryption transform into the <Signature> element it creates. XML elements encrypted after being signed must be decrypted so that verification applications can validate the signature. Those elements encrypted before the signature was created must be left intact by the decryption and signature verification process.

The receiving application that decrypts and verifies Alice's message to Bob, refers to the decryption transform inside the <Signature> element of the signed and encrypted document, to determine which elements it must decrypt before it can verify Alice's signature. In this example, the application must proceed as follows:

- Decrypt Alice's last encryption step, the <Super> element — ED3
- Decrypt the element encrypted for Bob, <ForBob> — ED2
- Verify Alice's signature

The other encrypted elements in the document were either encrypted before Alice

applied her signature (`ED0`), or are outside the signed content (`ED1`) and require no action from the receiving application.

Inserting the decryption transform into an XML digital signature

Inserting a decryption transform into the XML digital signatures you create ensures that verification applications can validate the signature, even if all, or part, of the signed content is encrypted after it has been signed. The sample applications that accompany the Toolkit include a comprehensive example of how to work with the decryption transform. Refer to the `etjava\examples\xml\xml_readme.html` document for information and links to the **Readme** document for the decryption transform sample application.

The following two steps show how to use the `TransformImplDecryption` class to insert a decryption transform into the `<disig:Reference>` element of the XML digital signature when you are configuring the references to the resources you want to sign.

- Create a decryption transform.

```
com.entrust.toolkit.xencrypt.core.TransformImplDecryption decTransform =
    new TransformImplDecryption();
```

- Set the decryption transform for the reference you are configuring.

```
reference.insertTransformAt(decTransform, reference.getTransformNumber());
```

  where, `reference` is an instance of the `iaik.ixsil.core.SignerReference` class you created earlier in the process of creating the digital signature.

Note

The decryption transform should be the last transform listed in the `<Transforms>` element.

Chapter 8

SSL/TLS session-based activities

This section presents information about how to integrate the capabilities of the Secure Sockets Layer (SSL) protocol and the Transport Layer Security (TLS) protocol into your applications. It describes procedures for securing client and server applications using the latest IAIK implementation of the SSL/TLS protocols.

Topics in this section:

- Concepts
- Securing the client application with SSL/TLS
- Securing the server application with SSL/TLS
- Using tunneling to communicate with a PKI
- Communicating with Entrust Authority Roaming Server
- SSL/TLS security considerations

Concepts

The Toolkit incorporates the capabilities of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. TLS version 1.0 (http://www.ietf.org/rfc/rfc2246.txt?number=2246) is an updated version of SSL version 3.0 (http://home.netscape.com/eng/ssl3/ssl-toc.html). Using the SSL and TLS capabilities of the Toolkit, your application can provide privacy and the assurance of data integrity to the exchanges that take place when two applications communicate with each other over a network (especially the Internet) using such protocols as HTTP, FTP, and LDAP.

Although TLS version 1.0 and SSL version 3.0 do not work together, if one of the parties in a communication session does not support TLS, the TLS protocol provides a means to revert to the older SSL protocol. To maintain compatibility with older servers, the Toolkit also supports SSL version 2.0 on the client side. Refer to the subsection entitiled **SSL version 2.0** under **SSL/TLS security considerations**.

The Security Toolkit for Java uses the latest IAIK implementation of SSL/TLS and implements both client-side and server-side SSL sockets. You can use the Toolkit to connect to an existing SSL/TLS server, to write a server application that accepts SSL/TLS connections from browsers or other SSL/TLS enabled applications, and to implement both client-side and server-side communications. The Toolkit can authenticate connections, using public key technology, in both client-authenticated and server-authenticated modes allowing you to configure communications sessions that use unilateral authentication (where the client is effectively anonymous) or mutual authentication (where both client and server authenticate each other).

SSL/TLS contexts

The exchange of protected messages in an SSL/TLS session takes place within a security context. A security context, established by the communicating parties at the beginning of a secure communications session and discarded at the end of the session, is the information shared by the participants that accomplishes the following tasks:

- Authenticating the identity of the communicating parties at the beginning of the secure communications session and protecting that identity until the close of the session
- Protecting the integrity, authenticity (optional), and confidentiality (optional) of exchanged messages
- Authenticating the identity of a message's sender through the use of a digital signature
- Protecting the confidentiality of messages through encryption techniques

CipherSuites

A `CipherSuite` object defines a cipher specification supported in SSL and TLS. The `CipherSuite` class declares nearly fifty static `CipherSuite` member variables that you can specify in your application when setting the SSL/TLS context you are using. `CipherSuite` names have the following format:

```
SSL_asym_WITH_sym_mac
```

Where,

- `asym` is the asymmetric algorithm used for authentication and key exchange (RSA and DSA are specified for authentication — Diffie-Hellman and RSA are specified for key exchange)
- `sym` is the symmetric algorithm used for encryption
- `mac` is the hash function

For example, the `CipherSuite`, `SSL_RSA_WITH_3DES_EDE_CBC_SHA`, uses a 168 bit key to encrypt data with Triple DES in CBC mode. Each `CipherSuite` object includes the following information:

- Name of the cipher suite
- Exportability status (limit on the key size used for key exchange)
- Name of the key exchange algorithm
- Name of the cipher algorithm
- Name of the hash algorithm
- A 2 byte identifier for the cipher suite
- The number of bytes that are used for generating the write keys
- The number of bytes actually fed into the encryption algorithm
- IV size (CBC mode)

You can also define your own `CipherSuite`, using the `CipherSuite` constructor, with some restrictions. Refer to the Javadoc reference for more information.

Securing the client application with SSL/TLS

To secure a client-side application using SSL/TLS, your application should perform the following steps:

- Instantiate a user and log in.
- Create an `SSLClientContext` object.
- Enable strong ciphers.
- Optionally, set the client's credentials and the `ChainVerifier`.
- Create a socket to establish a connection.
- Create input and output streams for communication between client and server.
- Close the connection at the end of the communications session.

Toolkit procedure — securing a client application

The following procedure demonstrates how to secure a client application for client -

server communication using SSL.

1 Instantiate a `User`, set the connection to the Directory (if the user provides the IP address), and log in to the Toolkit. (See the section entitled **Identifying a user** for more details on logging in a user).

```
FileInputStream credentials =
    new FileInputStream (<location of user credentials>);
SecureStringBuffer password =
    new SecureStringBuffer(new String(<user's password>));

User user = new User();

if (<IP address> != null)
{
  JNDIDirectory dir = new JNDIDirectory (<ip address>, <port number>);
  user.setConnections(dir, null);
}

CredentialReader credReader = new StreamProfileReader(credentials);
user.login(credReader, password>);
```

2 Create an `SSLClientContext` object to define an SSL/TLS security policy for an `SSLTransport` object — which implements the secure SSL/TLS communications session.

```
SSLClientContext clientContext = new SSLClientContext();
```

Note

If your application creates the `SSLClientContext` before you call the `User.login` method, you must set the cryptographic service providers first. To do this, initialize the Toolkit before creating the `SSLClientContext` using the `com.entrust.toolkit.security.provider.Initializer.setProviders` method. For example:

```
Initializer.getInstance().setProviders(Initializer.MODE_NORMAL);
```

Refer to the subsections entitled **Cryptographic Service Providers** and **Initializing the Toolkit** in the **High level classes** section of the **Toolkit overview** chapter in this guide for more information.

3 Enable ciphers that use RSA or DSA keys.

```
// RSA
context.setEnabledCipherSuiteList(new CipherSuiteList(CipherSuite.CS_DHE_RSA));
// DSA
//context.setEnabledCipherSuiteList(new CipherSuiteList(CipherSuite.CS_DHE_DSS));
```

4 Optionally, set the user credentials for client authentication.

```
X509Certificate[] certChain=new X509Certificate[2];
certChain[0]=user.getVerificationCertificate();
certChain[1]=user.getCaCertificate();

clientContext.addClientCredentials(certChain, user.getSigningKey());
```

Note

For an SSL connection, the client's verification public certificate must be specified in the client credentials.

5 Optionally, set the certificate chain verifier.

```
EntrustChainVerifier ecv = new EntrustChainVerifier(user);
clientContext.setChainVerifier(ecv);
```

Note

The `TrustDecider` class has been deprecated and should no longer be used — refer to the `SSLContext` class in the Javadoc reference. A `null` argument turns chain verification off.

6 Create a socket to connect to the remote host and establish communications.

```
SSLSocket socket =
    new SSLSocket(<IP address>, <port number>, clientContext);
```

7 Optionally, call `SSLSocket.getActiveCipherSuite()` to check which cipher suite is being used.

```
CipherSuite suite = socket.getActiveCipherSuite();
```

Note

You must call `SSLSocket.getInputStream()` or `SSLSocket.getOutputStream()` before calling `SSLSocket.getActiveCipherSuite()` and querying the returned `CipherSuite` object through the accessor methods in the `CipherSuite` class.

8 Close the socket to terminate the communications session.

```
socket.close();
```

Remarks

When you are building applications or applets with SSL/TLS capabilities, ensure that you include the `entssl.jar` file in your operating system's `CLASSPATH` environment variable or in the `-classpath` option of the Javac compiler. This jar file is needed in addition to the `entbase.jar` and `entuser.jar` files that should be part of your classpath whenever you are working with the Toolkit.

Your application should import classes from the following Toolkit packages:

- `com.entrust.toolkit`
- `com.entrust.toolkit.credentials`
- `com.entrust.toolkit.util`
- `com.entrust.toolkit.x509.directory`
- `com.entrust.toolkit.exceptions`
- `iaik.x509`
- `iaik.security.ssl`
- `java.io`
- `java.net`

Toolkit procedure — using the JSSE API

The Toolkit provides an implementation of the Java Secure Sockets Extension (JSSE), which was an optional extension package to versions 1.2 and 1.3 of the J2SDK and is now part of version 1.4. The JSSE provider (supporting SSL version 3.0 and TLS version 1.0) in the Toolkit's API is `iaik.security.jsse.provider.IAIKJSSEProvider`. This class installs both the `IAIK_JSSE` security provider and the Sun Microsystems JSSE provider.

The procedure described in this subsection illustrates how to install the Toolkit's JSSE provider and how to create an SSL socket using the JSSE API. Perform the following steps to accomplish this task:

- Instantiate a user and log in.
- Initialize the Toolkit's JSSE provider.
- Create an `SSLContext` object.
- Create a `KeyManagerFactory` objec.t
- Create a `TrustManagerFactory` object.
- Initialize the SSL `Context` object using the key and trust managers.
- Create an SSL socket.

The following steps provide a more detailed discussion of the procedure:

1 Instantiate a `User` and log in. (See the section entitled **Identifying a user** for more details about logging in a user).

```
FileInputStream credentials =
    new FileInputStream (<location of user credentials>);
SecureStringBuffer password =
    new SecureStringBuffer(new String(<user's password>));

User user = new User();

if (<IP address> != null)
{
  JNDIDirectory dir = new JNDIDirectory (<ip address>, <port number>);
  user.setConnections(dir, null);
}

CredentialReader credReader = new StreamProfileReader(credentials);
user.login(credReader, password);
```

2 Initialize the Toolkit's JSSE provider.

```
java.security.Security.addProvider(new iaik.security.jsse.provider.IAIKJSSEProvider());
```

3 Create an `SSLContext` object that implements the TLS protocol from the Toolkit's provider.

```
SSLContext context = SSLContext.getInstance("TLS","IAIK_JSSE");
```

4 Generate an instance of the `KeyManagerFactory` class that implements the `SunX509` key management algorithm, and initialize it using an Entrust key store.

```
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
EntrustKeyStore eks = new EntrustKeyStore(user, true);
eks.load(null,null);
kmf.init(eks, pw.toCharArray());
```

5 Create and initialize a `TrustManagerFactory` object, and initialize it with the Entrust key store.

```
TrustManagerFactory tmf =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init(eks);
```

6 Initialize the `SSLContext` object using the key and trust managers.

```
context.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
```

Note

If you do not want to perform client authentication, set the `KeyManager[]` argument to `null` (`context.init(null, tmf.getTrustManagers(), null);`). The `TrustManager[]` argument must be present or the server certificate will be rejected.

7 Create an SSL socket.

```
SSLSocket socket = (SSLSocket) context.getSocketFactory().createSocket(<IP address of
host>, <port number>);
```

8 Use the socket to send and receive encrypted data as you would normally. This simple code fragment sends a message and writes the response to the standard output stream for the system.

```
PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream())));
out.println("GET / HTTP/1.0");
out.println();
out.flush();

InputStream is = socket.getInputStream();
int r = is.read();

while(r!=-1)
{
    System.out.print((char)r);
    r=is.read();
}
```

Depending on your appliciation, you will need to import classes from the following Toolkitand Java packages:

- `com.entrust.toolkit`
- `com.entrust.toolkit.credentials`
- `com.entrust.toolkit.util`
- `com.entrust.toolkit.x509.directory`
- `com.entrust.toolkit.exceptions`
- `iaik.security.jsse.provider`
- `java.io`
- `java.security`
- `javax.net`
- `javax.net.ssl`

Note

If you are using J2SDK version 1.2 or 1.3, the JSSE is an optional extension package. Its classes are contained in the following packages:

- `com.sun.net.ssl`
- `javax.net`
- `javax.net.ssl`
- `javax.security.cert`

For detailed information about the JSSE version 1.0.2, refer to the JSSE 1.0.2 API User's Guide (http://java.sun.com/products/jsse/doc/guide/API_users_guide.html).

If you are using version 1.4 of the J2SDK, the JSSE packages are part of the SDK. The JSSE API is contained in the following packages :

- javax.net
- javax.net.ssl
- javax.security.cert

For detailed information about the JSSE API in version 1.4 of the J2SDK, refer to the Sun Microsystems Web site (http://java.sun.com/products/jsse/index-14.html).

Securing the server application with SSL/TLS

To secure a server-side application using SSL/TLS, your application should perform the following steps:

- Instantiate a user and log in.
- Create an `SSLServerContext` object.
- Enable strong ciphers.
- Optionally, set the server's credentials and the `ChainVerifier`.
- Call `SSLServerContext.setRequestClientCertificate()`to specify whether or not your application requires client authentication.
- Create a socket to establish a connection.
- Create input and output streams for communication between client and server.
- Close the connection at the end of the communications session.

Toolkit procedure — securing a server application

The following procedure demonstrates how to secure a server application for client - server communication using SSL.

1 Instantiate a `User` and log in. (See the section entitled **Identifying a user** for more details about logging in a user).

```
FileInputStream credentials =
    new FileInputStream (<location of user credentials>);
SecureStringBuffer password =
    new SecureStringBuffer(new String(<user's password>));

User user = new User();

if (<IP address> != null)
{
  JNDIDirectory dir = new JNDIDirectory (<ip address>, <port number>);
  user.setConnections(dir, null);
}

CredentialReader credReader = new StreamProfileReader(credentials);
user.login(credReader, password);
```

2 Create an `SSLServerContext` to define an SSL/TLS security policy for an `SSLTransport` object. This implements the secure SSL/TLS communications session.

```
SSLServerContext serverContext = new SSLServerContext();
```

Note

If your application creates the `SSLServerContext` before you call the `User.login` method, you must set the cryptographic service providers first. To do this, initialize the Toolkit before creating the `SSLServerContext` using the `com.entrust.toolkit.security.provider.Initializer.setProviders` method. For example:

```
Initializer.getInstance().setProviders(Initializer.MODE_NORMAL);
```

Refer to the subsections entitled **Cryptographic Service Providers** and **Initializing the Toolkit** in the **High level classes** section of the **Toolkit overview** chapter in this guide for more information.

3 Enable strong ciphers that use RSA or DSA keys

```
// RSA
context.setEnabledCipherSuiteList(new CipherSuiteList(CipherSuite.CS_DHE_RSA));
// DSA
//context.setEnabledCipherSuiteList(new CipherSuiteList(CipherSuite.CS_DHE_DSS));
```

4 Optionally, set the user credentials for client authentication and call `SSLServerContext.setRequestClientCertificate()` to specify, with a Boolean value, whether or not your application should authenticate the client.

```
X509Certificate[] certchain = new X509Certificate[2];
certchain[0] = user.getEncryptionCertificate();
certchain[1] = user.getCaCertificate();
serverContext.addServerCredentials(certChain, user.getSigningKey());

serverContext.setRequestClientCertificate(true);
```

5 Optionally, set the certificate chain verifier.

```
EntrustChainVerifier ecv = new EntrustChainVerifier(user);
clientContext.setChainVerifier(ecv);
```

Note

The `TrustDecider` class has been deprecated and should no longer be used — refer to the `SSLContext` class in the Javadoc reference. A `null` argument turns chain verification off.

6 Create a server socket and accept the connection.

```
SSLServerSocket server =
    new SSLServerSocket(<port number>, serverContext);

SSLSocket socket = (SSLSocket)server.accept();
socket.startHandshake();
```

7 Optionally, call `SSLSocket.getActiveCipherSuite()` to check which cipher suite is being used.

```
CipherSuite suite = socket.getActiveCipherSuite();
```

Note

You must call `SSLSocket.getInputStream()` and `SSLSocket.getOutputStream()` before calling `SSLSocket.getActiveCipherSuite()` and querying the returned `CipherSuite` object through the accessor methods in the `CipherSuite` class.

8 Close the socket and the `SSLServerSocket` to terminate the communications session.

```
socket.close();
server.close();
```

Remarks

When you are building applications or applets with SSL/TLS capabilities, ensure that you include the appropriate jar files in the `CLASSPATH` environment variable or in the `-classpath` option of the Javac compiler. The jar file that contains the classes that provide SSL/TLS capabilities of the Toolkit is `entssl.jar`. Refer to the table in the **Modularization** section of the **Toolkit Overview** chapter for a description of the capabilities of each of the Toolkit's jar files.

Your application should import classes from the following Toolkit packages:

- `com.entrust.toolkit`
- `com.entrust.toolkit.credentials`
- `com.entrust.toolkit.util`
- `com.entrust.toolkit.x509.directory`
- `com.entrust.toolkit.exceptions`
- `iaik.x509`
- `iaik.security.ssl`
- `java.io`
- `java.net`

Using tunneling to communicate with a PKI

When there is a need to connect to a PKI remotely, using a virtual private network (VPN) from outside a corporate firewall for example, the Java language provides an efficient mechanism, in the form of servlets, for creating a tunnel to the network. The Security Toolkit for Java incorporates capabilities to connect users to the Directory and CA key management servers of a PKI using the HTTP or HTTPS communications protocols. This connection is made through a tunnel to servlets on a Web server that act as a proxy for client applications. The tunnel allows continuous two-way communication between the client and the proxy servlet.

The Toolkit's classes allow you to use HTTP to communicate with a Web server, or to use HTTPS on top of SSL for secure communication. The `HttpsDirectoryClient` and `HttpsManagerClient` classes contain objects of class `SSLClientContext`. During the login operation of an Entrust `User` entity, the `SSLClientContext` object is updated with the user's credentials to enable client authentication during such communication sessions.

The following classes, in `enttunnel.jar`, support the Toolkit's tunneling capabilities:

HTTP and HTTPS client classes

- `com.entrust.toolkit.util.HttpDirectoryClient` — implements the `com.entrust.toolkit.x509.LdapDirectory` interface and works with the `HttpDirectoryServlet` class to provide access to information stored in an X.509 directory structure.
- `com.entrust.toolkit.util.HttpsDirectoryClient` — a clone of the `HttpDirectoryClient` class that provides communication with an SSL-enabled Web server to support secure tunneling
- `com.entrust.toolkit.util.HttpManagerClient` — extends the `com.entrust.toolkit.util.ManagerTransport` class and works with the `HttpManagerServlet` class to transfer PKIX messages to and from the PKI's CA key management server
- `com.entrust.toolkit.util.HttpsManagerClient` — a clone of the `HttpManagerClient`class that provides communication with an SSL-enabled Web server to support secure tunneling

Servlets

- `com.entrust.toolkit.util.HttpDirectoryServlet` — a servlet that acts as a proxy for client requests to an LDAP Directory over HTTP
- `com.entrust.toolkit.util.HttpManagerServlet` — a servlet that acts as a proxy for client requests to a PKI, allowing users to be initialized through an HTTP tunnel

To use SSL tunneling in your applications, include the following jar files in your computer's `CLASSPATH` environment varaible, or in the `-classpath` option of the `Javac` compiler and `Java` interpreter:

- `entssl.jar`
- `enttunnel.jar`

Toolkit procedure — creating an HTTP tunnel to proxy servlets

The following steps demonstrate how to connect to a PKI using a tunnel to proxy servlets.

1 Create a new `User` object.

```
com.entrust.toolkit.User user = new User();
```

2 Prepare a connection to the PKI's CA key management server.

```
com.entrust.toolkit.util.ManagerTransport transport =
    new HttpManagerClient(managerServletURL, 0, managerIP);
```

3 Prepare a connection to the Directory.

```
com.entrust.toolkit.x509.LdapDirectory directory =
    new HttpDirectoryClient(directoryServletURL, 0);
```

Note

The next step shows how to encode ID and password information, if you are using an authenticating proxy server, for example.

4 Optionally, set the HTTP header information (in this case, the authorization request field whose values are base64 encoded).

```
String webId = <"ID">;
String webPassword = <"password">;

Base64Encoder encoder = new Base64Encoder( webId + ":" + webPassword );

directory.addHttpHeader( "Authorization", "Basic " + encoder.processString() );
transport.addHttpHeader( "Authorization", "Basic " + encoder.processString() );
```

Note

Section 14 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14) of RFC2616, Hypertext Transfer Protocol — HTTP/1.1, contains detailed information about HTTP header field definitions.

5 Optionally, you can specify an authenticating proxy server.

```
directory.setWebProxy(<"IP address of proxy">, <port number on proxy>);
transport.setWebProxy(<"IP address of proxy">, <port number on proxy>);
```

6 Establish the connections to the Directory and the CA key management server.

```
user.setConnections(directory, transport);
```

Note

The Web server you select must support servlets. Set up servlets to act as proxies for communication with the Directory and with the PKI's CA key management

server. Configure the servlets to recieve paramaters for the IP address and port number of the Directory (`directoryIP` and `directoryPort`) and CA key management (`managerIP` and `managerPort`) servers.

Toolkit procedure — creating an HTTPS tunnel to proxy servlets

The following procedure demonstrates how to connect to a PKI using communications secured with SSL through a tunnel to proxy servlets.

1 Create a new `User` object.

```
com.entrust.toolkit.User user = new User();
```

2 Prepare a connection to the PKI's CA key management server.

```
com.entrust.toolkit.util.ManagerTransport transport =
    new HttpsManagerClient(managerServletURL, 0, managerIP);
```

3 Prepare a connection to the Directory.

```
com.entrust.toolkit.x509.LdapDirectory directory =
    new HttpsDirectoryClient(directoryServletURL, 0);
```

4 Establish the connections to the Directory and the CA key management server.

```
user.setConnections(directory, transport);
```

Note

The Web server you select must support servlets. Enable SSL on the Web server and set up servlets to act as proxies for communication with the Directory and with the PKI's CA key management server. Configure the servlets to receive paramaters for the IP address and port number of the Directory (`directoryIP` and `directoryPort`) and CA key management (`managerIP` and `managerPort`) servers.

The Javadoc reference documentation for the `httpDirectoryClient`, `httpManagerClient`, `httpDirectoryServlet`, and `httpManagerServlet` classes provide guidance about the use of these classes with Web servers that support servlets.

Bypassing a proxy server

In release 6.1, the Toolkit's classes in the `org.w3c.www.http` and `org.w3c.www.protocol.http` packages (in the `enthttp.jar` file) check system properties to determine how to direct communications to proxy servers. Two system properties tell the HTTP protocol handler which proxy server and port to use:

- `http.proxyHost` — a `String` value specifying the proxy server (no default)
- `http.proxyPort` — a `String` value specifying a port number on the proxy server (default port number is 80 if `http.proxyHost` is specified)

In some situations you might want to bypass a proxy server or to specify a server for which there is no proxy. To accomplish this, set the following system property:

- `http.nonProxyHosts`

Using the `http.nonProxyHosts` property, you can designate a list of servers that are to receive requests and communications directly. Servers in the list must be separated with a pipe symbol (|) and you can use an asterisk (*) as a wildcard character. For example, `http:nonProxyHosts="www.acme.com|*.w3c.org|www.university.edu"`.

There are several ways to set `http.nonProxyHosts` to bypass a proxy host set by the `http.proxyHost` and `http.proxyPort` system properties:

- From the command line when you run the `java` interpreter with the `-D` option

```
-Dhttp:nonProxyHosts="www.acme.com|*.w3c.org|www.university.edu"
```

- Using the `java.lang.System.setProerties` method in your application

```
// Set proxy
System.setProperty("http.proxyHost", "localhost");

//set non-proxy host
System.setProperty("http.nonProxyHosts", "www.acme.com|*.w3c.org|www.university.edu");
```

Note

In this example, the default port number, 80, is used on `localhost`.

- As entries in a properties file

```
# ================================================================ #
# File name: proxy.properties                                      #
#                                                                  #
# Proxy properties file                                            #
# ================================================================ #

# Set proxy server
http.proxyHost="localhost"
http.proxyPort="729"

# Bypass proxy server
http.nonProxyHosts="www.acme.com|*.w3c.org|www.university.edu"
```

Note

The following code fragment is a simplified example of how to retrieve the three system properties set in the `proxy.properties` file.

```
// Create a Properties object containing existing system properties.
Properties props = new Properties(System.getProperties());

// Load proxy.properties file into props.
String proxyProps = <"path to proxy.properties file">;
FileInputStream fis = new FileInputStream(proxyProps);
props.load(fis);
fis.close();

// Set the combined properties as the system properties.
System.setProperties(props);
```

Communicating with Entrust Authority Roaming Server

The Toolkit includes API support for secure sockets layer (SSL) and transportation layer (TLS) communication with Entrust Authority Roaming Server (formerly Entrust/Roaming Server).

Note

The Toolkit supports only release 6.0 of Entrust Authority Roaming Server — earlier releases are not supported.

The Roaming Server SSL connection supports only the RSA signing algorithm, so Toolkit-based applications that communicate with Roaming Server expect to work with Roaming Server users created using this algorithm.

Support for Roaming Server includes the following activities:

- Roaming version query — returns the Roaming Server version number.
- Roaming login — takes a user's ID and password to retrieve credentials (Roaming Profile) from the Roaming Server, and performs the login operation.
- Roaming registration — sends a request to register a user with the Roaming Server.
- Roaming deregistration — sends a request to cancel a user's registration with the Roaming Server.
- Roaming profile creation — takes a user's ID, password, and a `CredentialCreator` and sends a request to the Roaming Server to create credentials.
- Roaming profile recovery — takes a user's ID, password, and a `CredentialRecoverer` and sends a request to the Roaming Server to recover a user's credentials.
- Roaming update — sends a request to the Roaming Server to update a given set of files related to a user.
- Roaming password change — sends a request to the Roaming Server to update a user's credentials with a new password.
- Roaming key update — sends a request to update a user's encryption public key and signing private key.
- Roaming update of another user — sends a request to the Roaming Server to update a given set of files related to another user.
- Roaming deregistration of another user — sends a request to cancel another user's registration with the Roaming Server.

Refer to the `com.entrust.toolkit.roaming.RoamingUser` class in the Javadoc reference documentation for more detailed information about the capabilities listed here.

Toolkit procedure — Roaming user operations

This procedure demonstrates the following tasks:

- Registering a user with Roaming Server
- Logging in a user to Roaming Server
- Creating an Entrust `User` instance to perform cryptographic operations
- Performing a key update operation for a user's signing private key

1 Create an instance of the `com.entrust.toolkit.roaming.RoamingUser` class, specifying the location of the user's Entrust `ini` file in the constructor.

```
roamingUser = new RoamingUser("<path to entrust.ini file>");
```

2 Register the user with Roaming Server.

```
roamingUser.register(<path to user's Entrust Profile>,
                     <userID>
                     new SecureStringBuffer(<user's Entrust Profile password>));
```

3 Perform a Roaming Server login operation.

```
roamingUser.login(<userID>,
                  new SecureStringBuffer(<user's Entrust Profile password>));
```

4 Create an Entrust `User` instance to perform cryptographic operations.

a Retrieve the user's Entrust Profile as a `byte` array.

```
byte[] profile = roamingUser.getProfile();
```

b Create a `ByteArrayInputStream` and a `StreamProfileReader` to read the Profile.

```
ByteArrayInputStream bais = new ByteArrayInputStream( profile );
com.entrust.toolkit.credentials.StreamProfileReader reader =
    new StreamProfileReader( bais );
```

c Instantiate a `User` instance and log in.

```
com.entrust.toolkit.User user = new User();
user.login( reader, new SecureStringBuffer(<user's Entrust Profile password>) );
```

Note

At this stage, you can perform cryptographic operations with the `User` instance.

d Log out the user when your cryptographic operations are complete.

```
user.logout();
```

5 Determine the need for a signing private key update and perform the update operation if necessary.

```
if (roamingUser.signingKeyUpdateRequired())
{
    roamingUser.updateSigningKeys();
}
```

6 Log out the user from Roaming Server.

```
roamingUser.logout();
```

SSL/TLS security considerations

Client and server certificates

After the handshake phase of the client-server negotiation, your client application should examine the server certificates and compare them with the expected identity of the server. A server application should perform the same check of the client's certificates. The following code fragments illustrate how to obtain certificate information.

Client

```
// Obtain information about server certificate.  (Note: socket is an SSLSocket)
     X509Certificate[] serverChain = socket.getPeerCertificateChain();
     System.out.println("Server certificate:");
     System.out.println("subject: " + serverChain[0].getSubjectDN().getName());
     System.out.println("issuer:  " + serverChain[0].getIssuerDN().getName());

     // ... code to verify this certificate information against the expected
     // server identity ...
```

Server

If the server requires client authentication, the server application must also ensure that the client really did send a certificate.

```
// Obtain information about client certificate.  (Note: socket is an SSLSocket)
      X509Certificate[] clientChain = socket.getPeerCertificateChain();
      if (clientChain == null) {
          System.out.println("no client certificate received");
          // Throw an exception if client authentication was required and
          // invoke socket.close()
      }
      else {
          System.out.println("Client certificate:");
          System.out.println("subject: " + clientChain[0].getSubjectDN().getName());
          System.out.println("issuer:  " + clientChain[0].getIssuerDN().getName());
      }

      // ... code to verify this certificate information against the expected
      // client identity ...
```

Cipher suites

Except for communications between clients and servers that are known to support only weak ciphers, the cipher suites you use in your applications should consist exclusively of strong ciphers. This prevents the possibility of a weak cipher being chosen even when both parties support the same strong cipher.

SSL and JSSE

When you use the JSSE interface the server application always uses the RSA decryption key for sensitive operations. You can choose any of the cipher suites in the Toolkit, provided the necessary keys are in place. However, for sensitive operations where you want to specify which key to use for signing operations, and which to use for encryption operations, build the cipher suites as described in this subsection.

The `addServerCredentials(X509Certificate[], PrivateKey)` and `addServerCredentials(KeyAndCert, int)` methods in the `iaik.security.ssl.SSLServerContext` class do not operate in the same way.

Using the `addServerCredentials(X509Certificate[], PrivateKey)` method, as demonstrated in the sample applications that accompany the Toolkit, the server supports either an RSA encryption public key and encryption public certificate, or an RSA verification public certificate, but not both. To set an RSA encryption public key, encryption public certificate, and verification public certificate (steps 2a and 2b below) at the same time, use the `addServerCredentials(KeyAndCert, int)` method.

Using the Toolkit's SSL interface

The following list shows combinations of certificates and keys that you can set using the `iaik.security.ssl.SSLServerContext` class and its methods.

1 The server has a DSA verification certificate and an RSA encryption certificate — set the keys, certificates, and the cipher suite in the `SSLServerContext` class. Allowed ciphers

Allowed ciphers

Refer to `iaik.security.ssl.CipherSuite` in the Javadoc reference documentation.

All ciphers with DSA in the cipher suite name.

All ciphers that contain `SSL_RSA_EXPORT` in the cipher suite name, provided they have, at most, a 512-bit RSA modulus.

All ciphers that contain `SSL_RSA_EXPORT1024` in their name, provided they have, at most, a 1024-bit RSA modulus.

All ciphers that contain `SSL_RSA_WITH` in the cipher suite name.

2 The server has an RSA verification certificate and an RSA encryption certificate

a Set the encryption key, certificate, and the cipher suite using the iaik.security.ssl.SSLServerContext class and its methods.

Allowed ciphers

All ciphers that contain `SSL_RSA_EXPORT` in the cipher suite name, provided they have, at most, a 512-bit RSA modulus.

All ciphers that contain `SSL_RSA_EXPORT1024` in their name, provided they have, at most, a 1024-bit RSA modulus.

All ciphers that contain `SSL_RSA_WITH` in the cipher suite name.

b Set the verification certificate and the cipher suite using the

iaik.security.ssl.SSLServerContext class and its methods.

Allowed ciphers

All ciphers that contain `SSL_RSA_EXPORT` in the cipher suite name, provided they have an RSA modulus greater than 512 bits.

All ciphers that contain `SSL_RSA_EXPORT1024` in their name, provided they have an RSA modulus greater than 512 bits.

All ciphers that contain `SSL_RSA_WITH` in the cipher suite name, except those that contain `SSL_RSA_WITH`.

SSL version 2.0

SSL version 2.0 is generally not considered to be secure. Its negotiation of the cipher suite is susceptible to a cipher suite rollback attack, which can cause the negotiating parties to accept a weaker cipher suite than both are capable of using. Use SSL version 2.0 only if your applications have to deal with older servers.

Chapter 9

Glossary

Glossary

Advanced Encryption Standard (AES)

Advanced Encryption Standard algorithm — refer to FIPS PUB 197 (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf)

AES

see **Advanced Encryption Standard (AES)**

authentication

confirmation of the identities of parties involved in a transaction

ARL

see **Authority Revocation List (ARL)**

Authority Revocation List

Authority Revocation List (ARL) — signed, time stamped list of the serial numbers of CA public certificates

CA

see **Certification Authority (CA)**

Certification Authority

trusted entity whose central responsibility is certifying the authenticity of users by issuing signed public key certificates, policy certificates, cross-certificates, certificate revocation lists (CRLs), and authority revocation lists (ARLs)

CA domain

population of users for which the Certification Authority (CA) has the authority to issue certificates

canonical form (of an XML document)

normalized physical representation that establishes a standard baseline for processing an document

CAST

design procedure used to build algorithms, where the resulting algorithm is a CAST Cipher

CBS

see **Cipher Block Chaining (CBC)**

Cipher Block Chaining (CBC)

cipher mode that takes each block of plaintext and combines it with the previous block's ciphertext, using an exclusive OR (XOR) operation; the result is encrypted to form a block of ciphertext

confidentiality

concealment of information from unauthorized parties

credentials

set of data in a generic PKI that defines an entity and contains a user's critical information, or keying material; an Entrust Profile is a specific type of the more general concept of credentials

credentials file

set of critical information about a user of a PKI; in an Entrust PKI, a user's credentials file is usually stored in a file with an `epf` file name extension

Certificate Revocation List

lists, published in the Directory of a PKI, that specify the unique serial numbers of all revoked certificates

CRL

see **Certificate Revocation List**

cross-certification

process by which two CAs securely exchange keying information so that each can effectively certify the trustworthiness of the other's keys

Data Encryption Standard (DES)

Data Encryption Standard (DES) algorithm — refer to FIPS PUB 46-2 (http://www.itl.nist.gov/fipspubs/fip46-2.htm).

DES

see **Data Encryption Standard (DES)**

detached XML signature

type of XML digital signature that signs data that is not contained in the XML signature's `<Signature>` element

digital signature

electronic signature — the result of applying a hash function to data and encrypting the resulting hash value with the user's signing private key

Digital Signature Algorithm (DSA)

Digital Signature Algorithm (DSA) — developed for NIST by the NSA, and designed to be used in NIST's Digital Signature Standard(DSS).

DSA

see **Digital Signature Algorithm (DSA)**

ECB

see **Electronic Code Book (ECB)**

Electronic Code Book (ECB)

cipher mode that takes each block of plaintext and encrypts it to form a block of

ciphertext

ECDSA

see **Elliptic Curve Digital Signature Algorithm (ECDSA)**

Elliptic Curve Digital Signature Algorithm (ECDSA)

Elliptic Curve Digital Signature Algorithm (ECDSA) — provides digital signature capabilities using the Java Cryptography Extension (JCE) for both stand-alone applications and applications that work with an Entrust PKI; the elliptic curve counterpart of the digital signature algorithm (DSA) using arithmetic based on elliptic curves

EDE

see **Encrypt Decrypt Encrypt (EDE)**

Encrypt Decrypt Encrypt (EDE)

cipher mode defined by ANSI X9.17 [2] for key encryption and decryption with pairs of 64-bit keys

Entrust Login Interface (ELI)

centralizes control of inactivity time outs across applications, built using Entrust Authority Toolkit, that are running on a computer; part of the runtime components, which provide the underlying cryptographic capabilities that Entrust/PKI and Entrust client applications are based upon

Entrust Profile

set of data that contains an Entrust user's critical information, such as a user's identity, decryption private keys, signing keys, and the CA signing keys; usually stored as a file with an `epf` file name extension

Entrust/Entelligence

application that provides security management across all Entrust desktop applications; automates the management of identification, verification, and privacy on behalf of users by handling user certificates, and enabling encryption and digital signatures for applications across the desktop; also integrates into the desktop environment providing the Single Login capability to all applications secured by Entrust

enveloped XML signature

type of XML digital signature that embeds the XML signature's `<Signature>` element in signed content

enveloping XML signature

type of XML digital signature that embeds signed content into the XML signature's `<Signature>` element

Federal Public Key Infrastructure (FPKI)

Federal Public Key Infrastructure — refer to http://www.cio.gov/fpkisc/

FPKI

see **Federal Public Key Infrastructure (FPKI)**

Hash function

algorithm that creates a fixed length, mathematical summary of a piece of data creating a unique digital fingerprint of the data.; used in the creation of digital signatures

HTTPS

see **HyperText Transport Protocol Secure (HTTPS)**

HyperText Transport Protocol Secure (HTTPS)

transport protocol used to access a secure Web server through a secure port number rather than the default port number of 80

IAIK

Institute for Applied Information Processing and Communications, in Austria (http://www.iaik.tu-graz.ac.at)

identification

see **authentication**

integrity

assurance that the content of a communication has not been modified during transmission

Java Native Interface (JNI)

native programming interface for the Java programming language that allows Java code running inside a Java Virtual Machine to operate with applications and libraries written in other programming languages such as C, C++, and assembly language

JNI

see **Java Native Interface (JNI)**

Java Naming and Directory Interface™ (JNDI)

standard extension to the JavaTM platform, providing Java technology-enabled applications with a unified interface to multiple naming and directory services in the enterprise [from Java Naming and Directory Interface™ (JNDI), http://java.sun.com/products/jndi/]

JNDI

see **Java Naming and Directory Interface™ (JNDI)**

Java Secure Socket Extension (JSSE)

set of Java packages that enable secure Internet communications [from Java™ Secure Socket Extension (JSSE), http://java.sun.com/products/jsse/]

JSSE

see **Java Secure Socket Extension (JSSE)**

Key store

container that holds cryptographic keys and certificates

MD2

message-digest algorithm that takes as input a message of arbitrary length, and produces as output a 128-bit digital fingerprint or message digest of the input

MD5

secure hashing function that converts an arbitrarily long data stream into a digest of fixed size

non-repudiation

binding an entity to a transaction in which it participates, so that the validity of the transaction cannot be rejected or denied at a later date, allowing the recipient of the transaction to demonstrate to a neutral third party that the sender was indeed send the originator of the transaction

Profile

see **Entrust Profile**.

PKCS

see **Public Key Cryptography Standards**

Public Key Cryptography Standards

standards for cryptographic operations used in public key cryptography — refer to http://www.rsasecurity.com/rsalabs/pkcs/

PKIX

see **Public Key Infrastructure X.509**

Public Key Infrastructure X.509

working group within the Internet Engineering Task Force (IETF) that has developed standards for formatting and transporting information within a PKI (the Certificate Management Protocol, or PKIX-CMP, for example)

Public Key Cryptography

a cryptographic method that uses keys that are public, for encryption and verification, and private, for decryption and digitally signing data

Public Key Infrastructure (PKI)

system that manages keys and certificates used in public key cryptography

RC2

64 bit block cipher that supports variable key lengths

RC4

stream cipher that supports variable key lengths

RSA

popular public key algorithm, named after its developers Ron Rivest, Adi Shamir, and

Leonard Adleman, that can be used for both encryption and digital signatures

Secure Multipurpose Internet Mail Extensions (S/MIME)

widely used application of PCKS #7 for exchanging messages and data by transport protocols capable of conveying MIME data, such as email, and http

S/MIME

see **Secure Multipurpose Internet Mail Extensions (S/MIME)**

Secure Hash Algorithm (SHA)

developed by NIST and NSA as a hash algorithm to be used with the Digital Signature Standard (DSS) — refer to FIPS PUB 180-1 (http://www.itl.nist.gov/fipspubs/fip180-1.htm)

Server Login

feature that allows an application to gain access to Entrust credentials without the need for manual authentication

Single Login

feature of the Entrust Login Interface (ELI) that allows users to log in to more than one application, built using Entrust Authority Toolkits, running in Microsoft Windows operating systems, without having to identify, or authenticate, themselves each time they want to use an application

Simple Object Access Prototcol (SOAP)

light-weight message-based protocol for exchanging information and invoking services on the World Wide Web — refer to the W3C's architecture domain Web site (http://www.w3.org/2000/xp/Group/)

SOAP

see **Simple Object Access Prototcol (SOAP)**

Secure Sockets Layer (SSL)

secure communications protocol that encrypts and decrypts messages for online transmission; SSL sessions involve the exchange of a secret key and encryption of data using the exchanged key; originally developed by Netscape, now merged with other protocols by the Internet Engineering Task Force (IETF) to create the Transportation Layer Security (TLS) protocol

SSL

see **Secure Sockets Layer (SSL)**

Triple-DES

variation of the DES algorithm that uses three 64-bit keys

TLS

see **Transport Layer Security (TLS)**

Transport Layer Security (TLS)

security protocol that combines SSL version 3.0 with other protocols; backwards-compatible with SSL version 3.0

Transform

mathematical term that refers to the result of a transformation

Uniform Resource Identifier (URI)

short strings that identify Web resources, such as documents, images, and files

User

in a PKI defines an entity that has been identified and approved by a Certification Authority (CA)

User class

represents an entity, or end user in a PKI environment and is the primary class in the high-level API. of the Security Toolkit for Java

X.509 certificates

standard that ensures interoperability between systems that use digital certificates

XML digital signature

digital signature represented by a `<Signature>` element and its contents in an XML document, that ensures the authenticity, integrity, and non-repudiation of data exchanged over computer networks; refer to the World Wide Web Consortium's (W3C) XML-Signature Syntax and Processing W3C Recommendation 12 February 2002 (http://www.w3.org/TR/xmldsig-core/)

XML document

well-formed document, or data object, as defined by the W3C Recommendation 6 October 2000 Extensible Markup Language (XML) 1.0 (Second Edition) (http://www.w3.org/TR/2000/REC-xml-20001006); must contain at least one element, the root or document element, which encloses all other elements in the document

XML fragment

one or more well-formed elements in an XML document; incomplete XML document

XML encryption

cryptographic method for securing all, or part of, an XML document or arbitrary data; based upon the proposals published by the W3C's XML Encryption Syntax and Processing candidate recommendation of 02 August 2002 (http://www.w3.org/TR/2002/CR-xmlenc-core-20020802/).

Index

A

C

D

E

F

H

I

J

K

M

N

O

P

R

S

T

U

X