Oracle® Fusion Middleware

Developing Applications with Oracle Security Developer Tools

12c (12.2.1.4.0)

E95936-01

September 2019

ORACLE

Oracle Fusion Middleware Developing Applications with Oracle Security Developer Tools, 12c (12.2.1.4.0)

E95936-01

Primary Author: Sudhira Subudhi

Contributing Authors: Gaurav Sharma

Contents

Preface

What's New in Oracle Security Developer Tools?

1   Introduction to Oracle Security Developer Tools

ORACLE®

2   Oracle Crypto

3   Oracle Security Engine

4   Oracle CMS

5   Oracle S/MIME

6   Oracle PKI SDK

7   Oracle XML Security

8   Oracle SAML

9   Oracle Web Services Security

10 Oracle Liberty SDK

11 Oracle XKMS

12   Oracle JSON Web Token

A   Migrating to the JCE Framework

B   References

List of Tables

List of Tables

Preface

Developing Applications with Oracle Security Developer Tools provides reference information about the Oracle Security Developer Tools. This Preface contains the following topics:

- Intended Audience
- Documentation Accessibility
- Related Documents
- Conventions

Intended Audience

Developing Applications with Oracle Security Developer Tools is intended for Java developers responsible for developing secure applications. This documentation assumes programming proficiency using Java, and familiarity with security concepts such as cryptography, public key infrastructure, Web services security, and identity federation.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documentation available in the Oracle Fusion Middleware 12*c* (12.2.1) documentation set:

- *Oracle Fusion Middleware Securing Web Services and Managing Policies with Oracle Web Services Manager*
- *Oracle Fusion Middleware Securing Applications with Oracle Platform Security Services*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in Oracle Security Developer Tools?

This preface introduces the new and changed features of Oracle Security Developer Tools.

New Features in 12*c* (12.2.1)

The API doc set (javadocs) has been consolidated into a single javadoc, the *Java API Reference for Oracle Security Developer Tools*.

This edition of *Developing Applications with Oracle Security Developer Tools* contains usability enhancements and editorial corrections.

Updates in September 2014 Documentation Refresh for 12*c* (12.1.3)

This edition of *Developing Applications with Oracle Security Developer Tools* contains usability enhancements and editorial corrections.

New Features in 12*c* (12.1.3)

New Java API references (javadocs) have been published for all the tools.

1

Introduction to Oracle Security Developer Tools

Oracle Security Developer Tools provide the cryptographic building blocks necessary for developing robust security applications, ranging from basic tasks such as digital signatures and secure messaging to more complex projects such as securely implementing a service-oriented architecture. The tools are built upon the core foundations of cryptography, public key infrastructure, web services security, and federated identity management.
Security tools are a critical component for application development projects. Commercial requirements and government regulations dictate that sensitive data be kept confidential and protected from tampering or alteration.

A wide range of Oracle products utilize the Oracle Security Developer Tools, including:

*   the Oracle JDeveloper integrated service environment

*   Oracle Platform Security Services, which include SSL configuration features for system components, and Oracle Wallet, which is utilized in multiple components including Oracle Database

*   system components like Oracle Web Services Manager (OWSM); Business Integration (B2B); and Oracle SOA Suite

This chapter takes a closer look at the underlying security technologies and introduces the components of the Oracle Security Developer Tools. It covers these topics:

*   About Cryptography

*   About Public Key Infrastructure (PKI)

*   About Web Services Security

*   About SAML

*   About Identity Federation

*   About Oracle Security Developer Tools

*   About Supported Standards

*   Setting the CLASSPATH Environment Variable

*   References

1.1 About Cryptography

Cryptography protects the transmitted messages in communication channels from being intercepted (a passive attack) or modified (an active attack) by an intruder. To protect the message, an originator uses a cryptographic tool to convert plain, readable messages or plaintext into encrypted ciphertext. The message recipient likewise uses a cryptographic tool to decrypt the ciphertext into its original readable format.

Cryptography secures communications over a network such as the internet by providing:

- Authentication, which assures the receiver that the information is coming from a trusted source. Authentication is commonly achieved through the use of a Message Authentication Code (MAC), digital signature, and digital certificate.

- Confidentiality, which ensures that only the intended receiver can read a message. Confidentiality is commonly attained through encryption.

- Integrity, which ensures that the received message has not been altered from the original. Integrity is commonly ensured by using a cryptographic hash function.

- Non-repudiation, which is a way to prove that a given sender actually sent a particular message. Non-repudiation is typically achieved through the use of digital signatures.

For additional cryptography resources, refer References.

1.1.1 Types of Cryptographic Algorithms

Cryptographic algorithms or ciphers use keys to convert plain text to ciphertext and vice versa. Essentially, there are three types of cryptographic algorithms categorized by the number of keys used for encryption and decryption, and by their application and usage. These are Symmetric Cryptographic Algorithms, Asymmetric Cryptographic Algorithms, and Hash Functions.

Each type is optimized for certain applications. Hash functions are suited for ensuring data integrity. Symmetric cryptography is ideally suited for encrypting messages. Asymmetric cryptography is used for the secure exchange of keys, authentication, and non-repudiation. Asymmetric cryptography could also be used to encrypt messages, although this is rarely done. Symmetric cryptography operates about 1000 times faster, and is better suited for encryption than asymmetric cryptography.

The cryptographic algorithm types are:

- About Symmetric Cryptographic Algorithms
- About Asymmetric Cryptographic Algorithms
- Understanding Hash Functions

1.1.1.1 About Symmetric Cryptographic Algorithms

A symmetric cryptography algorithm (also known as secret key cryptography) uses a single key for both encryption and decryption. The sender uses the key to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key to decrypt the message and recover the plaintext. The key must be known to both the sender and receiver. The biggest problem with symmetric cryptography is the secure distribution of the key.

Symmetric cryptography schemes are generally categorized as being either a block cipher or stream cipher. A block cipher encrypts one fixed-size block of data (usually 64 bits) at a time using the same key on each block. Some common block ciphers used today include Blowfish, AES, DES, and 3DES.

Stream ciphers operate on a single bit at a time and implement some form of feedback mechanism so that the key is constantly changing. RC4 is an example of a stream cipher that is used for secure communications using the SSL protocol.

1.1.1.2 About Asymmetric Cryptographic Algorithms

An asymmetric cryptography algorithm (also known as public key cryptography) uses one key to encrypt the plaintext and another key to decrypt the ciphertext. It does not matter which key is applied first, but both keys are required for the process to work.

In asymmetric cryptography, one of the keys is designated the public key and is made widely available. The other key is designated the private key and is never revealed to another party. To send messages under this scheme, the sender encrypts some information using the receiver's public key. The receiver then decrypts the ciphertext using her private key. This method can also be used to prove who sent a message (non-repudiation). The sender can encrypt some plaintext with her private key, and when the receiver decrypts the message with the sender's public key, the receiver knows that the message indeed came from that sender.

Some of the common asymmetric algorithms in use today are RSA, DSA, and Diffie-Hellman.

1.1.1.3 Understanding Hash Functions

A hash function (also known as a **message digest**) is a one-way encryption algorithm that essentially uses no key. Instead, a fixed-length hash value is computed based upon the plaintext that makes it impossible for either the contents or length of the plaintext to be recovered. Hash algorithms are typically used to provide a digital fingerprint of a file's contents, often used to ensure that the file has not been altered by an intruder or virus. Hash functions are also commonly employed by many operating systems to encrypt passwords. Hash functions help preserve the integrity of a file. Some common hash functions include MD2, MD4, MD5 and SHA.

1.2 About Public Key Infrastructure (PKI)

A public key infrastructure (PKI) is designed to enable secure communications over public and private networks. Besides secure transmission and storage of data, PKI also enables secure e-mail, digital signatures, and data integrity. PKI uses public key cryptography, a mathematical technique that uses a pair of related cryptographic keys to verify the identity of the sender (digital signature), and to ensure the privacy of a message (encryption). PKI facilities secure information exchange over Internet.

Critical elements for achieving the goals of PKI include:

• Encryption algorithms and keys to secure communications

• Digital certificates that associate a public key with the identity of its owner

• Key distribution methods to permit widespread, secure use of encryption

• A trusted entity, known as a Certificate Authority (CA), to vouch for the relationship between a key and its legitimate owner

• A Registration Authority (RA) that is responsible for verifying the information supplied in requests for certificates made to the CA

Relying third parties use the certificates issued by the CA and the public keys contained in them to verify digital certificates and encrypt data.

1.2.1 Understanding Key Pairs

Encryption techniques often use a key, known only to the sender and the recipient. Public key cryptography uses a key pair of mathematically related cryptographic keys—the public key and the private key.

When both use the same key, the encryption scheme is called symmetric. Difficulties with relying on a symmetric system include getting that key to both parties without allowing an eavesdropper to get it, too; and the fact that a separate key is needed for every two people, so that each individual must maintain many keys, one for each recipient.

For an explanation of the use of key pairs, see "About Asymmetric Cryptographic Algorithms".

Table 1-1 summarizes who uses public and private keys and when:

Table 1-1   Summary of Public and Private Key Usage

Function	Key Type	Whose Key
Encrypt data for a recipient	Public key	Receiver
Sign data	Private key	Sender
Decrypt data received	Private key	Receiver
Verify a signature	Public key	Sender

1.2.2 About the Certificate Authority

A Certificate Authority (CA) is a trusted third party that vouches for the public key owner's identity.

Examples of certificate authorities include Verisign and Thawte.

1.2.3 What are Digital Certificates?

The certification authority validates the public key's link to a particular entity by creating a digital certificate. This digital certificate contains the public key and information about the key holder and the signing certification authority.

Using a PKI certificate to authenticate one's identity is analogous to identifying oneself with a driver's license or passport.

1.2.4 Related PKI Standards

A number of standards and protocols support PKI certificate implementation. These are Cryptographic Message Syntax (CMS), Secure/Multipurpose Internet Mail Extension (S/MIME), Lightweight Directory Access Protocol (LDAP), Time Stamp

Protocol (TSP), Online Certificate Status Protocol (OCSP), and Certificate Management Protocol (CMP).

Cryptographic Message Syntax

Cryptographic Message Syntax (CMS) is a general syntax for data protection developed by the Internet Engineering Task Force (IETF). It supports a wide variety of content types including signed data, enveloped data, digests, and encrypted data, among others. CMS allows multiple encapsulation so that, for example, previously signed data can be enveloped by a second party.

Values produced by CMS are encoded using X.509 Basic Encoding Rules (BER), meaning that the values are represented as octet strings.

Secure/Multipurpose Internet Mail Extension

Secure/Multipurpose Internet Mail Extension (S/MIME) is an Internet Engineering Task Force (IETF) standard for securing MIME data through the use of digital signatures and encryption.

S/MIME provides the following cryptographic security services for electronic messaging applications:

- Authentication
- Message integrity and non-repudiation of origin (using digital signatures)
- Privacy and data security (using encryption)

Lightweight Directory Access Protocol

Lightweight Directory Access Protocol (LDAP) is the open standard for obtaining and posting information to commonly used directory servers. In a public key infrastructure (PKI) system, a user's digital certificate is often stored in an LDAP directory and accessed as needed by requesting applications and services.

Time Stamp Protocol

In a Time Stamp Protocol (TSP) system, a trusted third-party Time Stamp Authority (TSA) issues time stamps for digital messages. Time stamping proves that a message was sent by a particular entity at a particular time, providing non-repudiation for online transactions.

The Time Stamp Protocol, as specified in RFC 3161, defines the participating entities, the message formats, and the transport protocol involved in time stamping a digital message.

To see how a time-stamping system can work, suppose Sally signs a document and wants it time stamped. She computes a message digest of the document using a secure hash function and then sends the message digest (but not the document itself) to the TSA, which sends her in return a digital time stamp consisting of the message digest, the date and time it was received at the TSA server, and the signature of the TSA. Since the message digest does not reveal any information about the content of the document, the TSA cannot eavesdrop on the documents it time stamps. Later, Sally can present the document and time stamp together to prove when the document was written. A verifier computes the message digest of the document, makes sure it matches the digest in the time stamp, and then verifies the signature of the TSA on the time stamp.

Online Certificate Status Protocol

Online Certificate Status Protocol (OCSP) is one of two common schemes for checking the validity of digital certificates. The other, older method, which OCSP has superseded in some scenarios, is known as the certificate revocation list (CRL).

OCSP overcomes the chief limitation of CRL: the fact that updates must be frequently down-loaded to keep the list current at the client end. When a user attempts to access a server, OCSP sends a request for certificate status information. The server sends back a response of good, revoked, or unknown. The protocol specifies the syntax for communication between the server (which contains the certificate status) and the client application (which is informed of that status).

Certificate Management Protocol

The certificate management protocol (CMP) handles all relevant aspects of certificate creation and management. CMP supports interactions between public key infrastructure (PKI) components, such as Certificate Authorities (CAs), Registration Authorities (RAs), and end entities that are issued certificates.

1.2.5 Benefits of PKI

PKI provides secure and reliable authentication. It provides data integrity, non-repudiation, and prevents unauthorized access to transmitted or stored information.

PKI provides users with the following benefits:

- Secure and reliable authentication of users

  Reliable authentication relies on two factors. The first is proof of possession of the private key part of the public/private pair, which is verified by an automatic procedure that uses the public key. The second factor is validation by a certification authority that a public key belongs to a specific identity. A PKI-based digital certificate validates this identity connection based on the key pair.

- Data integrity

  Using the private key of a public/private key pair to sign digital transactions makes it difficult to alter the data in transit. This "digital signature" is a coded digest of the original message encrypted by the sender's private key. Recipients can readily use the sender's corresponding public key to verify who sent the message and the fact that it has not been altered. Any change to the message or the digest would have caused the attempted verification using the public key to fail, telling the recipient not to trust it.

- Non-repudiation

  PKI can also be used to prove who sent a message. The sender encrypts some plaintext with her private key to create a digital signature, and when the receiver decrypts the message with the sender's public key, the receiver knows that the message indeed came from that sender, making it difficult for the message originator to disown the message; this capability is known as non-repudiation.

- Prevention of unauthorized access to transmitted or stored information

  The time and effort required to derive the private key from the public key makes it unlikely that the message would be decrypted by anyone other than the key pair owner.

1.3 About Web Services Security

Web services provide a standard way for organizations to integrate Web-based applications using open standard technologies such as XML, SOAP, and WSDL. While the core SOAP specification solves many problems related to XML and Web Services, it does not provide a means to address message security requirements such as confidentiality, integrity, message authentication, and non-repudiation.

SOAP is a lightweight protocol for exchange of information in a service oriented environment. In such an environment, applications can expose selected functionality (business logic, for example) for use by other applications. SOAP provides the means by which applications supply and consume these services; it is an XML-based protocol for message transport in a distributed, decentralized Web Services application environment.

The need for securing SOAP prompted OASIS to put forward the Web Services Security standard, which:

- Specifies enhancements to allow signing and encryption of SOAP messages.

- Describes a general-purpose method to associate security tokens with messages.

- Provides additional means for describing the characteristics of tokens that are included with a message.

1.4 About SAML

Security Assertions Markup Language (SAML) is an XML-based framework for exchanging security information over the Internet. SAML enables the exchange of authentication and authorization information between various security services systems that otherwise would not be able to interoperate.

The SAML 1.0, 1.1, and 2.0 specifications were adopted by the Organization for the Advancement of Structured Information Standards (OASIS) in 2002, 2003, and 2005 respectively. OASIS is a worldwide not-for-profit consortium that drives the development, convergence, and adoption of e-business standards.

SAML 2.0 marks the convergence of the Liberty ID-FF, Shibboleth, and SAML 1.0/1.1 federation protocols.

1.4.1 Understanding SAML Assertions

SAML associates an identity, such as an e-mail address or a directory listing, with a subject, such as a user or system, and defines the access rights within a specific domain. The basic SAML document is the `Assertion`, which contains declarations of facts about a `Subject` (typically a user).

SAML provides three kinds of declarations, or `Statements`:

- `AuthnStatement` asserts that the user was authenticated by a particular method at a specific time.

- `AttributeStatement` asserts that the user is associated with particular attributes or details, for example an employee number or account number.

- `AuthzDecisionStatement` asserts that the user's request for a certain access to a particular resource has been allowed or denied.

Assertions are XML documents generated about events that have already occurred. While SAML makes assertions about credentials, it does not actually authenticate or authorize users. Example 1-1 shows a typical SAML authentication assertion wrapped in a SAMLP response message:

Example 1-1   Sample SAMLP Response Containing a SAML 1.0 Authentication Assertion

```
<samlp:Response
      MajorVersion="1" MinorVersion="0"
      ResponseID="128.14.234.20.90123456"
      InResponseTo="123.45.678.90.12345678"
      IssueInstant="2005-12-14T10:00:23Z"
      xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
      xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol">
      <samlp:Status>
          <samlp:StatusCode Value="samlp:Success" />
      </samlp:Status>
      <saml:Assertion
          MajorVersion="1" MinorVersion="0"
          AssertionID="123.45.678.90.12345678"
          Issuer="IssuingAuthority.com"
          IssueInstant="2005-12-14T10:00:23Z" >
          <saml:Conditions
              NotBefore="2005-12-14T10:00:30Z"
              NotAfter="2005-12-14T10:15:00Z" />
          </saml:Conditions>
          <saml:AuthenticationStatement
              AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
              AuthenticationInstant="2005-12-14T10:00:20Z">
              <saml:Subject>
                  <saml:NameIdentifier NameQualifier="RelyingParty.com">
                    john.smith
                  </saml:NameIdentifier>
                  <saml:SubjectConfirmation>
                      <saml:ConfirmationMethod>
                          urn:oasis:names:tc:SAML:1.0:cm:artifact-01
                      </saml:ConfirmationMethod>
                  </saml:SubjectConfirmation>
              </saml:Subject>
          </saml:AuthenticationStatement>
      </saml:Assertion>
</samlp:Response>
```

1.4.2 Understanding SAML Requests and Responses

When a user signs into a SAML-compliant service, the service sends a "request for authentication assertion" to the issuing authority (identity provider). The issuing authority returns an "authentication assertion" reference stating that the user was authenticated by a particular method at a specific time.

The authority that issues assertions is known as the **issuing authority** or identity provider. An issuing authority can be a third-party service provider or an individual business that is serving as an issuing authority within a private federation of businesses. SAML-compliant applications and services, which trust the issuing authority or identity provider and make use of its services, are called **relying parties** or service providers.

1.4.2.1 About the SAML Request and Response Cycle

In a typical SAML cycle, the relying party (or service provider), which needs to authenticate a specific client request, sends a SAML request to its issuing authority or identity provider. The identity provider responds with a SAML assertion, which supplies the relying party or service provider with the requested security information.

For example, when a user signs into a SAML-compliant service of a relying party or identity provider, the service sends a "request for authentication assertion" to the issuing authority (identity provider). The issuing authority returns an "authentication assertion" reference stating that the user was authenticated by a particular method at a specific time. The service can then pass this assertion reference to other relying party/identity provider sites to validate the user's credentials. When the user accesses another SAML-compliant site that requires authentication, that site uses the reference to request the "authentication assertion" from the issuing authority or identity provider, which states that the user has already been authenticated.

At the issuing authority, an assertion layer handles request and response messages using the SAML protocol, which can bind to various communication and transport protocols (HTTP, SOAP, and so on). Note that while the client always consumes assertions, the issuing authority or identity provider can act as producer and consumer since it can both create and validate assertions.

This cycle is illustrated in Figure 1-1.

Figure 1-1   SAML Request-Response Cycle
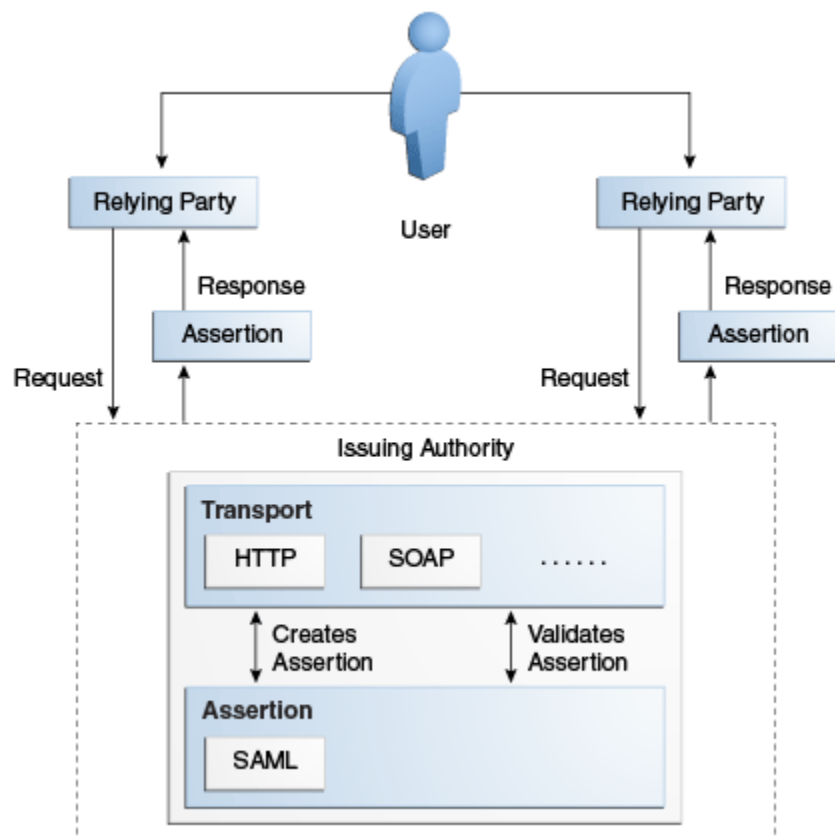
This figure shows a SAML request and response cycle, and shows a user, boxes for relying parties, and a box for the issuing authority. The user or client request first goes to the relying party, which sends a SAML request to its issuing authority. The issuing authority responds with a SAML assertion, which supplies the relying party with the requested security information. Two-way arrows denote the client communication with the relying party (there can be more than one relying party), and also denote the request-response communication between the relying party and issuing authority.

Finally, the box for the issuing authority separates out the assertion layer (SAML) from the transport layer (HTTP, SOAP, and so on) to show that the communication between these layers enables the issuing authority to create and validate assertions.

1.4.2.2 About SAML Protocol Bindings and Profiles

SAML defines a protocol, SAMLP, for requesting and obtaining assertions. Bindings define the standard way that SAML request and response messages are transported between the issuing authorities (identity providers) and relying parties (identity providers) by providing mappings between SAML messages and standard communication protocols. For example, the defined transport mechanism for SAML requests and responses is Simple Object Access Protocol (SOAP) over HTTP. This enables the exchange of SAML information across several Web services in a standard manner.

A profile describes how SAML assertion and protocol messages are combined with particular transport bindings to achieve a specific practical use case. Among the most widely-implemented SAML profiles, for example, are Web browser profiles for single sign-on and SOAP profiles for securing SOAP payloads.

1.4.2.3 How SAML Integrates with XML Security

In addition, SAML was designed to integrate with XML Signature and XML Encryption, standards from the World Wide Web Consortium for embedding encrypted data or digital signatures within an XML document. This support for XML signatures allows SAML to handle not only authentication, but also message integrity and nonrepudiation of the sender. See Oracle XML Security for more information about Oracle XML Security.

1.5 About Identity Federation

As global businesses strive for ever-closer relationships with suppliers and customers, they face challenges in creating more intimate, yet highly secure business transactions. Parties conducting a business transaction must be certain of the identity of the person or agent with whom they are dealing; they must also be assured that the other has the authority to act on behalf of the business with whom the transaction is being conducted. Federated Identity Management, makes parties establish trust relationships that allow one party to recognize and rely upon security tokens issued by another party. Identity Federation addresses challenges such as complexity, cost control, enabling secure access to resources for employees and customers, and regulatory compliance.

Historically, in the course of doing business with partners, companies have resorted to acquiring names, responsibilities, and other pertinent information about all entities who might act on behalf of the partner company. With changing roles and responsibilities, and particularly in large enterprises, this can create significant logistical problems as the data quickly becomes very costly to maintain and manage.

Key federation concepts include:

- **Principal** - the key actor in a federated environment, being an entity that performs an authorized business task

- **Identity Provider** - a service that authenticates a Principal's identity

- **Service Provider** - an entity that provides a service to a principal or another entity. For example, a travel agency can act as a Service Provider to a partner's employees (principals).

- **Single Sign-on** - the Principal's ability to authenticate with one system entity (the Identity Provider), and have other entities (the Service Providers) honor that authentication

> **Note:**
>
> For additional information about the standards mentioned here, see References.

1.6 About Oracle Security Developer Tools

Oracle Security Developer Tools are java tools that enable you implement a wide range of security tasks and projects by using the cryptography standards and protocols.

This section contains the topics:

- Understanding Toolkit Architecture
- Tools for XML, SAML, and Web Services Security Applications
- Tools for Public Key Cryptography (PKI) Applications
- Tools for E-mail Security Applications
- Tools for Low-level Cryptographic Applications
- Tools for Web Tokens

1.6.1 Understanding Toolkit Architecture

The Oracle Security Developer Tools consists of tools for XML, SAML, and Web Services Security Applications, tools for Public Key Cryptography (PKI) Applications, tools for E-mail Security Applications, tools for Low-level Cryptographic Applications, and tools for Web Tokens arranged across different layers of the setup.

It is useful to consider the tools in the toolkit as a whole, and then to look at functional subsets of tools for different applications.

Figure 1-2   The Oracle Security Developer Tools
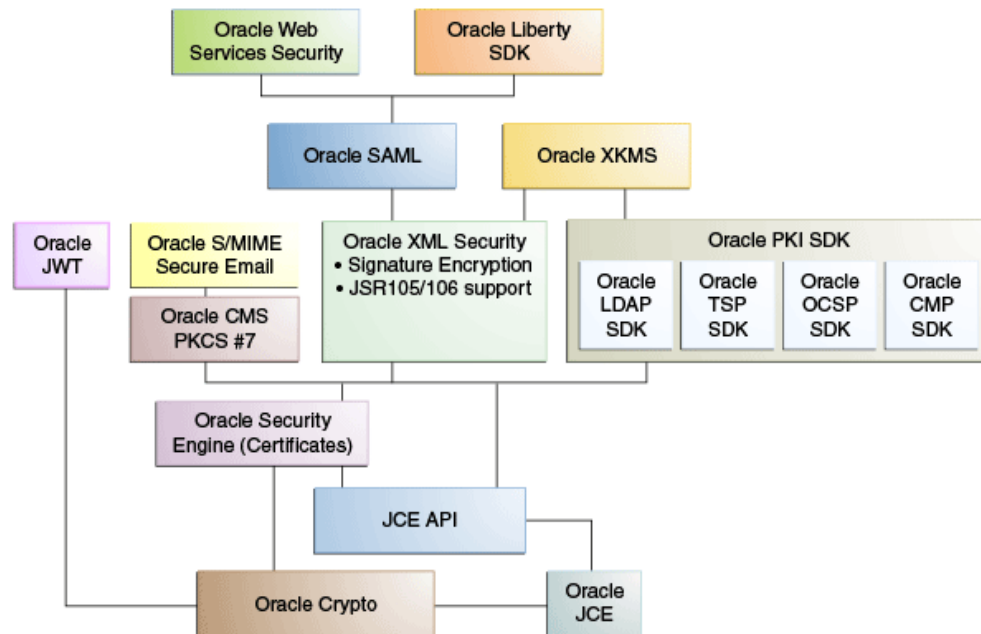
[Figure 1-2](#) shows the components of the Oracle Security Developer Tools. Typically, a tool will utilize functions provided by the tool immediately below it in the stack. For example, the Oracle SAML tool leverages functions provided by the Oracle XML Security tool.

Note that:

- Conceptually, the tools are arranged in layers with the fundamental building blocks at the bottom layer; each additional layer utilizes and builds upon the layer immediately below, to provide tools for specific security applications.

- The figure is not intended as a hierarchy or sequence diagram. Rather, it illustrates the relationship among components and the progression from low-level tools to more specialized and application-specific components higher up the stack.

Oracle Crypto and Oracle Security Engine are the basic cryptographic tools of the set. The next layer consists of Oracle CMS for message syntax, Oracle XML Security for signature encryption, and Oracle PKI SDK, which is a suite of PKI tools consisting of Oracle PKI LDAP SDK, Oracle PKI TSP SDK, Oracle PKI OCSP SDK, and Oracle PKI CMP SDK. Oracle S/MIME exploits Oracle CMS to provide a toolset for secure e-mail. The next layer contains Oracle SAML and Oracle Liberty SDK, which provides structured assertion markup and federated identity management capabilities. Finally, Oracle Web Services Security facilitates secure interactions with web services.

1.6.2 Tools for XML, SAML, and Web Services Security Applications

Oracle XML Security package provides security for XML documents. It provides the foundation for Oracle Web Services Security, Oracle SAML, and Oracle Liberty SDK.

The Oracle XML Security package provides the foundation for the following components of the toolkit:

- Oracle Web Services Security

- Oracle SAML for developing SAML 1.0 and 2.0-compliant Java security services

- Oracle Liberty SDK for single sign-on (SSO) and federated identity applications based on Liberty Alliance specifications

Figure 1-3   Tools for XML, SAML, and WS Security
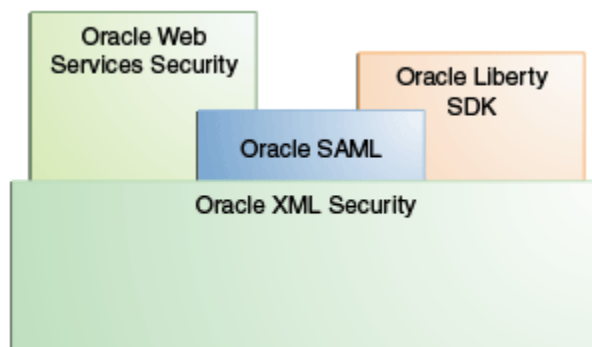
This graphic shows that Oracle SAML, Oracle Web Services Security, and Oracle Liberty tools are built on Oracle XML Security.

> **Note:**
>
> A diagram like this is necessarily simplified; in practice the jar relationships between the Oracle Security Developer Tools are complex and depend upon implementation details. For example, to use the SAML libraries, you actually need several components:
>
> - The Oracle XML Security library is needed as SAML requires signatures.
>
> - Oracle Security Engine provides certificate and CRL management features
>
> See Figure 1-2 for a more complete picture of dependencies. See the subsequent tool chapters in this guide for instructions on setting up the classpath for each tool, so that you have the correct environment for each type of application.

1.6.2.1 About Oracle XML Security

XML Security refers to the common data security requirements of XML documents, such as confidentiality, integrity, message authentication, and non-repudiation.

Oracle XML Security fulfills these needs by providing the following features:

- Support for the Decryption Transform proposed standard

- Support for the XML Canonicalization standard

- Support for the Exclusive XML Canonicalization standard

- Compatibility with a wide range of JAXP 1.1 compliant XML parsers and XSLT engines

1.6.2.2 About Oracle SAML

The Oracle SAML API provides tools and documentation to assist developers of SAML-compliant Java security services. You can integrate Oracle SAML into existing Java solutions including applets, applications, EJBs, servlets, and JSPs.

Oracle SAML provides the following features:

- Support for the SAML 1.0/1.1 and 2.0 specifications
- Support for SAML-based single sign-on (SSO), Attribute, Metadata, Enhanced Client Proxy, and federated identity profiles

1.6.2.3 About Oracle Web Services Security

Oracle Web Services Security provides an authentication and authorization framework based on Organization for the Advancement of Structured Information Standards (OASIS) specifications. Oracle Web Services Security provides the following features:

- Support for the SOAP Message Security standard (SOAP 1.1, 1.2)
- Support for the Username Token Profile standard (UsernameToken Profile 1.1)
- Support for the X.509 Certificate Token Profile standard
- Support for the WSS SAML Token Profile (version 1.0)

> **Note:**
>
> The WSS SAML Token Profile version is different from the SAML version.

1.6.2.4 About Oracle Liberty SDK

Oracle Liberty SDK allows Java developers to design and develop single sign-on (SSO) and federated identity solutions based on the Liberty Alliance specifications. Oracle Liberty SDK, available in versions 1.1 and 1.2, aims to unify, simplify, and extend all aspects of development and integration of systems conforming to the Liberty Alliance 1.1 and 1.2 specifications.

Oracle Liberty SDK provides the following features:

- Support for the Liberty Alliance Project version 1.1 and 1.2 specifications
- Support for Liberty-based Single Sign-on and Federated Identity

> **Note:**
>
> For additional information about the standards and specifications mentioned in this chapter, see References.

1.6.3 Tools for Public Key Cryptography (PKI) Applications

The Oracle PKI package consists of tools for working with digital certificates within an LDAP repository, for developing timestamp services conforming to RFC 3161, for OCSP messaging compliant with RFC 2560, and for the certificate management protocol (CMP) specification. The Oracle PKI package also provides the foundation for Oracle XKMS, which enables you to develop XML transactions for digital signature processing.

Figure 1-4   PKI Tools
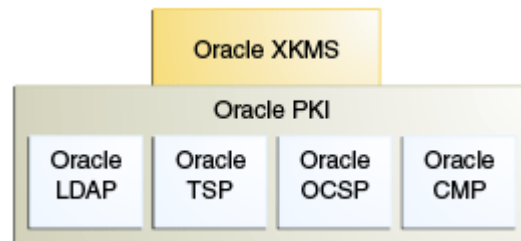
This graphic shows that Oracle's XKMS tool is built on Oracle PKI tools, which consist of Oracle LDAP, Oracle TSP, Oracle OCSP, and Oracle CMP.

1.6.3.1 About Oracle PKI LDAP SDK

Oracle PKI LDAP SDK provides facilities for accessing a digital certificate within an LDAP directory. Some of the tasks you can perform using the Oracle PKI LDAP SDK are:

- Validating a user's certificate in an LDAP directory
- Adding a certificate to an LDAP directory
- Retrieving a certificate from an LDAP directory
- Deleting a certificate from an LDAP directory

1.6.3.2 About Oracle PKI TSP SDK

The Oracle PKI TSP SDK provides the following features and functionality:

- Oracle PKI TSP SDK conforms to RFC 3161 and is compatible with other products that conform to this time stamp protocol (TSP) specification.
- Oracle PKI TSP SDK provides an example implementation of a TSA server to use for testing TSP request messages, or as a basis for developing your own time stamping service.

1.6.3.3 About Oracle PKI OCSP SDK

The Oracle PKI OCSP SDK provides the following features and functionality:

- The Oracle PKI OCSP SDK conforms to RFC 2560 and is compatible with other products that conform to this specification, such as Valicert's Validation Authority.

- The Oracle PKI OCSP SDK API provides classes and methods for constructing OCSP request messages that can be sent through HTTP to any RFC 2560 compliant validation authority.

- The Oracle PKI OCSP SDK API provides classes and methods for constructing responses to OCSP request messages, and an OCSP server implementation that you can use as a basis for developing your own OCSP server to check the validity of certificates you have issued.

1.6.3.4 About Oracle PKI CMP SDK

Certificate management protocol (CMP) messages support the following set of functions:

- Registration of an entity, which takes place prior to issuing a certificate

- Initialization, such as the generation of a key pair

- Certification (issuing certificates)

- Key pair recovery for reissuing lost keys

- Key pair updates when a certificate expires and a new key pair and certificate needs to be generated

- Revocation requests to the CA to include a certificate in a CRL

- Cross-certification between two CAs

The Oracle PKI CMP SDK conforms to RFC 2510 and is compatible with other products that conform to this certificate management protocol specification. In addition, it conforms to RFC 2511 and is compatible with other products that conform to this certificate request message format (CRMF) specification.

1.6.3.5 About Oracle XKMS

Oracle XKMS (XML Key Management Specification) provides a convenient way to handle public key infrastructures by allowing developers to write XML transactions for digital signature processing. Oracle XKMS implements the W3C XKMS standard and avoids some of the cost and complexity involved with public key infrastructures.

1.6.4 Tools for E-mail Security Applications

Oracle CMS provides tools for reading and writing CMS objects, as well as the foundation for the Oracle S/MIME tools for e-mail security, including certificate parsing and verification, X.509 certificates, private key encryption, and related features.

Figure 1-5   CMS and S/MIME Tools
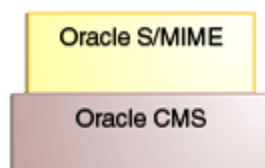
This graphic shows that Oracle's S/MIME tool is built on Oracle CMS.

1.6.4.1 About Oracle CMS

Oracle CMS provides an extensive set of tools for reading and writing CMS objects, and supporting tools for developing secure message envelopes.

Oracle CMS implements the IETF Cryptographic Message Syntax specified in RFC-2630. Oracle CMS implements all the RFC-2630 content types.

1.6.4.2 About Oracle S/MIME

Oracle S/MIME provides the following Secure/Multipurpose Internet Mail Extension (S/MIME) features:

- Full support for X.509 Version 3 certificates with extensions, including certificate parsing and verification

- Support for X.509 certificate chains in PKCS#7 and PKCS#12 formats

- Private key encryption using PKCS#5, PKCS#8, and PKCS#12

- An integrated ASN.1 library for input and output of data in ASN.1 DER/BER format

1.6.5 Tools for Low-level Cryptographic Applications

Oracle Crypto provides a broad range of cryptographic algorithms, message digests, and MAC algorithms, as well as the basis for the Oracle Security Engine for X.509 certificates and CRL extensions.

Figure 1-6   Cryptographic Tools
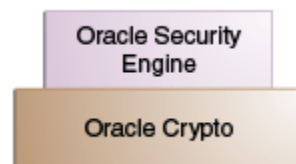
This graphic shows that Oracle Security Engine is built upon the Oracle Crypto tool.

1.6.5.1 About Oracle Crypto

The Oracle Crypto toolkit provides the following features:

- Public key cryptography algorithms such as RSA

- Digital signature algorithms such as Digital Signature Algorithm (DSA) and RSA

- Key exchange algorithms such as Diffie-Hellman

- Symmetric cryptography algorithms such as Blowfish, AES, DES, 3DES, RC2, and RC4

- Message digest algorithms such as MD2, MD4, MD5, SHA-1, SHA-256, SHA-384, and SHA-512

- MAC algorithms such as HMAC-MD5 and HMAC-SHA-1
- Methods for building and parsing ASN.1 objects

1.6.5.2 About Oracle Security Engine

The Oracle Security Engine toolkit provides the following features:

- X.509 Version 3 Certificates, as defined in RFC 3280
- Full PKCS#12 support
- PKCS#10 support for certificate requests
- CRLs as defined in RFC 3280
- Implementation of Signed Public Key And Challenge (SPKAC)
- Support for X.500 Relative Distinguished Name
- PKCS#7 support for wrapping X.509 certificates and CRLs
- Implementation of standard X.509 certificates and CRL extensions

1.6.6 Tools for Web Tokens

Oracle JWT enables you to create a JSON object that is digitally signed using a JSON Web Signature (JWS) and optionally encrypted using JSON Web Encryption (JWE).

1.6.6.1 About Oracle JWT

Oracle JWT (JSON Web Token) provides support for the JSON Web Token standard. Using Oracle JWT, you can construct and maintain JSON objects to represent claims being transferred between parties using a compact token format.

1.7 About Supported Standards

Oracle Security Developer Tools support multiple standards for SAML, XML Security Transforms, and WS-Security.

The supported standards and protocols are shown in the following table:

Table 1-2   Supported Standards

Feature/Component	Standard
SAML	- SAML 1.0 - SAML 1.1 - SAML 2.0
XML Security Transforms	The following transforms are supported: - canonicalization 1.0 - canonicalization 1.1 - exclusive canonicalization - decrypt transform - xpath filter transform - xpath filter 2.0 transform - enveloped signature transform

Table 1-2   (Cont.) Supported Standards

Feature/Component	Standard
WS-Security	WS-Security 1.1, including: • WS-Security Core Specification 1.1 • Username Token Profile 1.1 • X.509 Token Profile 1.1 • SAML Token profile 1.1 • Kerberos Token Profile 1.1 • SOAP with Attachments (SWA) Profile 1.1

> **Note:**
>
> By way of clarification, note that SAML token profile 1.1 applies to SAML 2.0, while SAML token profile 1.0 applies to SAML 1.0 and SAML 1.1.

1.8 Setting the CLASSPATH Environment Variable

Each tool in the OSDT toolkit has specific `CLASSPATH` requirements. You must set the `CLASSPATH` environment variable. Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files.

To determine which jars you need for a specific OSDT tool, refer the Setting Up Your Environment section of the chapter that describes the tool.

1.8.1 Setting the CLASSPATH on Windows

On Windows, set your `CLASSPATH` environment variable to include the full path and file name of all the required jar files, by using the Windows Control Panel.

To set the `CLASSPATH` on Windows:

1. In your Windows Control Panel, select **System**.

2. In the System Properties dialog, select the **Advanced** tab.

3. Click **Environment Variables**.

4. In the User Variables section, click **New** to add a `CLASSPATH` environment variable for your user profile. If a `CLASSPATH` environment variable already exists, select it and click **Edit**.

5. Add the full path and file names for all the required jar files to the `CLASSPATH`.

   For example, your `CLASSPATH` might look like this:

   ```
   %CLASSPATH%;%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
   %ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
   %ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_xmlsec.jar;
   %ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_saml.jar;
   %ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_saml2.jar;
   %ORACLE_HOME%\modules\org.jaxen_1.1.1.jar;
   ```

6. Click **OK**.

1.8.2 Setting the CLASSPATH on UNIX

On UNIX, set your `CLASSPATH` environment variable to include the full path and file name of all the required jar and class files.

For example:

```
setenv CLASSPATH $CLASSPATH:$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_core.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cert.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_xmlsec.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_saml.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_saml2.jar:
$ORACLE_HOME/modules/org.jaxen_1.1.1.jar
```

2
Oracle Crypto

Oracle Crypto Software Development Kit (SDK) allows Java developers to create applications that ensure data security and integrity.

> **Note:**
>
> The use of the Oracle Crypto library is not recommended with Release 11gR1 and higher. Instead, use the standard JCE interface for all cryptographic operations.
>
> However, for ASN.1 parsing you should continue to use the Oracle Crypto library, as there are no standard APIs in the JDKs for that task.
>
> For more information, see these resources:
>
> *   JDK documentation on using the JCE interfaces at `http://www.oracle.com/technetwork/java/index.html`
> *   Migrating to the JCE Framework

This chapter contains the following topics:

*   About Oracle Crypto Features and Benefits
*   Setting Up Your Oracle Crypto Environment
*   Understanding and Using Core Classes and Interfaces of Oracle Crypto
*   The Oracle Crypto and Crypto FIPS Java API References

2.1 About Oracle Crypto Features and Benefits

Oracle Crypto supports public key cryptography algorithms, digital signature algorithms, key exchange algorithms, symmetric cryptography algorithms, message digest algorithms, MAC algorithms, and methods for building and parsing ASN.1 objects.

Oracle Crypto provides the following features:

*   Public key cryptography algorithms such as RSA
*   Digital signature algorithms such as DSA and RSA
*   Key exchange algorithms such as Diffie-Hellman
*   Symmetric cryptography algorithms such as Blowfish, AES, DES, 3DES, RC2, and RC4
*   Message digest algorithms such as MD2, MD4, MD5, SHA-1, SHA-256, SHA-384, and SHA-512
*   MAC algorithms such as HMAC-MD5 and HMAC-SHA-1

- Methods for building and parsing ASN.1 objects

2.2 About the Oracle Crypto Packages

Oracle Crypto contains packages of basic cryptographic primitives, utility classes for handling mathematical functions, various other utility classes, and facilities for reading and writing both BER-encoded and DER-encoded ASN.1 structures.

Oracle Crypto contains the following packages:

- `oracle.security.crypto.core` - Basic cryptographic primitives
- `oracle.security.crypto.core.math` - Utility classes for handling mathematical functions
- `oracle.security.crypto.util` - Various utility classes
- `oracle.security.crypto.asn1` - Facilities for reading and writing both BER-encoded and DER-encoded ASN.1 structures

2.3 Setting Up Your Oracle Crypto Environment

In order to use the Oracle Crypto SDK, your system must have the Java Development Kit (JDK) version 1.6 or higher. Your `CLASSPATH` environment variable must contain the full path and file names to the required jar and class files.

Make sure that the `osdt_core.jar` file is included in your `CLASSPATH`.

For example, your `CLASSPATH` might look like this:

```
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar
```

> ✎ **See Also:**
>
> Setting the CLASSPATH Environment Variable

2.4 Understanding and Using Core Classes and Interfaces of Oracle Crypto

Oracle Crypto consists of multiple core classes and interfaces in the categories of Key Classes, Key Generation Classes, Cipher Classes, Signature Classes, Message Digest Classes, Key Agreement Class, and Pseudo-Random Number Generator Classes.

This section provides information and code samples for using the core classes and interfaces of Oracle Crypto. The following sections explain it further:

- About Oracle Crypto Key Classes
- Using the Oracle Crypto Key Generation Classes
- Using Oracle Crypto Cipher Classes
- Using the Oracle Crypto Signature Classes

- Using Oracle Crypto Message Digest Classes
- Using the Oracle Crypto Key Agreement Class
- Using Oracle Crypto Pseudo-Random Number Generator Classes

2.4.1 About Oracle Crypto Key Classes

Oracle Crypto provides multiple classes and interfaces to work with keys.

These classes and interfaces are:

- The oracle.security.crypto.core.Key Interface
- The oracle.security.crypto.core.PrivateKey Interface
- The oracle.security.crypto.core.PublicKey Interface
- The oracle.security.crypto.core.SymmetricKey Class

2.4.1.1 The oracle.security.crypto.core.Key Interface

This interface represents a key which may be used for encryption or decryption, for generating or verifying a digital signature, or for generating or verifying a MAC. A key may be a private key, a public key, or a symmetric key.

2.4.1.2 The oracle.security.crypto.core.PrivateKey Interface

This interface represents a private key which may be an `RSAPrivateKey`, a `DSAPrivateKey`, a `DHPrivateKey`, an `ECPrivateKey` or a `PrivateKeyPKCS8` instance that holds an encrypted private key.

2.4.1.3 The oracle.security.crypto.core.PublicKey Interface

This interface represents a public key which may be a `RSAPublicKey`, a `DSAPublicKey`, a `DHPublicKey` or a `ECPublicKey` instance.

2.4.1.4 The oracle.security.crypto.core.SymmetricKey Class

This class represents a symmetric key which may be used for encryption, decryption or for MAC operations.

2.4.2 Using the Oracle Crypto Key Generation Classes

Oracle Crypto provides classes for key generation.

These classes are:

- Using the oracle.security.crypto.core.KeyPairGenerator Class
- Using the oracle.security.crypto.core.SymmetricKeyGenerator Class

2.4.2.1 Using the oracle.security.crypto.core.KeyPairGenerator Class

This abstract class is used to generate key pairs such as RSA, DSA, Diffie-Hellman or ECDSA key pairs.

To get a new key pair generator, create a new instance of `KeyPairGenerator` by calling the static `getInstance()` method with an `AlgorithmIdentifier` object as a parameter. This example shows how to create a new `KeyPairGenerator` instance:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance(AlgID.rsaEncryption);
```

This creates a `KeyPairGenerator` object from one of the concrete classes: `RSAKeyPairGenerator`, `DSAKeyPairGenerator`, `DHKeyPairGenerator`, or `ECKeyPairGenerator`.

Initialize the key pair generator by using one of the `initialize()` methods. Generate the key pair with the `generateKeyPair()` method. This example shows how to initialize the key pair generator and then generate a key pair:

```
kpg.initialize(1024, RandomBitsSource.getDefault());
KeyPair kp = kpg.generateKeyPair();
PrivateKey privKey = kp.getPrivate();
PublicKey pubKey = kp.getPublic();
```

Save the keys using the `output()` method, or in the case of the private key, encrypt it and save it using the `PrivateKeyPKCS8` class. This example shows how to save a key pair.

```
FileOutputStream pubKeyFos = new
FileOutputStream("my-pub-key.der");
pubKey.output(pubKeyFos);
pubKeyFos.close();

PrivateKeyPKCS8 privKeyPKCS8 =
    new PrivateKeyPKCS8(privKey, "myPassword");
FileOutputStream privKeyFos =
    new FileOutputStream("my-encrypted-priv-key.der");
privKeyPKCS8.output(privKeyFos);
privKeyFos.close();
```

2.4.2.2 Using the oracle.security.crypto.core.SymmetricKeyGenerator Class

This class generates symmetric key pairs such as Blowfish, DES, 3DES, RC4, RC2, AES, and HMAC keys.

To get a new symmetric key generator, create a new instance of `SymmetricKeyGenerator` by calling the static `getInstance()` method with an `AlgorithmIdentifier` object as a parameter. This example shows how to create a new `SymmetricKeyGenerator` instance:

```
SymmetricKeyGenerator skg = SymmetricKeyGenerator.getInstance(AlgID.desCBC);
```

Generate the key pair with the `generateKey()` method. You can then save the key by using the `getEncoded()` method. This example shows how to generate and save a symmetric key pair.

```
SymmetricKey sk = skg.generateKey();

FileOutputStream symKeyFos =
    new FileOutputStream("my-sym-key.der");
symKeyFos.write(sk.getEncoded());
symKeyFos.close();
```

2.4.3 Using Oracle Crypto Cipher Classes

Oracle Crypto provides classes for symmetric ciphers, RSA cipher, and methods for password based encryption.

The Oracle Crypto Cipher classes and interfaces are divided into the following categories:

- Using Symmetric Ciphers
- Using the RSA Cipher
- Using Password Based Encryption (PBE)

2.4.3.1 Using Symmetric Ciphers

The symmetric ciphers are made up of two categories: the block ciphers (such as Blowfish, DES, 3DES, RC2, and AES) and the stream ciphers (such as RC4).

A symmetric cipher can be used for four types of operations:

- Encryption of raw data. Use one of the `encrypt()` methods by passing data to be encrypted.
- Decryption of encrypted data. Use one of the `decrypt()` methods by passing encrypted data to be decrypted.
- Wrapping of private or symmetric keys. Use one of the `wrapKey()` methods by passing the private or symmetric key to be encrypted.
- Unwrapping of private or symmetric encrypted keys. Use either the `unwrapPrivateKey()` or the `unwrapSymmetricKey()` method by passing the encrypted private or symmetric key to be decrypted.

The concrete block cipher classes extend the abstract `oracle.security.crypto.core.BlockCipher` class, which extends the `oracle.security.crypto.core.Cipher` class. The stream cipher classes directly extend the `oracle.security.crypto.core.Cipher` class.

To create a new instance of `Cipher`, call the static `getInstance()` method with an `AlgorithmIdentifier` and a `Key` object as parameters.

This example shows how to create a new Cipher instance. First an RC4 object is created and initialized with the specified key. Second a block cipher DES object is created and initialized with the specified key and padding. This creates a cipher and initializes it with the passed parameters. To re-initialize an existing cipher, call one of the `initialize()` methods.

```
Cipher rc4 = Cipher.getInstance(AlgID.rc4, rc4SymKey);

Cipher desCipher = Cipher.getInstance(AlgID.desCBC, desSymKey, Padding.PKCS5);
```

When using CBC ciphers, the `AlgorithmIdentifier` object may hold cryptographic parameters such as the initialization vector (IV) or the effective key length for RC2 ciphers. To specify these parameters when creating or initializing block ciphers, build a `CBCAlgorithmIdentifier` object or `RC2AlgorithmIdentifier` object with the cryptographic parameters. This example shows how to create and initialize a CBC cipher and a RC2 cipher.

```
CBCAlgorithmIdentifier cbcAlgID =
    new CBCAlgorithmIdentifier(AlgID.desCBC, iv);
desCipher.initialize(cbcAlgID, desSymKey, Padding.PKCS5);
RC2AlgorithmIdentifier rc2AlgID =
    new RC2AlgorithmIdentifier(iv, 56);
BlockCipher rc2Cipher =
    (BlockCipher)Cipher.getInstance(rc2AlgID, rc2SymKey, Padding.PKCS5);
```

2.4.3.2 Using the RSA Cipher

The RSA cipher is an implementation of PKCS#1 v2.0 that supports the RSAES-OAEP and RSAES-PKCS1-v1_5 encryption schemes. According to the specification, RSAES-OAEP is recommended for new applications, and RSAES-PKCS1-v1_5 is included only for compatibility with existing applications and protocols.

The encryption schemes are used to combine RSA encryption and decryption primitives with an encoding method. Encryption and decryption can only be done through the methods `encrypt(byte[])` and `decrypt(byte[])`.

You can use an RSA cipher for four types of operations:

- Encryption of raw data. Use one of the `encrypt()` methods by passing data to be encrypted.

- Decryption of encrypted data. Use one of the `decrypt()` methods by passing encrypted data to be decrypted.

- Wrapping of keys. Use the `wrapKey()` method by passing the key to be encrypted.

- Unwrapping of encrypted keys. Use the `unwrapSymmetricKey()` method by passing the encrypted key to be decrypted.

To create a new instance of `Cipher`, call the static `getInstance()` method with `AlgorithmIdentifier` and `Key` objects as parameters. This example demonstrates how to create an `RSApkcs1` object and initialize it with the specified key. The cipher can then be used to encrypt or decrypt data.

```
Cipher rsaEnc = Cipher.getInstance(AlgID.rsaEncryption, pubKey);
byte[] encryptedData = rsaEnc.encrypt(data);
Cipher rsaDec = Cipher.getInstance(AlgID..rsaEncryption, privKey);
byte[] decryptedData = rsaDec.decrypt(encryptedData);
```

When using RSA ciphers, the `AlgorithmIdentifier` object may hold cryptographic parameters such as the mask generation function for RSAES-OAEP. To specify these parameters when creating or initializing RSA ciphers, build an `OAEPAlgorithmIdentifier`, or use the default one located in the `oracle.security.crypto.core.AlgID` interface.

2.4.3.3 Using Password Based Encryption (PBE)

The abstract `oracle.security.crypto.core.PBE` class provides methods for Password Based Encryption (PBE) operations. The concrete classes extending the PBE are the `PKCS5PBE` and `PKCS12PBE` classes.

You may use a PBE object for four types of operations:

- Encryption of raw data. For example:

```
byte[] encData = pbeEnc.encrypt("myPassword", data);
```

- Decryption of encrypted data. For example:

```
byte[] decData = pbeDec.decrypt("myPassword", encData);
```

- Wrapping of private or symmetric keys. For example:

```
byte[] encPrivKey = pbeEnc.encryptPrivateKey("myPassword", privKey);
byte[] encSymKey = pbeEnc.encryptSymmetricKey("myPassword", symKey);
```

- Unwrapping of private or symmetric encrypted keys. For example:

```
PrivateKey decPrivKey = pbeDec.decryptPrivateKey("myPassword", encPrivKey);
SymmetricKey decSymKey = pbeDec.decryptSymmetricKey("myPassword", encSymKey);
```

To create a new instance of PBE, call the static `getInstance()` method with a `PBEAlgorithmIdentifier` object as a parameter. For example:

```
PBE pbeEnc = PBE.getInstance(pbeAlgID);
```

This will create a `PKCS5PBE` object and initialize it with the specified PBE algorithm. The PBE can then be used to encrypt or decrypt data, wrap or unwrap keys.

When using PBE objects, the `AlgorithmIdentifier` object may hold cryptographic parameters such as the salt or the iteration count as well as the ASN.1 Object Identifier specifying the PBE algorithm to use. To specify these parameters when creating or initializing PBEs, build a `PBEAlgorithmIdentifier` object with the cryptographic parameters.

Here is an example of creating a PBE object:

```
PBEAlgorithmIdentifier pbeAlgID =
    new PBEAlgorithmIdentifier(PBEAlgorithmIdentifier.pbeWithMD5AndDES_CBC, salt,
1024);
pbeEnc.initialize(pbeAlgID);
PBE pbeDec = PBE.getInstance(pbeAlgID);
```

2.4.4 Using the Oracle Crypto Signature Classes

The `oracle.security.crypto.core.Signature` abstract class provides methods to sign and verify signatures. The concrete classes extending the `Signature` class are the `RSAMDSignature`, `DSA` and the `ECDSA` classes.

The algorithms available for signature operations are:

- For RSA: `AlgID.md2WithRSAEncryption`, `AlgID.md5WithRSAEncryption` and `AlgID.sha_1WithRSAEncryption`

- For DSA: `AlgID.dsaWithSHA1`

- For ECDSA: `AlgID.ecdsaWithSHA1`

To create a new instance of `Signature`, call the static `getInstance()` method with an `AlgorithmIdentifier` and a `PrivateKey` or `PublicKey` objects as parameters. This example shows how to create a new `Signature` object and initialize it with the specified algorithm.

```
Signature rsaSign = Signature.getInstance(AlgID.md5WithRSAEncryption);
Signature rsaVerif = Signature.getInstance(AlgID.md5WithRSAEncryption);
```

This example shows how to set the keys for the `Signature` objects and set the document to be signed or verified.

```
rsaSign.setPrivateKey(privKey);
rsaSign.setDocument(data);
rsaVerif.setPublicKey(pubKey);
rsaVerif.setDocument(data);
```

This example shows how to compute the signature using the private key or to verify the signature using the public key and the signature bytes.

```
byte[] sigBytes = rsaSign.sign();
boolean verified = rsaVerif.verify(sigBytes);
```

2.4.5 Using Oracle Crypto Message Digest Classes

Oracle Crypto contains message digest classes to hash, digest and compute data.

Oracle Crypto provides the following message digest classes:

- Using the oracle.security.crypto.core.MessageDigest Class
- Using the oracle.security.crypto.core.MAC Class

2.4.5.1 Using the oracle.security.crypto.core.MessageDigest Class

The `MessageDigest` abstract class provides methods to hash and digest data. The concrete classes extending the `MessageDigest` class are the `MD2`, `MD4`, `MD5` and the `SHA` classes.

The available algorithms for message digest operations are: `AlgID.md2`, `AlgID.md4`, `AlgID.md5`, `AlgID.sha_1`, `AlgID.sha_256`, `AlgID.sha_384` and `AlgID.sha_512`.

The basic process for creating a message digest is as follows:

1. Create a new instance of `MessageDigest` by calling the static `getInstance()` method with an `AlgorithmIdentifier` object as a parameter.

2. Add the data to be digested.

3. Compute the hash value.

This example shows how to create an MD5 message digest object.

```
//Create a new MD5 MessageDigest object
MessageDigest md5 = Signature.getInstance(AlgID.md5);

//Add the data to be digested
md5.udpate(data1);
md5.udpate(data2);

//Compute the hash value
md5.computeCurrent();
byte[] digestBits = md5.getDigestBits();
```

2.4.5.2 Using the oracle.security.crypto.core.MAC Class

The `MAC` abstract class provides methods to compute and verify a Message Authentication Code (MAC). The concrete class extending the `MAC` is the `HMAC` class.

The available algorithms for MAC operations are: `AlgID.hmacMD5` and `AlgID.hmacSHA`.

The basic process for creating a MAC is as follows:

1. Create a new instance of `MAC` by calling the static `getInstance()` method with an `AlgorithmIdentifier` and a `SymmetricKey` object as a parameter.

2. Add the data to be digested.

3. Compute the MAC value and verify it.

This example shows how to create a new HMAC object with the HMAC-SHA1 algorithm.

```
//Create an HMAC object with the HMAC-SHA1 algorithm
MAC hmacSha1Compute = MAC.getInstance(AlgID.hmacSHA, hmacSha1Key);

//Add the data to be digested
hmacSha1Compute.udpate(data);

//Compute the MAC value and verify
byte[] macValue = hmacSha1Compute.computeMAC();
boolean verified = hmacSha1Verify.verifyMAC(data, macValue);
```

2.4.6 Using the Oracle Crypto Key Agreement Class

The `oracle.security.crypto.core.KeyAgreement` class abstract class provides methods for public key agreement schemes such as Diffie-Hellman. The concrete classes extending the `KeyAgreement` class are the `DHKeyAgreement` and the `ECDHKeyAgreement` classes.

The available algorithms for key agreement operations are: `AlgID.dhKeyAgreement` and `ECDHKeyAgreement` (Elliptic Curve Diffie-Hellman key agreement).

The basic process for key agreement is as follows:

1. Create a new instance of `KeyAgreement` by calling the static `getInstance()` method with an `AlgorithmIdentifier` object as a parameter.

2. Set the local private key and the other party's public key.

3. Compute the shared secret value.

This example shows how to perform key agreement.

```
//Create a DH key agreement object
KeyAgreement dh = KeyAgreement.getInstance(AlgID.dhKeyAgreement);

//Set the private key and public key
dh.setPrivateKey(privKey);
dh.setPublicKey(otherPubKey);

//Compute the shared secret
byte[] sharedSecret = dh.generateSecret();
```

2.4.7 Using Oracle Crypto Pseudo-Random Number Generator Classes

In cryptography, random numbers are used to generate keys. Cryptographic systems need cryptographically strong (pseudo) random numbers that cannot be guessed by an attacker. Oracle Crypto provides pseudo-random number generator (PRNG) classes.

These pseudo-random number generator (PRNG) classes are:

- • Using the oracle.security.crypto.core.RandomBitsSource class
- • Using the oracle.security.crypto.core.EntropySource class

2.4.7.1 Using the oracle.security.crypto.core.RandomBitsSource class

RandomBitsSource is an abstract class representing secure PRNG implementations. Note that, by the very nature of PRNGs, the security of their output depends on the amount and quality of seeding entropy used. Implementing classes should provide guidance as to their proper initialization and use. The concrete classes extending the RandomBitsSource are the MD5RandomBitsSource, SHA1RandomBitsSource, and the DSARandomBitsSource classes.

Create a new instance of RandomBitsSource by calling the static getDefault() method to return the default PRNG:

```
RandomBitsSource rbs = RandomBitsSource.getDefault();
```

A RandomBitsSource object can also be created by instantiating one of the subclasses:

```
RandomBitsSource rbs = new SHA1RandomBitsSource();
```

By default, a newly created PRNG created from a subclass will be seeded. To seed a generic RandomBitsSource object, use one of the seed methods by using a byte array or an EntropySource object:

```
rbs.seed(myByteArray);
```

The object is then ready to generate random data:

```
rbs.randomBytes(myRandomByteArray);
```

2.4.7.2 Using the oracle.security.crypto.core.EntropySource class

The EntropySource class provides a source of seed material for the PRNGs. The concrete classes extending the EntropySource are the SpinnerEntropySource and SREntropySource classes.

Create a new instance of EntropySource by calling the static getDefault() method to return the default entropy source:

```
EntropySource es = EntropySource.getDefault();
```

You can also create an EntropySource object by instantiating one of the subclasses:

```
EntropySource rbs = new SpinnerEntropySource();
```

The entropy source is readied for use by using one of the generateByte methods:

```
es.generateBytes(mySeedingArray);
```

2.5 The Oracle Crypto and Crypto FIPS Java API References

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods for Oracle Crypto and Oracle Crypto FIPS.

You can access this guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

3

Oracle Security Engine

Oracle Security Engine Software Development Kit (SDK) is a superset of Oracle Crypto. It contains all of the libraries and tools provided with Oracle Crypto, plus additional packages and utilities for generating digital certificates.

> **Note:**
>
> The use of the Oracle Security Engine library is not recommended with Release 11gR1 and higher. Instead use the JDK's Certificate APIs.
>
> For details, see the JDK documentation at:
>
> http://www.oracle.com/technetwork/java/index.html
>
> However, the following Public-Key Cryptography Standards (PKCS) have no JCE equivalents:
>
> - PKCS#7
> - PKCS#10
> - Signed Public Key And Challenge (SPKAC)
>
> and you can continue using Oracle Security Engine for these features.

Oracle Crypto allows Java developers to develop applications that ensure data security and integrity. For more information about the Oracle Crypto functionality, see " Oracle Crypto ".

For an overview of public key infrastructure, see "About Public Key Infrastructure (PKI)".

This chapter contains the following topics:

- Oracle Security Engine Features and Benefits
- Setting Up Your Oracle Security Engine Environment
- Core Classes and Interfaces of Oracle Security Engine
- The Oracle Security Engine Java API Reference

3.1 Oracle Security Engine Features and Benefits

Oracle Security Engine supports X.509, PKCS#10, and PKCS#12 certificates, along with RDN and CRLs. It contains packages to support handling of digital certificates, CRLs, and PKCS#12. The packages also handle Standard X.509 certificates and CRL extensions.

Oracle Security Engine provides the following features:

- X.509 Version 3 Certificates, as defined in RFC 3280

- Full PKCS#12 support

- PKCS#10 support for certificate requests

- certificate revocation list (CRL) functionality as defined in RFC 3280

- Implementation of Signed Public Key And Challenge (SPKAC)

- Support for X.500 Relative Distinguished Names

- PKCS#7 support for wrapping X.509 certificates and CRLs

- Implementation of standard X.509 certificates and CRL extensions

The Oracle Security Engine toolkit contains the following packages:

- `oracle.security.crypto.cert` - Facilities for handling digital certificates, CRLs, and PKCS#12.

- `oracle.security.crypto.cert.ext` - Standard X.509 certificates and CRL extensions.

3.2 Setting Up Your Oracle Security Engine Environment

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`. In order to use Oracle Security Engine, your system must have the Java Development Kit (JDK) version 1.6 or higher. Your `CLASSPATH` environment variable must contain the full path and file names to the required jar and class files.

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`

- `osdt_cert.jar`

For example, your `CLASSPATH` might look like this:

```
%CLASSPATH%;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
```

> ✏️ **See Also:**
>
> Setting the CLASSPATH Environment Variable

3.3 Core Classes and Interfaces of Oracle Security Engine

Oracle Security Engine also includes all of the classes provided with Oracle Crypto. It also includes multiple core certificate facility classes.

Class Changes in Release 11*g*R1

In Release 11*g*R1, the oracle.security.crypto.cert.X509 class for certificate management was replaced with java.security.cert.X509Certificate

The Core Certificate Classes

The core certificate facility classes are:

- Using the oracle.security.crypto.cert.X500RDN Class
- Using the oracle.security.crypto.cert.X500Name Class
- Using the oracle.security.crypto.cert.CertificateRequest Class
- Using the java.security.cert.X509Certificate Class

3.3.1 Using the oracle.security.crypto.cert.X500RDN Class

The oracle.security.crypto.cert.X500RDN class represents an X.500 Relative Distinguished Name (RDN). This is the building block for X.500 names. A RDN consists of a set of attribute-value pairs. Typically, there is a single attribute-value pair in each RDN.

```
// Create the X500RDN object
X500RDN rdn = new X500RDN(PKIX.id_at_commonName, "Joe Smith");

// Retrieve the value
X500Name n = Instance of oracle.security.crypto.cert.X500Name;
String name = n.getAttribute(PKIX.id_at_commonName).getValue().getValue();
```

3.3.2 Using the oracle.security.crypto.cert.X500Name Class

The oracle.security.crypto.cert.X500Name class represents distinguished names as used in the X.500 series of specifications, defined in X.520. An X500Name object is made of X500RDN objects. An X500Name holds attributes defining an entity such as the common name, country, organization, and so on.

To create an X500Name object, use the standard constructor and then populate the object with attributes. Once created, the object can then be DER-encoded to make it available to other processes:

```
X500Name name = new X500Name();
name.addComponent(PKIX.id_at_commonName, "Joe Smith");
name.addComponent(PKIX.id_at_countryName, "USA");
name.addComponent(PKIX.id_at_stateOrProvinceName, "NY");
name.addComponent(PKIX.id_at_localityName, "New York");
name.addComponent(PKIX.id_at_organizationName, "Oracle");
name.addComponent(PKIX.id_at_organizationalUnitName, "Engineering");
name.addComponent(PKIX.emailAddress, "joe.smith@example.com");

// Make object DER-encoded so its available to other processes

byte[] encodedName = Utils.toBytes(name);
X500Name n = new X500Name(new ByteArrayInputStream(encodedName));
String name = n.getAttribute(PKIX.id_at_commonName).getValue().getValue();
String email = n.getAttribute(PKIX.emailAddress).getValue().getValue();
```

3.3.3 Using the oracle.security.crypto.cert.CertificateRequest Class

The oracle.security.crypto.cert.CertificateRequest class represents a PKCS#10 certificate request containing information about an entity and a signature of the content of the request. The certificate request is used to convey information and authentication

data (the signature) that will be used by a Certificate Authority (CA) to generate a certificate for the corresponding entity.

Creating a new certificate request involves the following high-level steps:

1. Create a new instance of `CertificateRequest` by using the empty constructor and setting the keys and the subject name, or by using the constructor taking an `X500Name` and a `KeyPair` object.

2. Add X.509 extensions to the certificate request.

3. Sign the certificate request and save it to a file.

4. Send the certificate request you created to a Certificate Authority.

```
//Create CertificateRequest by setting the keys and subject name
 CertificateRequest certReq = new CertificateRequest();
 certReq.setPrivateKey(privKey);
certReq.setPublicKey(pubKey);
 certReq.setSubject(subjectName);

//OR

// Create CertificateRequest by taking an X500Name and KeyPair object
CertificateRequest certReq = new CertificateRequest(subjectName, keyPair);

// Add X.509 certificate extensions in a extensionRequest attribute
X509ExtensionSet extSet = new X509ExtensionSet();

// Basic Constraints: non-CA, critical
extSet.addExtension(new BasicConstraintsExtension(false, true));

// Key Usage: signature, data encipherment, key agreement
// & non-repudiation flags, critical
extSet.addExtension(new KeyUsageExtension(new int[] {
     KeyUsageExtension.DIGITAL_SIGNATURE,
     KeyUsageExtension.DATA_ENCIPHERMENT,
     KeyUsageExtension.KEY_AGREEMENT,
     KeyUsageExtension.NON_REPUDIATION},
        true));

// Subject Alternative Name: email address, non-critical
if (email.length() > 0)
    extSet.addExtension(new SubjectAltNameExtension(
        new GeneralName(GeneralName.Type.RFC822_NAME, email), false));

// Subject Key Identifier: key ID bytes, non-critical
extSet.addExtension(new SubjectKeyIDExtension
    (CryptoUtils.generateKeyID(kp.getPublic())));
req.addAttribute(PKIX.extensionRequest, extSet);

// Sign the certificate request and save to file
req.sign();
req.output(reqOS);
reqOS.close();
    }
// The certificate request can then be sent to a CA
```

3.3.4 Using the java.security.cert.X509Certificate Class

The java.security.cert.X509Certificate class supports the generation of new certificates as well as parsing of existing certificates.

> **Note:**
>
> This class replaces oracle.security.crypto.cert.X509 for X.509 certificate management in Oracle WebLogic Server 11*g*.

Complete documentation of the java.security.cert.X509Certificate class is available at http://www.oracle.com/technetwork/java/index.html.

Converting Your Code to Use java.security.cert.X509Certificate

You can create the X509Certificate object using the certificate factory java.security.cert.CertificateFactory.

The certificate is generated from an input stream, which can be:

- a FileInputSream, if the certificate is stored in a file, or
- a ByteArrayInputStream, if the encoded bytes are from an existing X509 object, or
- any other source.

An example follows:

```
// Generating an X.509 certificate from a file-based certificate

CertificateFactory cf = CertificateFactory.getInstance("X.509");

X509Certificate cert = (X509Certificate)cf.generateCertificate(
                        new FileInputStream(certFileName);

**
```

3.4 The Oracle Security Engine Java API Reference

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods of Oracle Security Engine.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

4
Oracle CMS

The Oracle CMS SDK is a pure Java API with an extensive set of tools for reading and writing CMS objects, sample programs, and supporting tools for developing secure message envelopes.

This chapter contains these topics:

- Oracle CMS Features and Benefits
- Setting Up Your Oracle CMS Environment
- Understanding and Developing Applications with Oracle CMS
- The Oracle CMS Java API Reference

4.1 Oracle CMS Features and Benefits

The Oracle CMS SDK is a pure Java API with an extensive set of tools for reading and writing CMS objects, sample programs, and supporting tools for developing secure message envelopes. It implements the IETF Cryptographic Message Syntax specified in RFC 2630. This syntax is used to digitally sign, digest, authenticate, and encrypt messages.

The Cryptographic Message Syntax is derived from PKCS #7 version 1.5 as specified in RFC 2315 [PKCS#7].

> ✎ **See Also:**
>
> References for a link to the specifications.

4.1.1 Content Types in Oracle CMS

Oracle CMS supports various content types including signed, enveloped, encrypted, and other data. It supports all the content types specified in RFC-2630. It supports the Enhanced Security Services for S/MIME content type specified in RFC-2634. It also supports IETF PKIX TimeStamp Protocol content type corresponding to RFC-3161.

Table 4-1   Content Types Supported by Oracle CMS

Type	Identifier
data	1.2.840.113549.1.7.1
signed-data	1.2.840.113549.1.7.2
enveloped-data	1.2.840.113549.1.7.3
digested-data	1.2.840.113549.1.7.5
encrypted-data	1.2.840.113549.1.7.6

Table 4-1   (Cont.) Content Types Supported by Oracle CMS

Type	Identifier
authenticated-data	1.2.840.113549.1.9.16.1.2

Oracle CMS is a full implementation of RFC-2630 with these exceptions:

- There is no support for Attribute Certificates
- There is no support for Key Agreement RecipientInfo

Oracle CMS supports the following Enhanced Security Services for S/MIME content type specified in RFC-2634:

Type	Identifier
receipt	1.2.840.113549.1.9.16.1.2

The following IETF PKIX TimeStamp Protocol content type corresponding to RFC 3161 is supported:

Type	Identifier
TSTInfo	1.2.840.113549.1.9.16.1.4

> **Note:**
>
> Oracle CMS will not process a content type other than the ones specified earlier.

A link to RFC 3161 is available in References.

4.1.2 Differences Between Oracle CMS Implementation and RFCs

Oracle CMS differs from PKCS #7 v1.5 [RFC 2315] and IETF CMS [RFC 2630] in certain ways. You must know these differences if you require interoperability with PKCS#7 implementations.

The following are the differences:

- The enveloped-data contains an optional OriginatorInfo.
- In RFC 2630 Enveloped data also contains optional unprotected attributes.
- The SignerIdentifier in the signed-data SignerInfo is a **choice** of IssuerAndSerialNo or SubjectKeyIdentifier.
- In RFC 2630 the Signed Data contains encapsulatedcontentinfo, which contains an optional content, whereas RFC 2315 contains content data.

> **Note:**
>
> You must keep these differences in mind if you require interoperability with PKCS#7 implementations.

4.2 Setting Up Your Oracle CMS Environment

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`. In order to use Oracle CMS, your system must have the Java Development Kit (JDK) version 1.6 or higher. Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files.

Make sure the following items are included in your `CLASSPATH`:

- the `osdt_core.jar` file
- the `osdt_cert.jar` file
- the `osdt_cms.jar` file

For example, your `CLASSPATH` might look like this:

```
%CLASSPATH%;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cms.jar;
```

4.3 Understanding and Developing Applications with Oracle CMS

The Oracle CMS API enables you to build nested (wrapped) CMS objects with no limit on the number of wrappings. Through different approaches, you can use Oracle CMS classes to develop CMS objects.

In this section we introduce Oracle CMS classes, and explain how they enable you to take different approaches to developing CMS objects, and describe how to work with the objects.

This section contains these topics:

- About Oracle CMS Classes
- About CMS Object Types
- Constructing CMS Objects using the CMS***ContentInfo Classes
- CMS Objects using the CMS***Stream and CMS***Connector Classes

4.3.1 About Oracle CMS Classes

The Oracle CMS classes provides the ability to read and write CMS objects.

There are two approaches to reading and writing CMS objects with the `oracle.security.crypto.cms` package:

- Using the `CMSContentInfo` classes, which are relatively easy to utilize
- Using one of the following classes:
  - `CMSInputStream`
  - `CMSOutputStream`
  - `CMSInputConnector`
  - `CMSOutputConnector`

  These classes provide the ability to read and write CMS objects in a single pass, eliminating the need to accumulate the input data before writing any output.

4.3.2 About CMS Object Types

Detached object and **Degenerate object** are some CMS object types. A **detached object** applies to data and receipt content types. A **degenerate object** is a certificate-only signed-data object and is defined only for the signed-data content type.

A detached object applies to data and receipt content types. For these types, a detached object is one where the protected content is absent.

Degenerate object refers to the case where the signed-data object has no signers. It is normally used to store certificates and is associated with file extensions `p7b` and `p7c`.

An external signature is defined only for the signed-data content type. It is essentially a detached signed-data object; that is, the signed-data object has one or more signers but the content that was signed is not present in the signed-data object.

4.3.3 Constructing CMS Objects using the CMS***ContentInfo Classes

You can use the CMS***ContentInfo classes to read and write objects of the appropriate content type, construct and process detached objects, and create nested objects.

Table 4-2 lists the classes which make up the CMS***ContentInfo classes.

Table 4-2   CMS*ContentInfo Classes**

Class	Content Type
CMSDataContentInfo	CMS.id_data
ESSReceipt	CMS.id_ct_receipt (RFC-2634 receipt)
CMSDigestedDataContentInfo	CMS.id_digestedData
CMSSignedDataContentInfo	CMS.id_signedData
CMSEncryptedDataContentInfo	CMS.id_encryptedData
CMSEnvelopedDataContentInfo	CMS.id_envelopedData
CMSAuthenticateDataContentInfo	CMS.id_ct_authData

A detailed discussion of `CMS***ContentInfo` classes follows in these sections:

- Using the Abstract Base Class CMSContentInfo
- Using the CMSDataContentInfo Class
- Using the ESSReceipt Class

- The CMSDigestedDataContentInfo Class
- The CMSSignedDataContentInfo Class
- Using the CMSEncryptedDataContentInfo Class
- Understanding and Using the CMSEnvelopedDataContentInfo Class

4.3.3.1 Using the Abstract Base Class CMSContentInfo

`CMSContentInfo` is an abstract class representing a fundamental CMS object. Table 4-2 lists the subclasses of `CMSContentInfo`.

Some of the useful methods of this abstract class are described in Table 4-3.

Table 4-3   Useful Methods of CMSContentInfo

Method	Description
contentTypeName (oracle.security.crypto.asn1.ASN1ObjectID contentType)	Returns the content type of the object as a string.
getContentType()	Returns the content type of the object as an object identifier (OID).
input(java.io.InputStream is)	Initializes this object by reading a BER encoding from the specified input stream.
inputInstance(java.io.InputStream is)	Creates a new CMSContentInfo object by reading a BER encoding from the specified input stream.
isDegenerate()	Indicates if the object is degenerate.
isDetached()	Indicates if the object is detached.
output(java.io.OutputStream os)	Writes the encoding of the object to the given output stream.

4.3.3.1.1 Constructing a CMS Object

You can create a CMSContentInfo object by specifying the content type.

Perform the following steps to construct a CMS object:

1. Create the object of the specified content type.
2. Initialize the object.
3. Call the `output(..)` method to write the object encoding.

To create a new object, use one of the constructors of the concrete subclass with which you are working.

4.3.3.1.2 Reading a CMS Object

If you are reading in an existing `CMSContentInfo`, but you do not know the concrete type in advance, use `inputInstance()`. To read in one of a known concrete type, use the `no-args` constructor and then invoke the `input()` method.

Perform the following steps to read an object:

1. Call `CMSContentInfo.inputInstance(..)` to read in the object.

2. Call `getContentType()` to determine its content type.

3. You can now invoke the content type-specific operations.

4.3.3.2 Using the CMSDataContentInfo Class

The class `CMSDataContentInfo` represents an object of type id-data as defined by the constant CMS.id_data, and is intended to refer to arbitrary octet strings whose interpretation is left up to the application.

A useful method of this class is:

```
byte[] getData()
```

which returns the data stored in the data object.

To create a CMS data object:

1. Create an instance of `CMSDataContentInfo` using the constructor that takes a byte array, documentBytes, that contains the information:

   ```
   CMSDataContentInfo exdata =
       new CMSDataContentInfo(byte[] documentBytes)
   ```

2. Write the data object to a file, for example `data.p7m`:

   ```
   exdata.output(new FileOutputStream("data.p7m"));
   ```

The steps you use when reading a CMS data object depend on whether you know the object's content type.

1. Open a connection to the file using `FileInputStream`.

   If you know that the object stored in the file `data.p7m` is of content type id-data:

   ```
   CMSDataContentInfo exdata =
       new CMSDataContentInfo(new FileInputStream("data.p7m"));
   ```

   However, if you do not know the content type in advance, check the type prior to reading:

   ```
   CMSContentInfo cmsdata =
       CMSContentInfo.inputInstance(new FileInputStream("data.p7m"));
   if (cmsdata instanceof CMSDataContentInfo)
   {
       CMSDataContentInfo exdata =  (CMSDataContentInfo) cmsdata;
       // .....
   }
   ```

2. To access the information stored in the CMS data object:

   ```
   byte[] docBytes = exdata.getData();
   ```

4.3.3.3 Using the ESSReceipt Class

Class `ESSReceipt` represents an object of type id-ct-receipt as defined by the constant CMS.id_ct_receipt, and refers to an RFC-2634 receipt.

Table 4-4 lists some useful methods of this class.

Table 4-4   Useful Methods of ESSReceipt

Method	Description
byte[] getOriginatorSignatureValue()	Returns the signature value of the message that triggered the generation of this receipt.
ASN1ObjectID getReceiptContentType()	Returns the content type of the message that triggered the generation of this receipt.
byte[] getReceiptData()	Returns the encoded receipt.
byte[] getSignedContentIdentifier()	Returns the signed content identifier of the message that triggered the generation of this receipt.
void inputContent(InputStream is)	Initialize this object by reading the BER encoding from the specified input stream.

4.3.3.3.1 Creating an ESSReceipt Object

Take the following steps to create a CMS receipt object.

1. Create an instance of `ESSReceipt` using the constructor that takes a content type identifier, a byte array containing the signed content identifier and a byte array containing the originator signature value:

    ```
    ESSReceipt rcpt =
        new ESSReceipt(contentType, signedContentIdentifier,
        originatorSignatureValue);
    ```

2. Write the receipt object to a file, for example `data.p7m`:

    ```
    rcpt.output(new FileOutputStream("data.p7m"));
    ```

    > **Note:**
    >
    > When you create an ESSReceipt object, do not leave any input parameters set to `null`.

4.3.3.3.2 Reading an ESSReceipt Object

To read a receipt object:

1. Open a connection to the file using `FileInputStream`.

    If you know that the object stored in the file `data.p7m` is of content type id-ct-receipt:

    ```
    ESSReceipt rcptdata = new ESSReceipt(new FileInputStream("data.p7m"));
    ```

    Otherwise, if the content type is unknown:

    ```
    CMSContentInfo cmsdata =
        CMSContentInfo.inputInstance(new FileInputStream("data.p7m"));
    if (cmsdata instanceof ESSReceipt)
    {
    ```

```
        ESSReceipt rcptdata =  (ESSReceipt) cmsdata;
        // .....
}
```

2. Access the information stored in the receipt object:

```
ASASN1ObjectID contentType = rcptdata.getReceiptContentType();
byte[] sciBytes = rcptdata.getSignedContentIdentifier()
byte[] osvBytes = rcptdata.getOriginatorSignatureValue();
```

4.3.3.4 The CMSDigestedDataContentInfo Class

The class `CMSDigestedDataContentInfo` represents an object of type id-digestedData as defined by the constant CMS.id_digestedData.

Table 4-5 lists some of the useful methods of this class.

Table 4-5   Useful Methods of `CMSDigestedDataContentInfo`

Method	Description
`byte[] getDigest()`	Returns the message digest value.
`AlgorithmIdentifier getDigestAlgID()`	Returns the message digest algorithm ID.
`CMSContentInfo getEnclosed()`	Returns the digested content.
`ASN1ObjectID getEnclosedContentType()`	Returns the content type of the digested content.
`ASN1Integer getVersion()`	Returns the version number of this object.
`boolean isDetached()`	Indicates if this object is detached.
`void setEnclosed(CMSContentInfo content)`	Sets the encapsulated content, that is, the object that was originally digested.
`void writeDetached(boolean writeDetached)`	Indicates if the object that is being digested should be omitted when creating the CMSDigestedDataContentInfo object.

4.3.3.4.1 Constructing a CMS Digested-data Object

Take the following steps to create a CMS digested-data object.

1. Create an instance of `CMSDigestedDataContentInfo` using the constructor that takes the object to be digested and the digest algorithm identifier. For example, if contentInfo is a `CMSDataContentInfo` object and MD5 is the digest algorithm:

```
CMSDigestedDataContentInfo dig =
    new CMSDigestedDataContentInfo(contentInfo, CMS.md5);
```

2. Write the CMS digested-data object to a file named `data.p7m`.

```
dig.output(new FileOutputStream("data.p7m"));
```

4.3.3.4.2 Reading a CMS Digested-data Object

The steps you need to read a CMS `digested-data` object depend on whether you know the object's content type.

1. Open a connection to the `data.p7m` file using `FileInputStream`.

   If you know that the object stored in the file is of content type id-digestedData, open the connection as follows:

   ```
   CMSDigestedDataContentInfo digdata =
       new CMSDigestedDataContentInfo(new FileInputStream("data.p7m"));
   ```

   However, if you do not know the content type in advance, open it as follows:

   ```
   CMSContentInfo cmsdata =
       CMSContentInfo.inputInstance(new FileInputStream("data.p7m"));
   if (cmsdata instanceof CMSDigestedDataContentInfo)
   {
       CMSDigestedDataContentInfo digdata =
           (CMSDigestedDataContentInfo) cmsdata;
       // .....
   }
   ```

2. To access the information stored in the CMS digested-data object:

   ```
   int version = digdata.getVersionNumber().intValue();
   AlgorithmIdentifier digestAlgID = digdata.getDigestAlgID();
   byte[] digestValue = digdata.getDigest();
   CMSContentInfo digContentInfo = digData.getEnclosed()
   if (digData.getEnclosedContentType().equals(CMS.id_data))
       CMSDataContentInfo contentInfo = (CMSDataContentInfo)digContentInfo;
   ```

3. To verify the integrity of the protected data, verify the digest:

   ```
   digData.verify();
   ```

4.3.3.4.3 Working with Detached digested-data Objects

When working with a detached object, the object that is digested is not a part of the resulting CMS digested-data structure. To generate a detached object, call the `writeDetached (true | false)` method. For example:

```
dig.writeDetached(true);
```

While you can read in a detached CMS digested-data object as shown earlier, the digest verification will fail because the original object that was digested is not present. To resolve this, call the `setEnclosed (CMScontentInfo)` method to set the `digestedContent`:

```
digdata.setEnclosed(CMScontentInfo object);
```

followed by digest verification:

```
digdata.verify();
```

4.3.3.5 The CMSSignedDataContentInfo Class

The class `CMSSignedDataContentInfo` represents an object of type `id-signedData` as defined by the constant `CMS.id_signedData`.

Oracle CMS supports a *choice* of `IssuerAndSerialNo` or `SubjectKeyIdentifier` for use as the `SignerIdentifier`. For interoperability with PKCS #7 and S/MIME, however, the `IssuerAndSerialNo` must be used as the `SignerIdentifier`.

Table 4-6 lists some useful methods of this class:

Table 4-6   Useful Methods of CMSSignedDataContentInfo

Method	Description
void addCertificate(X509Certificate cert)	Appends the given certificate to the list of certificates which will be included with this signed data object.
void addCRL(CRL crl)	Appends the given CRL to the list of CRLs which will be included with this signed data object.
void addSignature(AttributeSet authenticatedAttributes, PrivateKey signerKey, X509Certificate signerCert, AlgorithmIdentifier digestAlgID, AlgorithmIdentifier digestEncryptionAlgID, AttributeSet unauthenticatedAttributes)	Adds a signature using the IssuerAndSerialNumber as the SignerIdentifier, that is, a Version1 CMSSignerInfo.
void addSignature(AttributeSet authenticatedAttributes, PrivateKey signerKey, X509Certificate signerCert, AlgorithmIdentifier digestAlgID, AlgorithmIdentifier digestEncryptionAlgID, AttributeSet unauthenticatedAttributes, boolean useSPKI64)	Adds a signature using the SubjectKeyIdentifier as the SignerIdentifier; that is, a Version3 CMSSignerInfo.
void addSignerInfo(X509Certificate signerCert, CMSSignerInfo signerInfo)	Adds a CMSSignerInfo to the list of signers.
Vector getCertificates()	Returns the list of certificates included with this signed data object.
Vector getCRLs()	Returns the list of CRLs included with this signed data object.
CMSContentInfo getEnclosed()	Returns the signed document.
ASN1ObjectID getEnclosedContentType()	Returns the content type of the document which was signed.
CMSSignerInfo getSignerInfo(signerCert)	Returns the CMSSignerInfo corresponding to the certificate.
ASN1Integer getVersion()	Returns the version number of this object.
boolean isDegenerate()	IIndicates if this is a degenerate CMSSignedDataContentInfo object (that is, has no SignerInfo structures)
boolean isDetached()	Indicates if this is a detached object.
boolean isExternalSignature()	Checks for the presence of external signatures.
void setEnclosed(CMSContentInfo content)	Sets the content which was signed.

Table 4-6   (Cont.) Useful Methods of CMSSignedDataContentInfo

Method	Description
Enumeration signers()	Returns the signatures on this signed data object in the form of an enumeration, each element of which is an instance of `CMSSignerInfo`.
void verify(CertificateTrustPolicy trustPolicy)	Returns normally if this CMS signed data object contains at least one valid signature, according to the given trust policy.
void verify(CertificateTrustPolicy trustPolicy,CMSContentInfo contentInfo)	Returns normally if this signed data object contains at least one valid signature, according to the given trust policy.
void verifySignature(X509Certificate signerCert)	Returns successfully if this signed data object contains a signature which is validated by the given certificate.
void verifySignature(X509Certificate signerCert, CMSContentInfo contentInfo)	Returns successfully if this signed data object contains a signature which is validated by the given certificate and data.
void writeExternalSignature(boolean createExternalSignature)	Indicates if an external signature must be created.

Users of RSA and DSA signature algorithms should note that the providers are pluggable in the Oracle CMS implementation.

4.3.3.5.1 Constructing a CMS Signed-data Object

Follow these steps to create a CMS signed-data object:

1. Create an instance of `CMSSignedDataContentInfo`. For example, to create the `CMSSignedDataContentInfo` object, pass the contentInfo object (the data that is to be signed):

    ```
    CMSSignedDataContentInfo sig =
        new CMSSignedDataContentInfo(contentInfo);
    ```

2. Add signatures:

    ```
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
    X509Certificate envCert =  (X509Certificate)cf.generateCertificate(new
    FileInputStream("name1"));
    PrivateKey signerKey =
        ...;
    ```

    a. To add a signature using the IssuerAndSerialNo as the SignerIdentifier, MD5 digests and RSA Signature Algorithm:

```
sig.addSignature(null, signerKey, signerCert, CMS.md5,
    CMS.rsaEncryption, null);
```

b. To add a signature using the 64 bit SubjectKeyIdentifier as the SignerIdentifier, SHA-1 digests and DSS Signature Algorithm:

```
sig.addSignature(null, signerKey, signerCert, CMS.sha_1,
    CMS.dsaWithSHA, null, true);
```

c. To add a signature using the 160 bit SubjectKeyIdentifier as the SignerIdentifier, SHA-1 digests and RSA Signature Algorithm:

```
sig.addSignature(null, signerKey, signerCert, CMS.sha_1,
    CMS.rsaEncryption, null, false);
```

3. Add any Certificates and CRLs:

```
sig.addCertificate (....)
sig.addCRL (...)
```

4. Write the CMS signed-data object to a file, for example `data.p7m`:

```
sig.output(new FileOutputStream("data.p7m"));
```

4.3.3.5.2 Reading a CMS Signed-data Object

The steps you need to read a CMS signed-data object depend on whether you know the object's content type.

1. Open a connection to the `data.p7m` file using `FileInputStream`.

   If you know that the object stored in the file is of content type `id-signedData`:

```
CMSSignedDataContentInfo sigdata =
    new CMSSignedDataContentInfo(new FileInputStream("data.p7m"));
```

   However, if you do not know the content type in advance:

```
CMSContentInfo cmsdata =
    CMSContentInfo.inputInstance(new FileInputStream("data.p7m"));
if (cmsdata instanceof CMSSignedDataContentInfo)
{
    CMSSignedDataContentInfo sigdata =
        (CMSSignedDataContentInfo) cmsdata;
    // .....
}
```

2. Access the information stored in the CMS `signed-data` object:

```
int version = sigdata.getVersion().intValue();
CMSContentInfo sigContentInfo = sigData.getEnclosed()
Vector certs = sigdata.getCertificates();
Vector crls = sigData.getCRLs();
Enumeration e = sigData.signers();
CMSContentInfo sigContentInfo = sigData.getEnclosed();
if (sigData.getEnclosedContentType().equals(CMS.id_data))
    CMSDataContentInfo contentInfo = (CMSDataContentInfo)sigContentInfo;
```

3. Verify the signature using the signer's public key certificate:

```
sigData.verifySignature(signerCert);
```

4. To get more information about the signer:

```
CMSSignerInfo sigInfo = sigdata.getSignerInfo(signerCert);
byte[] signatureValue = sigInfo.getEncryptedDigest();
AlgorithmIdentifier digest = sigInfo.getDigestAlgID();
AlgorithmIdentifier signature = sigInfo.getDigestEncryptionAlgID();
AttributeSet signedAttributes = sigInfo.getAuthenticatedAttributes();
AttributeSet unsignedAttributes = sigInfo.getUnauthenticatedAttributes();
```

4.3.3.5.3 Working with External Signatures (Detached Objects)

For a detached object, the signed object is not part of the resulting CMS signed-data structure. To generate a detached object, call the `writeExternalSignature()` method:

```
sig.writeExternalSignature(true);
```

While you can read in a detached CMS signed-data object as shown in "Reading a CMS Signed-data Object", the signature verification will fail because the original object that was signed is not present. To address this, first call the `setEnclosed (..)` method to set the signed content:

```
sigdata.setEnclosed(contentInfo);
```

followed by signature verification:

```
sigdata.verifySignature(signerCert);
```

4.3.3.5.4 Working with Certificates/CRL-Only Objects

These are essentially `CMSSignedDataContentInfo` objects with attached certificates, or CRLs, or both, but without any signatures. To generate a Certificate/CRL-only object:

```
CMSSignedDataContentInfo sigdata =
    new CMSSignedDataContentInfo(new CMSDataContentInfo(new byte[0]));
sigData.addCertificate (...);
sigData.addCRL( ...);
sigData.output(..);
```

You can read in a Certificate/CRL-only signed-data object as shown in "Reading a CMS Signed-data Object".

4.3.3.6 Using the CMSEncryptedDataContentInfo Class

The class `CMSEncryptedDataContentInfo` represents an object of type id-encryptedData as defined by the constant CMS.id_encryptedData.

Table 4-7 lists some useful methods of this class.

Table 4-7   Useful Methods of CMSEncryptedDataContentInfo

Method	Description
AlgorithmIdentifier getContentEncryptionAlgID()	Returns the content encryption algorithm
CMSContentInfo getEnclosed(SecreKey decryptionKey)	Returns the decrypted content
ASN1ObjectID getEnclosedContentType()	Returns the content type of the encrypted content
byte[] getEncryptedContent()	Returns the encrypted content

Table 4-7   (Cont.) Useful Methods of CMSEncryptedDataContentInfo

Method	Description
AttributeSet getUnprotectedAttributes()	Returns the set of unprotected attributes
ASN1Integer getVersion()	Returns the version number
boolean isDetached()	Indicates if this is a detached CMS object
void setUnprotectedAttributes (oracle.security.crypto.cert.AttributeSet unprotectedAttributes)	Sets the unprotected attributes
void writeDetached (boolean writeDetachedObject)	Indicates if the encryptedContent will be a part of the EncryptedContentInfo structure in this object's output encoding

Users of encryption operations, including RC2, DES, Triple-DES, AES, and so on, should note that the cipher providers are pluggable in the Oracle Security Engine implementation.

4.3.3.6.1 Constructing a CMS Encrypted-data Object

To create an `encrypted-data` object:

1. Create an instance of `CMSEncryptedDataContentInfo`. For example, if `contentInfo` is a `CMSDataContentInfo` object and the cipher is Triple-DES in CBC mode:

   ```
   SecretKey contentEncryptionKey =
      KeyGenerator.getInstance("DESede").generateKey();

   CMSEncryptedDataContentInfo enc =
      new CMSEncryptedDataContentInfo(contentInfo, contentEncryptionKey,
         CMS.des_ede3_cbc);
   ```

2. Write the `encrypted-data` object to a file, say `data.p7m`:

   ```
   enc.output(new FileOutputStream("data.p7m"));
   ```

4.3.3.6.2 Reading a CMS Encrypted-data Object

The steps you need to read an `encrypted-data` object depend on whether you know the object's content type.

1. Open a connection to the `data.p7m` file using `FileInputStream`.

   If you know that the object stored in the file `data.p7m` is of content type id-encryptedData:

   ```
   CMSEncryptedDataContentInfo encdata =
      new CMSEncryptedDataContentInfo(new FileInputStream("data.p7m"));
   ```

   However, if you do not know the content type in advance:

```
CMSContentInfo cmsdata =
    CMSContentInfo.inputInstance(new FileInputStream("data.p7m"));
if (cmsdata instanceof CMSEncryptedDataContentInfo)
{
    CMSEncryptedDataContentInfo encdata =
        (CMSEncryptedDataContentInfo) cmsdata;
    // .....
}
```

2. To access the information stored in the CMS `encrypted-data` object:

```
int version = encdata.getVersion().intValue();
AlgorithmIdentifier encAlgID = encdata.getContentEncryptionAlgID();
byte[] encValue = encdata.getEncryptedContent();
CMSContentInfo encContentInfo =
    encdata.getEnclosed(ContentEncryptionKey);  //Decrypt the Content
if (encData.getEnclosedContentType().equals(CMS.id_data))
    CMSDataContentInfo contentInfo = (CMSDataContentInfo)encContentInfo;
```

4.3.3.6.3 Generating a Detached encrypted-data CMS Object

If it is a detached object, the encrypted object is not a part of the resulting CMS `encrypted-data` structure. To generate a detached object, call the `writeDetached (..)` method:

```
encData.writeDetached(true);
```

While you can read in a detached CMS encrypted-data object as shown in "Reading a CMS Encrypted-data Object", the content decryption will fail because the original object that was encrypted is not present. Call the `setEnclosed (..)` method to set the encryptedContent:

```
encData.setEnclosed(encryptedcontent());
```

followed by content decryption:

```
encdata.getEnclosed(ContentEncryptionKey);
```

4.3.3.7 Understanding and Using the CMSEnvelopedDataContentInfo Class

The class `CMSEnvelopedDataContentInfo` represents an object of type id-envelopedData as defined by the constant CMS.id_envelopedData.

Table 4-8 lists some useful methods of this class:

Table 4-8   Useful Methods of CMSEnvelopedDataContentInfo

Method	Description
void addRecipient(AlgorithmIdentifier keyEncryptionAlgID, SecretKey keyEncryptionKey, byte[] keyIdentifier, Date keyDate, ASN1Sequence otherKeyAttribute)	Adds a recipient using the key encryption (wrap) key exchange mechanism.
void addRecipient(CMSRecipientInfoSpec ris)	Adds a recipient using the key exchange mechanism specification
void addRecipient(X509Certificate recipientCert, AlgorithmIdentifier keyEncryptionAlgID)	Adds a recipient using the key transport (IssuerAndSerialNo) key exchange mechanism

Table 4-8   (Cont.) Useful Methods of CMSEnvelopedDataContentInfo

Method	Description
void addRecipient(X509Certificate recipientCert, AlgorithmIdentifier keyEncryptionAlgID, boolean useSPKI64)	Adds a recipient the key transport (SubjectKeyIdentifier) key exchange mechanism
AlgorithmIdentifier getContentEncryptionAlgID()	Returns the content encryption algorithm
CMSContentInfo getEnclosed(PrivateKey privateKey, X509Certificate recipientCert)	Returns the enclosed content after decryption using Key Transport RecipientInfo
CMSContentInfo getEnclosed(SecretKey symmetricKey, byte[] keyIdentifier)	Returns the enclosed content after decryption using Key Encryption RecipientInfo
CMSContentInfo getEnclosed(SecretKey symmetricKey, byte[] keyIdentifier,Date keyDate)	Returns the enclosed content after decryption
ASN1ObjectID getEnclosedContentType()	Returns the content type of the encrypted content
byte[] getEncryptedContent()	Returns the enclosed content which is encrypted
OriginatorInfo getOriginatorInfo()	Returns the OriginatorInfo
AttributeSet getUnprotectedAttribs()	Returns the unprotected attributes
ASN1Integer getVersion()	Returns the version number
boolean isDetached()	Indicates if the encrypted content is not present
Enumeration recipients()	Returns the list of message recipients
void setEnclosed(byte[] encryptedContent)	Sets the Encrypted Content
void setOriginatorInfo(OriginatorInfo origInfo)	Sets the OriginatorInfo
void setUnprotectedAttribs (oracle.security.crypto.cert.AttributeSet unprotectedAttributes)	Sets the unprotected attributes
void writeDetached(boolean writeDetached)	Indicates if the encrypted content must be omitted from this object's output encoding

4.3.3.7.1 Constructing a CMS Enveloped-data Object

Take these steps to create an `enveloped-data` object:

1. Create an instance of `CMSEnvelopedDataContentInfo`. For example, if contentInfo is a `CMSDataContentInfo` object and the cipher is Triple-DES in CBC mode:

```
CMSEnvelopedDataContentInfo env =
    new CMSEnvelopedDataContentInfo(contentInfo, CMS.des_ede3_cbc);
```

2. Add recipients, keeping in mind the recipient's key management technique.

   • If the recipient uses the key encryption (wrap) key management mechanism:

```
env.addRecipient(keyEncryptionAlgID, keyEncryptionKey,
    keyIdentifier, keyDate, otherKeyAttribute);
```

- If the recipient key exchange mechanism was specified using a `CMSRecipientInfoSpec` object:

```
env.addRecipient(ris)
```

- If the recipient uses the key transport (`IssuerAndSerialNo` recipient identifier) key management mechanism:

```
env.addRecipient(recipientCert, CMS.rsaEncryption);
```

- If the recipient uses the key transport (64-bit `SubjectKeyIdentifier` recipient identifier) key management mechanism:

```
env.addRecipient(recipientCert, CMS.rsaEncryption, true)
```

- If the recipient uses the key transport (160-bit `SubjectKeyIdentifier` recipient identifier) key management mechanism:

```
env.addRecipient(recipientCert, CMS.rsaEncryption, false)
```

3. Set any optional arguments:

```
env.setOriginatorInfo(originatorInfo);
env.setUnprotectedAttribs(unprotectedAttributes);
```

4. Write the CMS `enveloped-data` object to a file, say `data.p7m`:

```
enc.output(new FileOutputStream("data.p7m"));
```

4.3.3.7.2 Reading a CMS Enveloped-data Object

The steps you need to read the object depend on whether you know the object's content type.

1. Open a connection to the `data.p7m` file using `FileInputStream`. If you know that the object stored in the file is of content type id-envelopedData, open the connection as follows:

```
CMSEnvelopedDataContentInfo envdata =
    new CMSEnvelopedDataContentInfo(new FileInputStream("data.p7m"));
```

However, if you do not know the content type in advance, open it as follows:

```
CMSContentInfo cmsdata =
    CMSContentInfo.inputInstance(new FileInputStream("data.p7m"));
if (cmsdata instanceof CMSEnvelopedDataContentInfo)
{
    CMSEnvelopedDataContentInfo envdata =
        (CMSEnvelopedDataContentInfo) cmsdata;
    //
    .....
}
```

2. To access the information stored in the `enveloped-data` object:

```
int version = envdata.getVersion().intValue();
AlgorithmIdentifier encAlgID = envdata.getContentEncryptionAlgID();
ASN1ObjectID contentType = envdata.getEnclosedContentType();
byte[] encryptedContent = envdata.getEncryptedContent();
OriginatorInfo origInfo = envdata.getOriginatorInfo();
AttributeSet unprotected = envdata.getUnprotectedAttribs();
```

3. Decrypt the content depending on the recipient information:

```
CMSContentInfo envContentInfo =
    env.getEnclosed(privateKey, recipientCert);
```

or

```
CMSContentInfo envContentInfo =
    env.getEnclosed(symmetricKey, keyIdentifier);
```

or

```
CMSContentInfo envContentInfo =
    env.getEnclosed(symmetricKey, keyIdentifier, keyDate)
if (envContentInfo instanceof CMSDataContentInfo)
{
    CMSDataContentInfo contentInfo = (CMSDataContentInfo) envContentInfo;
    // ...
}
```

4.3.3.7.3 About the Key Transport Key Exchange Mechanism

This mechanism supports the use of either `IssuerAndSerialNo` or `SubjectKeyIdentifier` as the recipient identifier.

4.3.3.7.4 About the Key Agreement Key Exchange Mechanism

This mechanism is not currently supported.

4.3.3.7.5 About the Key Encryption (Wrap) Key Exchange Mechanism

Oracle CMS supports CMS3DESWrap and CMSRC2Wrap algorithms. Mixed mode wrapping is not supported; for example, 3DES keys cannot be RC2-wrapped.

> **Note:**
>
> Using the `OtherKeyAttribute` could cause interoperability problems.

4.3.3.7.6 Using the Detached Enveloped-data CMS Object

If working with a detached object, note that the enveloped object is not part of the resulting CMS enveloped-data structure. Call the `writeDetached (..)` method to generate a detached object:

```
envdata.writeDetached(true);
```

While you can read in a detached `enveloped-data` object as shown in "Reading a CMS Enveloped-data Object", the content decryption will fail because the original, enveloped object is not present. Call the `setEnclosed (..)` method to set the enveloped content:

```
envdata.setEnclosed(env.getEncryptedContent());
```

followed by content decryption:

```
envdata.getEnclosed(............);
```

4.3.3.8 Using the CMSAuthenticatedDataContentInfo Class

The class `CMSAuthenticatedDataContentInfo` represents an object of type id-ct-authData as defined by the constant `CMS.id_ct_authData`.

> **Note:**
>
> Oracle CMS supports HMAC with SHA-1 Message Authentication Code (MAC) Algorithm.

Table 4-9 lists some useful methods of this class.

Table 4-9   Useful Methods of CMSAuthenticatedDataContentInfo

Method	Description
void addRecipient(AlgorithmIdentifier keyEncryptionAlgID, SecretKey keyEncryptionKey, byte[] keyIdentifier, java.util.Date keyDate, ASN1Sequence otherKeyAttribute)	Adds a recipient using the key wrap key exchange mechanism
void addRecipient(CMSRecipientInfoSpec ris)	Adds a recipient using the specified key exchange mechanism
void addRecipient(X509Certificate recipientCert, AlgorithmIdentifier keyEncryptionAlgID)	Adds a recipient using the key transport key exchange mechanism using the IssuerAndSerialNo as the recipient identifier
void addRecipient(X509Certificate recipientCert, AlgorithmIdentifier keyEncryptionAlgID, boolean useSPKI64)	Adds a recipient using the key transport key exchange mechanism using the SubjectKeyIdentifier as the recipient identifier
AttributeSet getAuthenticatedAttributes()	Returns the Authenticated Attributes
AlgorithmIdentifier getDigestAlgID()	Returns the digest algorithm
CMSContentInfo getEnclosed()	Returns the authenticated content
ASN1ObjectID getEnclosedContentType()	Returns the content type of the enclosed content
byte[] getMAC()	Returns the message authentication code
AlgorithmIdentifier getMACAlgID()	Returns the MAC algorithm used for authentication

Table 4-9   (Cont.) Useful Methods of CMSAuthenticatedDataContentInfo

Method	Description
OriginatorInfo getOriginatorInfo()	Returns the Originator information
AttributeSet getUnauthenticatedAttributes()	Returns the Unauthenticated Attributes
ASN1Integer getVersion()	Returns the version number
boolean isDetached()	Indicates if this object is detached
java.util.Enumeration recipients()	Returns the list of message recipients
void setAuthenticatedAttributes(AttributeSet authenticatedAttributes, AlgorithmIdentifier digestAlgorithm)	Sets the Authenticated attributes
void setEnclosed(CMSContentInfo content)	Sets the authenticated content
void setOriginatorInfo(OriginatorInfo originatorInfo)	Sets the OriginatorInfo
void setUnauthenticatedAttributes(AttributeSet unauthenticatedAttributes)	Sets the unauthenticated attributes
void verifyMAC(PrivateKey privateKey, X509Certificate recipientCert)	Returns the enclosed content after decryption
void verifyMAC(SecretKey symmetricKey, byte[] keyIdentifier)	Returns the enclosed content after decryption
void verifyMAC(SecretKey symmetricKey, byte[] keyIdentifier, Date keyDate)	Returns the enclosed content after decryption
void verifyMAC(SecretKey symmetricKey, byte[] keyIdentifier, Date keyDate, ASN1Sequence otherKeyAttribute)	Returns the enclosed content after decryption
void writeDetached(boolean writeDetachedObject)	Indicates if the authenticated content must be omitted from this object's output encoding

4.3.3.8.1 Constructing a CMS Authenticated-data Object

The starting point for working with authenticated-data objects is the
`CMSAuthenticatedDataContentInfo` class.

Take the following steps to create an authenticated-data object:

1. Create an instance of `CMSAuthenticatedDataContentInfo`. In the following example, contentInfo is a `CMSDataContentInfo` object, Triple-DES HMAC key and HMAC with SHA-1 MAC algorithm:

```
SecretKey contentEncryptionKey =
    KeyGenerator.getInstance("DESede").generateKey();
CMSAuthenticatedDataContentInfo auth =
    new CMSAuthenticatedDataContentInfo(contentInfo,
contentEncryptionKey, CMS.hmac_SHA_1);
```

2. Add recipients, keeping in mind the recipient's key management technique.

- If the recipient uses the key encryption (wrap) key management mechanism:

```
auth.addRecipient(keyEncryptionAlgID, keyEncryptionKey, keyIdentifier,
    keyDate, otherKeyAttribute);
```

- If the recipient key exchange mechanism was specified using a `CMSRecipientInfoSpec` object:

```
auth.addRecipient(ris)
```

- If the recipient uses the key transport (`IssuerAndSerialNo` recipient identifier) key management mechanism:

```
auth.addRecipient(recipientCert, CMS.rsaEncryption);
```

- If the recipient uses the key transport (64-bit `SubjectKeyIdentifier` recipient identifier) key management mechanism:

```
auth.addRecipient(recipientCert, CMS.rsaEncryption, true)
```

- If the recipient uses the key transport (160-bit `SubjectKeyIdentifier` recipient identifier) key management mechanism:

```
auth.addRecipient(recipientCert, CMS.rsaEncryption, false)
```

3. Set any optional arguments:

```
auth.setAuthenticatedAttributes(authenticatedAttributes, CMS.md5);
auth.setOriginatorInfo(originatorInfo);
auth.setUnauthenticatedAttributes(unauthenticatedAttributes);
```

4. Write the CMS authenticated-data object to a file, say `data.p7m`:

```
auth.output(new FileOutputStream("data.p7m"));
```

4.3.3.8.2 Reading a CMS Authenticated-data Object

The steps you need to read the object depend on whether you know the object's content type.

The steps to read an object are as follows:

1. Open a connection to the `data.p7m` file using `FileInputStream`. If you know that the object stored in the file is of content type `id-ct-authData`:

```
CMSAuthenticatedDataContentInfo authdata =
    new CMSAuthenticatedDataContentInfo(new FileInputStream("data.p7m"));
```

However, if you do not know the content type in advance:

```
CMSContentInfo cmsdata =
    CMSContentInfo.inputInstance(new FileInputStream("data.p7m"));
if (cmsdata instanceof  CMSAuthenticatedDataContentInfo)
{
CMSAuthenticatedDataContentInfo authdata =
                          (CMSAuthenticatedDataContentInfo) cmsdata;
    // .....
}
```

2. To access the information stored in the CMS `authenticated-data` object:

```
int version = authdata.getVersion().intValue();
AlgorithmIdentifier macAlgID = authdata.getMACAlgID();
byte[] macValue = authdata.getMAC();
CMSContentInfo authContentInfo = authdata.getEnclosed();
```

```
if (authData.getEnclosedContentType().equals(CMS.id_data))
    CMSDataContentInfo contentInfo = (CMSDataContentInfo)authContentInfo;
```

3. Verify the MAC depending on the recipient information:

```
authdata.verifyMAC(recipientPrivateKey, recipientCert);
```

or

```
authdata.verifyMAC(symmetricKey, keyIdentifier)
```

or

```
authdata.verifyMAC(symmetricKey, keyIdentifier, keyDate)
```

or

```
authdata.verifyMAC(symmetricKey, keyIdentifier, keyDate,
    otherKeyAttribute)
```

4.3.3.8.3 Working with Detached Authenticated-data CMS Objects

While you can read in a detached authenticated-data object as shown earlier, the MAC verification will fail because the original object that was authenticated is not present. To resolve this, call the `setEnclosed (..)` method to set the authenticated content:

```
authdata.setEnclosed(contentInfo);
```

followed by MAC verification using the appropriate key exchange mechanism:

```
authdata.verifyMAC(...)
```

4.3.3.9 Working with Wrapped (Triple or more) CMSContentInfo Objects

To wrap a `CMSContentInfo` object in another `CMSContentInfo` object, you simply pass an initialized `CMSContentInfo` object to the enclosing `CMSContentInfo` object through its constructor. Call the `output (..)` method of the enclosing outermost `CMSContentInfo` object to generate the nested object.

4.3.3.9.1 Reading a Nested (Wrapped) CMS Object

The approach to reading a nested object depends on whether you know the outermost content type in advance.

If you do not know the outermost content type in advance, call the static method:

```
CMSContentInfo.inputInstance( ... )
```

If you do know the outermost content type in advance, call the appropriate constructor:

```
new CMS***DataContentInfo( .... )
```

Then, recursively call the `getEnclosed(..)` method to extract the next inner object.

4.3.4 CMS Objects using the CMS***Stream and CMS***Connector Classes

The CMS**DataContentInfo classes provide the same functionality as the CMS***Stream classes. The primary advantage of the CMS***Stream classes over the CMS**DataContentInfo classes is that CMS objects can be created or read in one pass without having to accumulate all of the input data. A `CMSInputConnector` is used in place of a `CMSInputStream` when reading nested CMS objects.

Table 4-10 lists the content types of the CMS***Stream classes:

Table 4-10   The CMS*Stream Classes**

Class	Content Type
CMSDigestedDataInputStream, CMSDigestedDataOutputStream	CMS.id_digestedData
CMSSignedDataInputStream, CMSSignedDataOutputStream	CMS.id_signedData
CMSEncryptedDataInputStream, CMSEncryptedDataOutputStream	CMS.id_encryptedData
CMSEnvelopedDataInputStream, CMSEnvelopedDataOutputStream	CMS.id_envelopedData
CMSAuthenticatedDataInputStream, CMSAuthenticatedDataOutputStream	CMS.id_ct_authData

Table 4-11 lists the content types of the CMS***Connector classes:

Table 4-11   The CMS*Connector Classes**

Class	Content Type
CMSDigestedDataInputConnector, CMSDigestedDataOutputConnector	CMS.id_digestedData
CMSSignedDataInputConnector, CMSSignedDataOutputConnector	CMS.id_signedData
CMSEncryptedDataInputConnector, CMSEncryptedDataOutputConnector	CMS.id_encryptedData
CMSEnvelopedDataInputConnector, CMSEnvelopedDataOutputConnector	CMS.id_envelopedData
CMSAuthenticatedDataInputConnector, CMSAuthenticatedDataOutputConnector	CMS.id_ct_authData

4.3.4.1 Limitations of the CMS***Stream and CMS***Connector Classes

There are some limitations to CMS***Stream and CMS***Connector classes when processing objects:

1. They cannot verify the digest of a detached CMS `id-digestedData` object.

2. They cannot verify the signature of a detached CMS `id-signedData` object.

3. They cannot verify the MAC of a detached CMS `id-ct-authData` object.

> ⚠️ **Caution:**
>
> Always use the CMS**DataContentInfo classes when processing detached objects.

4.3.4.2 Difference between CMS***Stream and CMS***Connector Classes

The CMS***OutputStream class is an output stream filter which wraps the data written to it within a CMS (RFC-2630) ContentInfo structure, whose BER encoding is then written to the underlying output stream. The CMS***OutputConnector class is an output stream filter which likewise wraps the data written to it within a CMS (RFC-2630) `ContentInfo` structure, except that only the values octets of the Content field of the `ContentInfo` structure (minus the explicit [0] tag) are written to the underlying output stream.

The CMS***InputStream class is an input stream filter which reads in a BER encoding of a CMS (RFC-2630) `ContentInfo` structure from the underlying output stream. The CMS***InputConnector class is an input stream filter that expects the underlying input stream to be positioned at the start of the value octets of the `Content` field of the `ContentInfo` structure (after the explicit [0] tag).

CMS***Connectors are useful in creating and reading nested objects.

4.3.4.3 Using the CMS***OutputStream and CMS***InputStream Classes

CMS***InputStream includes methods to read in CMS objects. CMS***OutputStream writes a CMS object to the output stream.

To construct an object:

1. Create a CMS***OutputStream class of the appropriate content type. All the relevant parameters are passed through the constructor.

2. Write the data being protected to the CMS***OutputStream created in step 1.

3. After all the data is written, close the CMS***OutputStream created in step 1 .

To read an object:

1. Create a CMS***InputStream class of the appropriate content type by passing the underlying input stream through the constructor.

2. Read the protected data from the CMS***InputStream created in step 1 using the `read()` and `read (byte[],...)` methods.

3. Invoke `terminate()` after you have finished reading data from the CMS***InputStream created in step 1. This completes the reading of the object.

4. Invoke the appropriate methods to verify that the protected content is secure.

4.3.4.3.1 Working with the CMS id-data Object

The `getData()` method returns the data which can then be written to a CMS***OutputStream or CMS***OutputConnector.

4.3.4.3.2 Working with the CMS id-ct-receipt Object

The `getReceiptData()` method returns the encoded receipt which can then be written to a CMS***OutputStream or CMS***OutputConnector.

To read `ESSReceipt` data from the input stream:

```
byte[] rcptData = in.read(...);
ESSReceipt er = new ESSReceipt();
er.inputContent(rcptData);
```

4.3.4.3.3 Working with CMS id-digestedData Objects

You will not be able to verify the digest of a detached digested-data object. Setting the boolean parameter writeEContentInfo in the `CMSDigestedDataOutputStream` constructor to false enables you to create a detached digested-data object.

4.3.4.3.4 Working with CMS id-signedData Objects

You will not be able to verify the signature of a detached `signed-data` object.

The `CMSSignerInfoSpec` class stores signer-specific information. For every signature you want to add, you will need to create a corresponding `CMSSignerInfoSpec` object which is then passed to the constructor.

Setting the boolean parameter `createExternalSignatures` in the `CMSSignedDataOutputStream` constructor to true enables you to create a detached `signed-data` object or external signatures.

To create a Certificate/CRL only object, do not pass any signer information to the `CMSDSignedDataOutputStream` constructor.

4.3.4.3.5 Working with CMS id-encryptedData Objects

Setting the boolean parameter `writeEncryptedOutput` in the `CMSEncryptedDataOutputStream` constructor to false enables you to create a detached `encrypted-data` object.

4.3.4.3.6 Working with CMS id-envelopedData Objects

The `CMSRecipientInfoSpec` class stores recipient-specific information. For every recipient you want to add, you will need to create a corresponding `CMSRecipientInfoSpec` object which is then passed to the constructor.

Setting the boolean parameter `writeContent` in the `CMSEnvelopedDataOutputStream` constructor to `false` enables you to create a detached `enveloped-data` object.

Recipients are classified according to their exchange mechanism. This table defines the different mechanisms:

Exchange Mechanism	How to Use
Key Transport Key Exchange Mechanism	Use the `CMSKeyTransRecipientInfoSpec` class to store recipient information that uses the key transport key management mechanism.
Key Agreement Key Exchange Mechanism	This mechanism is not supported at this time.
Key Encryption (wrap) Key Exchange Mechanism	Use the `CMSKEKRecipientInfoSpec` class to store recipient information that uses the key wrap key management mechanism.

4.3.4.3.7 About CMS id-ct-authData Objects

You will not be able to verify the MAC of a detached `authenticated-data` object.

Setting the boolean parameter detachEncapContent in the `CMSAuthenticatedDataOutputStream` constructor to true enables you to create a detached `authenticated-data` object.

4.3.4.4 Wrapping (Triple or more) CMS***Connector Objects

You use CMS***OutputConnectors to create nested objects.

Use the following code to create signed, enveloped, digested, and encrypted data and write it to the file `nested.p7m`:

```
// nested.p7m <--- FileOutputStream <--- CMSSignedDataOutputConnector
//     <--- CMSEnvelopedDataOutputConnector <---
//        <----  CMSDigestedDataOutputConnector <---
//            <----  CMSEncryptedDataOutputConnector <---
//                <---- write the data (byte[] data)

FileOutputStream fos = new FileOutputStream("nested.p7m");
CMSSignedDataOutputConnector conn1 =
    new CMSSignedDataOutputConnector(fos, .....);
CMSEnvelopedDataOutputConnector conn2 =
    new CMSEnvelopedDataOutputConnector(conn1, ...);
CMSDigestedDataOutputConnector conn3 =
    new CMSDigestedDataOutputConnector(conn2, ...);
CMSEncryptedDataOutputConnector conn4 =
    new CMSEncryptedDataOutputConnector(conn3, ...);
OutputStream os = conn4.getOutputStream();
os.write(data);
os.close();
```

To read signed, enveloped, digested, and encrypted data stored in file `nested.p7m`:

```
// nested.p7m ---> FileInputStream ---> CMSSignedDataInputConnector -
//     ---> CMSEnvelopedDataInputConnector ---
//        -----> CMSDigestedDataInputConnector ---
//            ----> CMSEncryptedDataInputConnector ---
//                ---> read the data (byte[] data)

FileInputStream fos = new FileInputStream("nested.p7m");
CMSSignedDataInputConnector conn1 =
    new CMSSignedDataInputConnector(fos, .....);
CMSEnvelopedDataInputConnector conn2 =
```

```
    new CMSEnvelopedDataInputConnector(conn1, ...);
CMSDigestedDataInputConnector conn3 =
    new CMSDigestedDataInputConnector(conn2, ...);
CMSEncryptedDataInputConnector conn4 =
    new CMSEncryptedDataInputConnector(conn3, ...);
InputStream is = conn4.getInputStream();
is.read(data);
```

4.4 The Oracle CMS Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer
Tools guide explains the classes and methods of Oracle CMS.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

5

Oracle S/MIME

Oracle S/MIME API is a java solution, which includes classes, interfaces, and methods to work with S/MIME objects.

We provide a survey of the classes and features of Oracle S/MIME:

- Oracle S/MIME Features and Benefits
- Setting Up Your Oracle S/MIME Environment
- Developing Applications with Oracle S/MIME
- The Oracle S/MIME Java API Reference

5.1 Oracle S/MIME Features and Benefits

Oracle S/MIME API is a java solution with support for X.509 certificate and private key encryption.

It has the following features:

- Full support for X.509 Version 3 certificates with extensions, including certificate parsing and verification
- Support for X.509 certificate chains in PKCS #7 and PKCS #12 formats
- Private key encryption using PKCS #5, PKCS #8, and PKCS #12
- An integrated ASN.1 library for input and output of data in ASN.1 DER/BER format

5.2 Setting Up Your Oracle S/MIME Environment

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`. In order to use Oracle S/MIME, your system must have the Java Development Kit (JDK) version 1.6 or higher. Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files.

Oracle S/MIME also requires:

- An implementation of the JavaBeans Activation Framework (JAF). Oracle's royalty-free implementation is available at:

  `http://www.oracle.com/technetwork/java/jaf11-139815.html`

- An implementation of the JavaMail API. Oracle's royalty-free implementation is available at:

  `http://www.oracle.com/technetwork/java/index-138643.html`

If you are using POP or IMAP, be sure to download Oracle's POP3 (or IMAP) Provider, which is also available at the JavaMail page.

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar` file

- `osdt_cert.jar` file

- `osdt_cms.jar` file

- `osdt_smime.jar` file

- Your JAF (Java Activation Framework), JavaMail, and POP3 provider installations.

> **Note:**
>
> Java Activation Framework is included in JDK 1.6.

For example:

```
setenv CLASSPATH $CLASSPATH:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_core.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cert.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cms.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_smime.jar:
/usr/lib/jaf-1.1/activation.jar:
/usr/lib/javamail-1.4.1/mail.jar
```

Any application using the Oracle S/MIME API must have all the neccessary MIME types registered in its command map.

Some applications, specifically those reading S/MIME entries from a FileDataSource, will need to register the S/MIME file types.

5.3 Developing Applications with Oracle S/MIME

You can develop applications by using the core and supporting classes and interfaces in Oracle S/MIME API and the methods therein.

This section describes selected interfaces and classes in the Oracle S/MIME API and illustrates their use. It includes these topics:

- Core Classes and Interfaces of Oracle S/MIME

- Supporting Classes and Interfaces

- Using the Oracle S/MIME Classes

Selected methods are described as appropriate.

5.3.1 Core Classes and Interfaces of Oracle S/MIME

Oracle S/MIME API consists of multiple core classes and interfaces.

This section describes core classes and interfaces in the Oracle S/MIME API, and explains how to create and parse S/MIME objects.

Core classes and interfaces include:

- Using the oracle.security.crypto.smime.SmimeObject Interface

- Using the oracle.security.crypto.smime.SmimeSignedObject Interface

- Using the oracle.security.crypto.smime.SmimeSigned Class

- Using the oracle.security.crypto.smime.SmimeEnveloped Class
- Using the oracle.security.crypto.smime.SmimeMultipartSigned Class
- Using the oracle.security.crypto.smime.SmimeSignedReceipt Class
- Using the oracle.security.crypto.smime.SmimeCompressed Class

5.3.1.1 Using the oracle.security.crypto.smime.SmimeObject Interface

The `oracle.security.crypto.smime.SmimeObject` interface represents an S/MIME object.

Classes that implement this interface include:

- `SmimeSigned`
- `SmimeEnveloped`
- `SmimeMultipartSigned`
- `SmimeSignedReceipt`
- `SmimeCompressed`

Methods in this interface include:

`String generateContentType ()`

returns the content type string for this S/MIME object. For example:

`"application/pkcs7-mime; smime-type=signed-data"`

`String generateContentType (boolean useStandardContentTypes)`

If the argument is *true*, returns the same as `generateContentType()`; if *false*, returns old-style (Netscape) content type string. For example: "application/x-pkcs7-mime; smime-type=signed-data"

`void writeTo (java.io.OutputStream os, java.lang.String mimeType)`

outputs this object to the specified output stream.

5.3.1.2 Using the oracle.security.crypto.smime.SmimeSignedObject Interface

The `oracle.security.crypto.smime.SmimeSignedObject` interface extends `SmimeObject`, and specifies methods common to all S/MIME signed objects, including `SmimeSigned` and `SmimeMultipartSigned`.

Methods in this interface include:

`Vector getCertificates ()`

Returns the list of certificates included in this S/MIME object's signed content.

`Vector getCRLs ()`

Returns the list of certificate revocation lists in the S/MIME object's signed content.

`javax.mail.internet.MimeBodyPart getEnclosedBodyPart ()`

Returns the document which was signed.

```
oracle.security.crypto.smime.ess.EquivalentLabels getEquivalentLabels
(java.security.cert.X509Certificate signerCert)
```

Returns the `EquivalentLabels` if present or `null`.

```
oracle.security.crypto.smime.ess.ESSSecurityLabel getESSSecurityLabel
(java.security.cert.X509Certificate signerCert)
```

Returns the `ESSSecurityLabel` if present or `null`.

```
oracle.security.crypto.smime.ess.MLExpansionHistory getMLExpansionHistory(
 java.security.cert.X509Certificate signerCert)
```

Returns the `MLExpansionHistory` attribute if present or `null`.

```
oracle.security.crypto.smime.ess.ReceiptRequest getReceiptRequest(
 java.security.cert.X509Certificate signerCert)
```

Returns the `ReceiptRequest` attribute if present or `null`.

```
oracle.security.crypto.smime.ess.SigningCertificate getSigningCertificate(
 java.security.cert.X509Certificate signerCert)
```

Returns the `SigningCertificate`.

```
void verify (oracle.security.crypto.cert.CertificateTrustPolicy trustPolicy)
```

Returns normally if the signed contents include at least one valid signature according to the specified trust policy, otherwise throws an `AuthenticationException`.

```
void verifySignature (java.security.cert.X509Certificate signerCert)
```

Returns normally if the signed contents contain a signature which can be validated by the given certificate, otherwise throws an `AuthenticationException`.

The method can throw a `SignatureException`, if no signature exists corresponding to the given certificate.

5.3.1.3 Using the oracle.security.crypto.smime.SmimeSigned Class

The `oracle.security.crypto.smime.SmimeSigned` class represents an S/MIME signed message (.implements SmimeSignedObject). You may use this class to build a new message or parse an existing one.

Constructors and methods include:

```
SmimeSigned (javax.mail.internet.MimeBodyPart content)
```

Creates a new `SmimeSigned` object, using the specified MIME body part for the contents to be signed.

```
SmimeSigned ()
```

Creates a new empty `SmimeSigned` object, which is useful for building a "certificates-only" S/MIME message.

```
SmimeSigned (InputStream is)
```

Creates a new `SmimeSigned` object by reading its encoding from the specified input stream.

```
void addSignature (java.security.PrivateKey signerKey,
    java.security.cert.X509Certificate signerCert,
    oracle.security.crypto.core.AlgorithmIdentifier digestAlgID)
```

Adds a signature to the message, using the specified private key, certificate, and message digest algorithm.

```
void addSignature (java.security.PrivateKey signerKey,
 java.security.cert.X509Certificate signerCert,
 oracle.security.crypto.core.AlgorithmIdentifier digestAlgID,
      java.util.Date timeStamp)
```

Adds a signature to the message, including a time stamp.

```
void addSignature (java.security.PrivateKey signerKey,
    java.security.cert.X509Certificate signerCert,
    oracle.security.crypto.core.AlgorithmIdentifier digestAlgID,
    SmimeCapabilities smimeCaps)
```

Adds a signature to the message, including S/MIME capabilities.

```
javax.mail.internet.MimeBodyPart getEnclosedBodyPart ()
```

Returns the MIME body part that was signed.

To build a new message, use any of these three constructors:

```
// Create a new S/MIME Signed Message
SmimeSigned sig = new SmimeSigned();

//          -OR-
// Create a new S/MIME Signed Message with a specified MIME body part
MimeBodyPart bp = new MimeBodyPart();
bp.setText("Hello from SendSignedMsg!");
SmimeSigned sig1 = new SmimeSigned(bp);

//          -OR-
// Create a new S/MIME Signed Message with a specified MIME body part
// and a flag switching compression on or off
MimeBodyPart bp = new MimeBodyPart();
bp.setText("Hello from SendSignedMsg!");
boolean useCompression = true;
SmimeSigned sig2 = new SmimeSigned(bp, useCompression);
```

To parse a message, use the constructor that takes a `java.io.InputStream`:

```
InputStream is = Input stream containing message to be parsed
SmimeSigned sig = new SmimeSigned(is);
```

5.3.1.4 Using the oracle.security.crypto.smime.SmimeEnveloped Class

The `oracle.security.crypto.smime.SmimeEnveloped` class represents an S/MIME enveloped message (implements SmimeObject), and may be used to build a new message or parse an existing one.

Constructors and methods include:

```
SmimeEnveloped (javax.mail.internet.MimeBodyPart content,
    oracle.security.crypto.core.AlgorithmIdentifier contentEncryptionAlgID)
```

Creates a new `SmimeEnveloped` object from the specified MIME body part, using the specified content encryption algorithm.

```
SmimeEnveloped (InputStream is)
```

Creates a new `SmimeEnveloped` object by reading its encoding from the specified input stream.

```
void addRecipient (java.security.cert.X509Certificate cert)
```

Encrypts the message for the recipient using the given public key certificate.

```
byte[] getEncryptedContent ()
```

Returns the contents without decrypting.

```
javax.mail.internet.MimeBodyPart getEnclosedBodyPart (
 java.security.PrivateKey recipientKey,
 java.security.cert.X509Certificate recipientCert)
```

Returns the MIME body part for the recipient specified by `recipientCert`, after decryption using the given recipient private key.

Use the following code to build a new message:

```
// Create a new S/MIME Enveloped Message with a specified MIME body part and a
specified content
// encryption algorithm
MimeBodyPart bp = new MimeBodyPart();
bp.setText("Hello from SendSignedMsg!");
AlgorithmIdentifier algId = AlgID.aes256_CBC;
SmimeEnveloped env = new SmimeEnveloped(bp, algId);
```

To parse a message, use the constructor that takes a `java.io.InputStream`:

```
InputStream is = Input stream containing message to be parsed
    SmimeEnveloped env = new SmimeEnveloped(is);
```

5.3.1.5 Using the oracle.security.crypto.smime.SmimeMultipartSigned Class

The `oracle.security.crypto.smime.SmimeMultipartSigned` class represents an S/MIME multi-part signed message. A multipart signed message is intended for email clients that are not MIME-aware. This class can be used to build a new message or parse an existing one.

Constructors and methods include:

```
SmimeMultipartSigned (javax.mail.internet.MimeBodyPart bodyPart,
    oracle.security.crypto.core.AlgorithmIdentifier digestAlgID)
```

Creates a new `SmimeMultipartSigned` message, with the specified MIME body part and message digest algorithm.

```
void addBodyPart (javax.mail.BodyPart part)
```

Inherited from `javax.mail.Multipart`, adds the specified body part to this `SmimeMultipartSigned` object. (See the javax.mail API documentation at http://www.oracle.com/technetwork/java/index-138643.html for more details.)

```
void addSignature (java.security.PrivateKey signerKey,
    java.security.cert.X509Certificate signerCert)
```

Adds a signature to the message, using the specified private key and certificate.

```
void addSignature (java.security.PrivateKey signerKey,
    java.security.cert.X509Certificate signerCert, java.util.Date timeStamp)
```

Adds a signature to the message, using the specified private key and certificate plus a time stamp.

```
void addSignature (java.security.PrivateKey signerKey,
    java.security.cert.X509Certificate signerCert, java.util.Date timeStamp,
    SmimeCapabilities smimeCaps)
```

Adds a signature to the message, using the specified private key and certificate, plus S/MIME capabilities.

```
javax.mail.internet.MimeBodyPart getEnclosedBodyPart ()
```

Returns the MIME body part that was signed.

Use the following code to build a new message:

```
// Create a new S/MIME Multipart Signed Message with a specified
// MIME body part and a specified digest algorithm
MimeBodyPart bp = new MimeBodyPart();
bp.setText("Hello from SendSignedMsg!");
AlgorithmIdentifier algId = AlgID.sha1;
SmimeMutlipartSigned sig = new SmimeMultipartSigned(bp, algId);
```

To parse a message, use the constructor that takes a `javax.activation.DataSource`:

```
DataSource ds = Data source containing message to be parsed
SmimeMultipartSigned sig = new SmimeMultipartSigned(ds);
```

5.3.1.6 Using the oracle.security.crypto.smime.SmimeSignedReceipt Class

The `oracle.security.crypto.smime.SmimeSignedReceipt` class represents an S/MIME wrapped and signed receipt. You may use this class to build a new message or parse an existing one.

To build a new message, use any of these four constructors:

```
// Create a new S/MIME wrapped and signed receipt with the specified receipt,
// the specified digest of the message's signed attributes
// and the addresses of the receipt recipients
ESSReceipt receipt = ESS receipt to include in message
byte [] msgSigDigest = Digest of signed attributes to be included in message
Address [] addresses = Addresses of receipt recipients
SmimeSignedReceipt sig = new SmimeSigned(receipt, msgSigDigest, addresses);

 //         -OR-
// Create a new S/MIME wrapped and signed receipt
// with a specified S/MIME Signed Message containing the receipt
SmimeSignedObject sso = S/MIME signed message containing receipt
SmimeSignedReceipt sig1 = new SmimeSignedReceipt(sso);

//         -OR-
// Create a new S/MIME wrapped and signed receipt with a
// specified S/MIME Signed Message containing the receipt,
```

```
// the signer's certificate and the addresses of the receipt recipients
SmimeSignedObject sso1 = S/MIME signed message containing receipt
X509Certificate signerCert = The message signer's certificate
Address [] addresses1 = Addresses of receipt recipients
SmimeSignedReceipt sig2 = new SmimeSignedReceipt(sso1, signerCert, addresses1);

//           -OR-

// Create a new S/MIME wrapped and signed receipt with a
// specified S/MIME Signed Message containing the receipt,
// the signer's certificate, the addresses of the receipt recipients and
// a specified MLExpansionHistory attribute.
SmimeSignedObject sso1 = S/MIME signed message containing receipt
X509Certificate signerCert = The message signer's certificate
Address [] addresses1 = Addresses of receipt recipients
MLExpansionHistory mlExpansionHistory = The MLExpansionHistory attribute
SmimeSignedReceipt sig2 =
    new SmimeSignedReceipt(sso1, signerCert, addresses1, mlExpansionHistory);
```

To parse a message, use the constructor that takes a `java.io.InputStream`:

```
InputStream is = Input stream containing message to be parsed
SmimeSignedReceipt sig = new SmimeSignedReceipt(is);
```

5.3.1.7 Using the oracle.security.crypto.smime.SmimeCompressed Class

The `oracle.security.crypto.smime.SmimeCompressed` class represents an S/MIME compressed message as defined in RFC 3274. You can use this class to build a new message or parse an existing one.

> **✏️ Note:**
>
> A link to RFC 3274 is available in References.

Use the following code to build a new message:

```
// Create a new S/MIME Compressed Message with a specified MIME body part
MimeBodyPart bp = new MimeBodyPart();
bp.setText("Hello from SendSignedMsg!");
SmimeCompressed comp = new SmimeCompressed(bp);

//            -OR-
// Create a new S/MIME Compressed Message with a specified MIME body part
// and a specified compression algorithm
MimeBodyPart bp = new MimeBodyPart();
bp.setText("Hello from SendSignedMsg!");
AlgorithmIdentifier algId = Smime.id_alg_zlibCompress;
SmimeCompressed comp = new SmimeCompressed(bp, algId);
```

To parse a message, use the constructor that takes a `java.io.InputStream`:

```
InputStream is = Input stream containing message to be parsed
SmimeCompressed comp1 = new SmimeCompressed(is);
```

5.3.2 Supporting Classes and Interfaces

Oracle S/MIME contains supporting interface that defines constants such as algorithm identifiers, content type identifiers, and attribute identifiers. It contains supporting classes that contains static utility methods, verify signatures on signed S/MIME objects, and encapsulate capabilities for an S/MIME object.

This section describes Oracle S/MIME supporting classes and interfaces.

5.3.2.1 Using the oracle.security.crypto.smime.Smime Interface

The `oracle.security.crypto.smime.Smime` interface defines constants such as algorithm identifiers, content type identifiers, and attribute identifiers.

5.3.2.2 Using the oracle.security.crypto.smime.SmimeUtils Class

The `oracle.security.crypto.smime.SmimeUtils` class contains static utility methods.

Methods of this class include:

```
public static FileDataSource createFileDataSource (File file,
     String contentTypeHeader)
public static FileDataSource createFileDataSource (String name,
     String contentTypeHeader)
```

For transparent handling of multipart or multipart/signed S/MIME types, use these methods instead of directly instantiating a `javax.activation.FileDataSource`.

> **Note:**
>
> The default `javax.activation.FileDataSource` included with JAF 1.0.1 does not handle multipart MIME boundaries when used with Javamail 1.1.x.

5.3.2.3 Using the oracle.security.crypto.smime.MailTrustPolicy Class

The `oracle.security.crypto.smime.MailTrustPolicy` class implements a certificate trust policy (`oracle.security.crypto.cert.CertificateTrustPolicy`) used to verify signatures on signed S/MIME objects.

5.3.2.4 Using the oracle.security.crypto.smime.SmimeCapabilities Class

The `oracle.security.crypto.smime.SmimeCapabilities` class encapsulates a set of capabilities for an S/MIME object including, for example, the supported encryption algorithms.

A useful method of this class is:

```
void addCapability(oracle.security.crypto.asn1.ASN1ObjectID capabilityID)
```

which adds the capability with the specified object ID to this set of S/MIME capabilities.

5.3.2.5 Using the oracle.security.crypto.smime.SmimeDataContentHandler Class

The `oracle.security.crypto.smime.SmimeDataContentHandler` class provides the DataContentHandler for S/MIME content types. It implements `javax.activation.DataContentHandler`.

5.3.2.6 Using the oracle.security.crypto.smime.ess Package

The `oracle.security.crypto.smime.ess` package contains the following classes:

Table 5-1   Classes in the oracle.security.crypto.smime.ess Package

Class	Description
ContentHints	Content hints
ContentReference	Content reference
EquivalentLabels	ESS EquivalentLabels
ESSSecurityLabel	An ESS security label
MLData	Represents the MLData element which is used in the MLExpansionHistory attribute
MLExpansionHistory	Mailing list expansion history
ReceiptRequest	An ESS Receipt Request
ReceiptRequest.AllOrFirstTier	An 'AllOrFirstTier' is a part of the 'ReceiptsFrom' field of a ReceiptRequest
SigningCertificate	An ESS Signing Certificate

5.3.3 Using the Oracle S/MIME Classes

You can use the Oracle S/MIME SDK to work with multi-part signed messages, sign messages and authenticate signed messages, create and open digital envelopes, and implement Enhanced Security Services (ESS).

It covers these topics:

- Using the Abstract Class SmimeObject
- Signing Messages
- Creating "Multipart/Signed" Entities
- Creating Digital Envelopes
- Creating "Certificates-Only" Messages
- Reading Messages
- Authenticating Signed Messages
- Opening Digital Envelopes (Encrypted Messages)
- Adding Enhanced Security Services (ESS)

5.3.3.1 Using the Abstract Class SmimeObject

`SmimeObject` is an abstract class representing a fundamental S/MIME message content entity. Subclasses of `SmimeObject` include :

- `SmimeSigned`

- `SmimeEnveloped`

- `SmimeMultipartSigned`

- `SmimeSignedReceipt`, and

- `SmimeCompressed`

One of the characteristics of `SmimeObject` implementations is that they "know their own MIME type" -- that is, they implement the `generateContentType` method. Thus, to place such an object inside a MIME message or body part, follow the same outline that was used in the `SmimeSigned` example:

1. Create the object.

2. Invoke `generateContentType` on the object to obtain a MIME type.

3. Pass the object, together with the generated content type, to the `setContent` method of a `MimeMessage` or `MimeBodyPart` object.

The `SmimeObject` class provides another version of the `generateContentType` method, which takes a boolean parameter. When given *true* as a parameter, `generateContentType` behaves exactly as in the case of no argument. When given *false* as a parameter, `generateContentType` returns the older MIME types required by certain mail clients, including Netscape Communicator 4.0.4. Specifically:

- "application/pkcs7-mime" becomes "application/x-pkcs7-mime"

- "application/pkcs7-signature" becomes "application/x-pkcs7-signature"

5.3.3.2 Signing Messages

Create a signed message, or signed MIME body part, using these steps:

1. Prepare an instance of MimeBodyPart which contains the content you wish to sign. This body part may have any content-type desired. In the following example we create a "text/plain" body part:

   ```
   MimeBodyPart doc = new MimeBodyPart();
   doc.setText("Example signed message.");
   ```

2. Create an instance of `SmimeSigned` using the constructor which takes the MimeBodyPart created earlier as argument.

   ```
   SmimeSigned sig = new SmimeSigned (doc);
   ```

3. Add all desired signatures. For each signature, you need to specify a private key, a certificate for the matching public key, and a message digest algorithm. For example:

   ```
   sig.addSignature (signatureKey, signatureCert, AlgID.sha1);
   ```

In this example we specified the SHA-1 message digest algorithm. Alternatively, we could have specified the MD5 algorithm by passing `AlgID.md5` as the argument.

4. Place your `SmimeSignedObject` into a `MimeMessage` or `MimeBodyPart`, as appropriate. For example:

```
MimeMessage m = new MimeMessage();
m.setContent (sig, sig.generateContentType());
```

or

```
MimeBodyPart bp = new MimeBodyPart();
bp.setContent (sig, sig.generateContentType());
```

The `generateContentType` method used in these examples returns a string identifying the appropriate MIME type for the object, which in this case is:

```
application/pkcs7-mime; smime-type=signed-data
```

With these simple steps, you can now transport the MIME message, place the body part containing S/MIME content into a MIME multipart object, or perform any other operation appropriate for these objects. See the JavaMail API for details.

5.3.3.3 Creating "Multipart/Signed" Entities

The `SmimeMultipartSigned` class provides an alternative way to create signed messages. These messages use the "multipart/signed" mime type instead of "application/pkcs7-mime". The advantage is that the content of the resulting message is readable with non-MIME enabled mail clients, although such clients will not, of course, be able to verify the signature.

Creating a multi-part/signed message is slightly different from creating a signed message. For example, to send a multi-part/signed text message:

```
// create the content text as a MIME body part
MimeBodyPart bp = new MimeBodyPart();
bp.setText("Example multipart/signed message.");
// the constructor takes the signature algorithm
SmimeMultipartSigned sig = new SmimeMultipartSigned(bp, AlgID.sha1);
// sign the content
sig.addSignature(signerKey, signerCert);
// place the content in a MIME message
MimeMessage msg = new MimeMessage();
msg.setContent(sig, sig.generateContentType());
```

The reason for identifying the message digest in the `SmimeMultipartSigned` constructor is that, unlike the case of application/pkcs7-mime signed data objects, multipart/signed messages require that all signatures use the same message digest algorithm.

The `generateContentType` method returns the following string:

```
multipart/signed; protocol="application/pkcs7-signature"
```

5.3.3.4 Creating Digital Envelopes

An S/MIME digital envelope (encrypted message) is represented by the `SmimeEnveloped` class. This is a MIME entity which is formed by encrypting a MIME body part with some symmetric encryption algorithm (eg, Triple-Des or RC2) and a

randomly generated session key, then encrypting the session key with the RSA public key for each intended message recipient.

In the following example, `doc` is an instance of `MimeBodyPart`, which is to be wrapped in an instance of `SmimeEnveloped`, and `recipientCert` is the recipient's certificate.

```
SmimeEnveloped env = new SmimeEnveloped(doc, Smime.dES_EDE3_CBC);
env.addRecipient (recipientCert);
```

You may add any number of envelope recipients by making repeated calls to `addRecipient`.

5.3.3.5 Creating "Certificates-Only" Messages

It is possible to create an S/MIME signed-data object that contains neither content nor signatures; rather, it contains just certificates, or CRLs, or both. Such entities can be used as a certificate transport mechanism. They have the special content type:

```
application/pkcs7-mime; smime-type=certs-only
```

This example shows how to create a signed-data object:

```
X509Certificate cert1, cert2;
SmimeSigned certBag = new SmimeSigned();
certBag.addCertificate(cert1);
certBag.addCertificate(cert2);
```

Now you can pass `certBag` to an appropriate `setContent` method. When `generateContentType` is invoked on `certBag`, it will automatically return a content type with the correct "certs-only" value for the `smime-type` parameter.

5.3.3.6 Reading Messages

The basic JavaMail API technique for extracting Java objects from MIME entities is to invoke the `getContent()` method on an instance of `MimePart`, an interface which models MIME entities and is implemented by the `MimeMesage` and `MimeBodyPart` classes.

The `getContent` method consults the currently installed default command map - which is part of the JavaBeans Activities Framework - to find a data content handler for the given MIME type, which is responsible for converting the content of the MIME entity into a Java object of the appropriate class.

The `mailcap` file provided with your distribution can be used to install the `SmimeDataContentHandler` class, which serves as a data content handler for the following types:

Content Type	Returns Instance Of
application/pkcs7-mime	SmimeSigned or Smime Enveloped
application/pkcs7-signature	SmimeSigned
application/pkcs10	oracle.security.crypto.cert.CertificateRequest
multipart/signed	SmimeMultipartSigned

5.3.3.7 Authenticating Signed Messages

Once you obtain an instance of `SmimeSigned` or `SmimeMutlipartSigned` from `getContent()`, you will naturally want to verify the attached signatures. To explain the available options for signature verification, it is neccessary to discuss the structure of an S/MIME signed message.

The content of a signed S/MIME message is a CMS object of type `SignedData`. Such an object itself has a content - the document to which the signatures are applied - which is the text encoding of a MIME entity. It also contains from zero to any number of signatures, and, optionally, a set of certificates, CRLs, or both, which the receiving party may use to validate the signatures.

The `SmimeSigned` and `SmimeMultipartSigned` classes encapsulate all of this information. They provide two authentication methods: `verifyingSignature` and `verify`.

To verify a particular signature with a certificate already in possession, ignoring any certificate and CRLs attached by the signer, use `verifySignature`. For example:

```
SmimeSignedObject sig =
    (SmimeSignedObject)msg.getContent(); // msg is a Message
sig.verifySignature(cert, msg.getFrom()); // cert is an X509Certificate object
```

If verification fails, the `verifySignature` method throws either a `SignatureException` or an `AuthenticationException` ; otherwise, it returns normally.

Use `verify` to verify that the content contains at least one valid signature; that is, there exists a valid certificate chain, starting from a trusted root CA, and terminating in a certificate for the private key which generated the signature. This method makes use of the attached certificate and CRLs in order to follow certificate chains.

For example, given a trusted certificate authority (CA) certificate already in hand:

```
TrustedCAPolicy trusts = new TrustedCAPolicy();
// if true, need CRL for each cert in chain
trusts.setRequireCRLs(false);
// caCert is an X509Certificate object with CA cert
trusts.addTrustedCA(caCert);
SmimeSignedObject sig = (SmimeSignedObject)msg.getContent();
sig.verify(trusts, msg.getFrom());
```

Like `verifySignature`, `verify` throws an `AuthenticationException` if the signature cannot be verified; otherwise it returns normally. In either case you can recover the document that was signed, which is itself a MIME entity, by invoking `getEnclosedBodyPart()`:

```
MimeBodyPart doc = sig.getEnclosedBodyPart();
```

5.3.3.8 Opening Digital Envelopes (Encrypted Messages)

An S/MIME digital envelope consists of:

- A protected MIME body part, which has been encrypted with a symmetric key algorithm (for example, DES or RC2)
- A randomly generated content encryption key

- Information that allows one or more intended recipients to decrypt the content

For each recipient, this information consists of the content encryption key, itself encrypted with the recipient's public key.

To obtain the encrypted content from an `SmimeEnveloped` object, you need the recipient's private key and the corresponding certificate; the certificate is used as an index into the recipient information table contained in the envelope's data structure.

For example:

```
SmimeEnveloped env = (SmimeEnveloped)msg.getContent();
MimeBodyPart mbp = env.getEnclosedBodyPart(privKey, cert)
// privKey is a PrivateKey object
// cert is an X509Certificate object
```

Passing the private key and the certificate to the `getEnclosedBodyPart` method returns the decrypted content as an instance of `MimeBodyPart`.

The `getContent` method can now be invoked on the `MimeBodyPart` object to retrieve the (now decrypted) content. This content may be a `String` (in the case of an encrypted text message), or any other object such as an `SmimeSigned`.

5.3.3.9 Adding Enhanced Security Services (ESS)

You can add the ESS services `ReceiptRequests`, `SecurityLabels`, and `SigningCertificates` to an S/MIME signed message by adding them to the `signedAttributes` of a signature.

```
// Create a Signed Message
SmimeSigned sig = new SmimeSigned();
    AttributeSet signedAttributes = new AttributeSet();
```

5.3.3.9.1 Requesting a Signed Receipt with ESS

`oracle.security.crypto.smime.ess.ReceiptRequest` supports the receipt request service.

To request a signed receipt from the recipient of a message, add a `receiptRequest` attribute to the `signedAttributes` field while adding a signature:

```
ReceiptRequest rr = new ReceiptRequest();
.........
signedAttributes.addAttribute(Smime.id_aa_receiptRequest, rr);
```

5.3.3.9.2 Attaching a Security Label with ESS

`oracle.security.crypto.smime.ess.ESSSecurityLabel` provides the security label service.

To attach a security label to a message, add an `ESSSecurityLabel` attribute to the `signedAttributes` field while adding a signature:

```
ESSSecurityLabel sl = new ESSSecurityLabel();
    .........
    signedAttributes.addAttribute(Smime.id_aa_securityLabel, sl);
```

ORACLE

5.3.3.9.3 Attaching a Signing Certificate with ESS

oracle.security.crypto.smime.ess.SigningCertificate enables you to attach a signing certificate.

To attach a signing certificate to a message, add a `SigningCertificate` attribute to the `signedAttributes` field while adding a signature:

```
SigningCertificate sc = new SigningCertificate();
    .........
    signedAttributes.addAttribute(Smime.id_aa_signingCertificate, sc);
```

Use the `signedAttributes` while adding a signature:

```
sig.addSignature(signerKey, signerCert, digestAlgID, signedAttributes);
```

The ESS signed receipts are generated using the `SmimeSignedReceipt` class in the `oracle.security.crypto.smime` package, in a manner similar to using a `SmimeSigned` class, except that the content that is signed is an `oracle.security.crypto.cms.ESSReceipt` object.

5.3.3.10 Processing Enhanced Security Services (ESS)

An S/MIME signed receipt must have correctly set content type parameters for the data content handlers to recognize it. If the content type parameters are missing, the signed receipt is treated as a signed message.

5.4 The Oracle S/MIME Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods available in Oracle S/MIME.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

6

Oracle PKI SDK

Public key infrastructure (PKI) is a security architecture that provides an increased level of confidence when exchanging information over the Internet. Oracle PKI SDK provides packages for PKI, LDAP, and timestamp functions for developing PKI-aware applications.

We explain PKI features and the various sub-packages of Oracle PKI:

- Oracle PKI CMP SDK
- Oracle PKI OCSP SDK
- Oracle PKI TSP SDK
- Oracle PKI LDAP SDK

6.1 Oracle PKI CMP SDK

You can use Oracle public key infrastructure (PKI) Software Development Kit (SDK) for certificate management protocol (CMP). Oracle PKI CMP SDK allows Java developers to quickly implement certificate management functionality such as issuing and renewing certificates, creating and publishing CRLs, and providing key recovery capabilities.

- Oracle PKI CMP SDK Features and Benefits
- Setting Up Your Oracle PKI CMP SDK Environment
- The Oracle PKI CMP SDK Java API Reference

6.1.1 Oracle PKI CMP SDK Features and Benefits

Oracle PKI CMP SDK provides packages that implement certificate management protocol (CMP) as described in RFC 2510, and certificate request message format (CRMF) as described in RFC 2511.

The Oracle PKI CMP SDK provides the following features and functionality:

- Oracle PKI CMP SDK conforms to RFC 2510, and is compatible with other products that conform to this certificate management protocol (CMP) specification. RFC 2510 defines protocol messages for all aspects of certificate creation and management.

- Oracle PKI CMP SDK conforms to RFC 2511, and is compatible with other products that conform to this certificate request message format (CRMF) specification. RFC 2511 describes the Certificate Request Message Format (CRMF), which is used to convey X.509 certificate requests to a Certification Authority (CA).

The Oracle PKI CMP SDK toolkit contains the following packages:

- The `oracle.security.crypto.cmp` package provides classes that implement certificate management protocol (CMP) as described in RFC 2510, and certificate request message format (CRMF) as described in RFC 2511.

- The `oracle.security.crypto.cmp.attribute` package provides attribute classes for registration controls, registration information, and general information. This package includes the following classes and their subclasses:

  - `RegistrationControl`

  - `RegistrationInfo`

  - `InfoTypeAndValue` (which extends `oracle.security.crypto.cert.AttributeTypeAndValue`)

- The `oracle.security.crypto.cmp.transport` package provides classes for CMP and CRMF transport protocols. It includes the `TCPMessage` class and its specific message-type subclasses.

6.1.2 Setting Up Your Oracle PKI CMP SDK Environment

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`. In order to use Oracle PKI CMP SDK, your system must have the Java Development Kit (JDK) version 1.6 or higher. Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files.

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`

- `osdt_cert.jar`

- `osdt_cms.jar`

- `osdt_cmp.jar`

For example, your classpath may look like:

```
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cms.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cmp.jar
```

> ✎ **See Also:**
>
> Setting the CLASSPATH Environment Variable

6.1.3 The Oracle PKI CMP SDK Java API Reference

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods available in Oracle PKI CMP SDK.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

6.2 Oracle PKI OCSP SDK

Oracle PKI OCSP SDK allows Java developers to quickly develop Online Certificate Status Protocol (OCSP) enabled client applications and OCSP responders that conform to RFC 2560 specifications.

This section contains the following topics:

- Oracle PKI OCSP SDK Features and Benefits
- Setting Up Your Oracle PKI OCSP SDK Environment
- The Oracle PKI OCSP SDK Java API Reference

6.2.1 Oracle PKI OCSP SDK Features and Benefits

Oracle PKI OCSP SDK conforms to RFC 2560 specifications. It provides classes and methods to constructing OCSP request messages, responses, and OSCP server implementations.

Oracle PKI OCSP SDK provides the following features and functionality:

- Oracle PKI OCSP SDK conforms to RFC 2560 and is compatible with other products that conform to this specification, such as Valicert's Validation Authority. RFC 2560 specifies a protocol useful in determining the current status of a digital certificate without requiring CRLs.

- The Oracle PKI OCSP SDK API provides classes and methods for constructing OCSP request messages that can be sent through HTTP to any RFC 2560 compliant validation authority.

- The Oracle PKI OCSP SDK API provides classes and methods for constructing responses to OCSP request messages, and an OCSP server implementation that you can use as a basis for developing your own OCSP server to check the validity of certificates you have issued.

6.2.2 Setting Up Your Oracle PKI OCSP SDK Environment

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`. In order to use Oracle PKI OCSP SDK, your system must have the Java Development Kit (JDK) version 1.6 or higher. Also, make sure that your `PATH` environment variable includes the Java bin directory. Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files.

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- osdt_ocsp.jar

For example:

```
setenv CLASSPATH $CLASSPATH:$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_core.jar:
$ORACLE_HOMEmodules/oracle.osdt_11.1.1/osdt_cert.jar:
$ORACLE_HOMEmodules/oracle.osdt_11.1.1/osdt_ocsp.jar
```

> ✎ **See Also:**
>
> Setting the CLASSPATH Environment Variable.

6.2.3 The Oracle PKI OCSP SDK Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods available in Oracle PKI OCSP SDK.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

6.3 Oracle PKI TSP SDK

Oracle PKI TSP SDK allows Java developers quickly implement time-stamping functionality within a public key infrastructure (PKI) framework.

This section contains the following topics:

- Oracle PKI TSP SDK Features and Benefits
- Setting Up Your Oracle PKI TSP SDK Environment
- The Oracle PKI TSP SDK Java API Reference

6.3.1 Oracle PKI TSP SDK Features and Benefits

Oracle PKI TSP SDK conforms to RFC 3161 and is compatible with other products that conform to this time stamp protocol (TSP) specification. It provides a sample implementation of a TSA server which you can use for testing TSP request messages, or as a basis for developing your own time stamping service.

Oracle PKI TSP SDK contains the following classes and interfaces:

Table 6-1   Oracle PKI TSP SDK Classes and Interfaces

Class or Interface Name	Description
`TSP` Interface	Defines various constants associated with the Time Stamp Protocol (TSP).
`HttpTSPRequest` Class	Implementation of a TSP request message over HTTP.
`HttpTSPResponse` Class	Implementation of a TSP response message over HTTP.
`MessageImprint` Class	This class represents a `MessageImprint` object as defined in RFC 3161.
`TSAPolicyID` Class	This class represents a `TSAPolicyID` object as defined in RFC 3161.
`TSPContentHandlerFactory` Class	A content handler for TSP over HTTP.
`TSPMessage` Class	A TSP message.
`TSPTimeStampReq` Class	A TSP message of type `TimeStampReq` as defined in RFC 3161.
`TSPTimeStampResp` Class	A TSP message of type `TimeStampResp` as defined in RFC 3161.

Table 6-1   (Cont.) Oracle PKI TSP SDK Classes and Interfaces

Class or Interface Name	Description
`TSPUtils` Class	Defines various utility methods for the `oracle.security.crypto.tsp` package.

6.3.2 Setting Up Your Oracle PKI TSP SDK Environment

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`. In order to use Oracle PKI TSP SDK, your system must have the Java Development Kit (JDK) version 1.6 or higher. Also, make sure that your `PATH` environment variable includes the Java bin directory. Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files.

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`

- `osdt_cert.jar`

- `osdt_cms.jar`

- `osdt_cmp.jar`

- `osdt_tsp.jar`

For example:

```
setenv CLASSPATH $CLASSPATH:$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_core.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cert.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cms.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cmp.jar;
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_tsp.jar
```

> ✏️ **See Also:**
>
> Setting the CLASSPATH Environment Variable.

6.3.3 The Oracle PKI TSP SDK Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods available in Oracle PKI TSP SDK.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

6.4 Oracle PKI LDAP SDK

Oracle PKI LDAP SDK allows Java developers quickly implement operations that involve publishing and retrieving digital certificates from a directory server.

This section contains the following topics:

- Oracle PKI LDAP SDK Features and Benefits
- Setting Up Your Oracle PKI LDAP SDK Environment
- The Oracle PKI LDAP SDK Java API Reference

6.4.1 Oracle PKI LDAP SDK Features and Benefits

Oracle PKI LDAP SDK provides classes and methods to access, validate, and manage a digital certificate within an LDAP directory.

Oracle PKI LDAP SDK provides facilities for accessing a digital certificate within an LDAP directory. Some of the tasks you can perform with Oracle PKI LDAP SDK are:

- Validating a user's certificate in an LDAP directory
- Adding a certificate to an LDAP directory
- Retrieving a certificate from an LDAP directory
- Deleting a certificate from an LDAP directory

The `oracle.security.crypto.LDAP` package contains two classes:

- `LDAPCertificateValidator`, which validates a user certificate by checking whether it exists in its subject's LDAP directory entry
- `LDAPUtils`, which is a collection of methods to add, retrieve, and remove certificates from a subject's LDAP directory entry

6.4.2 Setting Up Your Oracle PKI LDAP SDK Environment

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`. You must have Java Development Kit (JDK) version 1.6 or higher and Oracle's Java Naming and Directory Interface (JNDI) version 1.2.1 or higher in your system. Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files.

To use Oracle PKI LDAP SDK, your system must have the following:

- Java Development Kit (JDK) version 1.6 or higher. Also, make sure that the Java `bin` directory is added to your PATH environment variable.
- Oracle's Java Naming and Directory Interface (JNDI) version 1.2.1 or higher. You must add all of the JNDI jar files to your `CLASSPATH`.

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- `osdt_ldap.jar`
- `jndi.jar`, `ldapbp.jar`, `ldap.jar`, `jaas.jar`, and `providerutil.jar` (Oracle's Java Naming and Directory Interface (JNDI))

For example:

```
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_ldap.jar;
```

> **See Also:**
>
> Setting the CLASSPATH Environment Variable.

6.4.3 The Oracle PKI LDAP SDK Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods available in Oracle PKI LDAP SDK.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

7

Oracle XML Security

XML security refers to standard security requirements of XML documents such as confidentiality, integrity, message authentication, and non-repudiation. You can setup your environment and use Oracle XML Securityto fulfil these standard security requirements of XML documents.

The need for digital signature and encryption standards for XML documents prompted the World Wide Web Consortium (W3C) to put forth an XML Signature standard and an XML Encryption standard.

We cover features and provide code samples for implementing Oracle XML Security:

- Oracle XML Security Features and Benefits
- Setting Up Your Oracle XML Security Environment
- Signing Data with Oracle XML Security
- Verifying XML Data
- Understanding how Data is Encrypted
- Understanding Data Decryption with Oracle XML Security
- Understanding and Using Element Wrappers in the OSDT XML APIs
- Signing Data with the Oracle XML Security API
- Verifying Signatures with the Oracle XML Security API
- Encrypting Data with the Oracle XML Security API
- Decrypting Data with the Oracle XML Security API
- Common XML Security Questions
- Best Practices for Oracle XML Security
- The Oracle XML Security Java API Reference

> ✎ **See Also:**
>
> The following resources provide more information about XML and XML standards:
>
> - W3C's Recommendation for XML Signatures
> - W3C's Recommendation for XML Encryption
>
> Links to these resources are available in References.

7.1 Oracle XML Security Features and Benefits

Oracle Security Developer Tools provide a complete implementation of the XML Signature and XML Encryption specifications, and supports Signature Algorithms, Digest Algorithms, Data Encryption Algorithms, Key Encryption and Key Wrapping Algorithms, and Transforms.

These algorithms are:

Signature Algorithms

- DSA with SHA1
- RSA with SHA1
- HMAC-SHA1

Digest Algorithms

- MD5
- SHA1
- SHA256
- SHA512

Transforms

- Canonicalization – Canonical XML 1.0, Canonical XML 1.1, exclusive Canonical XML 1.0, (all forms are supported with and without comments)
- XSLT
- XPath Filter
- XPath Filter 2.0
- Base64 Decode
- Enveloped Signature
- Decrypt Transform

Data Encryption Algorithms

- AES-128 in CBC mode
- AES-192 in CBC mode
- AES-256 in CBC mode
- DES EDE in CBC mode

Key Encryption and Key Wrapping Algorithms

- RSAES-OAEP-ENCRYPT with MGF1
- RSAES-PKCS1-v1_5
- AES-128 Key Wrap
- AES-192 Key Wrap

- AES-256 Key Wrap
- DES-EDE Key Wrap

7.2 Setting Up Your Oracle XML Security Environment

Setup your Oracle XML Security environment by installing Oracle Security Developer Tools and JDK, and setting up CLASSPATH environment variable for jar files.

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME/modules/oracle.osdt_11.1.1`.

System Requirements

In order to use Oracle XML Security, you must have JDK 5 or higher.

CLASSPATH Environment Variable

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- `osdt_xmlsec.jar` (This is the main jar containing all the Oracle XML Security classes.)
- `org.jaxen_1.1.1.jar`, which is located in `$ORACLE_HOME/modules/`

  Oracle XML Security relies on the Jaxen XPath engine for XPath processing.

> ✏ **See Also:**
>
> Setting the CLASSPATH Environment Variable

7.3 Signing Data with Oracle XML Security

Using the Oracle Security Developer Tools Oracle XML Security API, you can sign an XML document, a fragment of an XML document, or some binary data.

This section explains the concepts behind data signing.

The basic steps are as follows:

1. Identify what to sign and where to place the signature.
2. Decide on a signing key.

> ✏ **See Also:**
>
> For details of data signing with the Oracle XML Security APIs, see Signing Data with the Oracle XML Security API through Decrypting Data with the Oracle XML Security API.

7.3.1 Identifying What to Sign

As a first step, you must identify the data that you need to sign and where your signature will be placed. You can do this by adding the `xml:id` attribute to the information element. The Signature uses this attribute to refer to the element.

The most common case of signing is when you are signing a part of a document, and the signature is also placed in the same document. For this you need to decide how you refer to that part. The simplest way is to use an ID, for example:

```
<myDoc>
  <importantInfo  xml:id="foo1">
    ...
  </importantInfo>
  <dsig:Signature>
    ...
      <dsig:Reference URI="#foo1">
    ...
  </dsig:Signature>
</myDoc>
```

In this example myDoc is the entire document, of which you only want to sign the `<importantInfo>` element, and the signature is placed right after the `<importantInfo>` element. The `<importantInfo>` has an `xml:id` attribute, which the Signature uses to refer to it.

`xml:id` is a generic identifying mechanism.

If your schema does not allow you to add this attribute to your `<importantInfo>` element, you can instead use an Xpath to refer to it.

7.3.1.1 Determining the Signature Envelope

This example uses a "disjoint" signature where the signature and element to be signed are completely separate.

There are two other ways of signing "enveloped":

- where the signature element is the child/descendant of the element to be signed, and
- "enveloping" where the signature element is a parent/ancestor of the element to be signed.

Here is an example of Enveloped Signing:

```
<myDoc>
  <importantInfo  xml:id="foo1">
    ...
    <dsig:Signature>
      ...
      <dsig:Reference URI="#foo1">
        ...
        <Transform Algorithm="...enveloped-signature">
        ...
      </dsig:Reference>
      ...
```

```
        </dsig:Signature>
      ...
    </importantInfo>
</myDoc>
```

When you use enveloped signature, you must use the `EnvelopedSignatureTransform` to exclude the signature itself from the signature calculation, otherwise the very act of generating a signature changes the content of the `importtantInfo` element, and the verification will fail.

7.3.1.2 Deciding How to Sign Binary Data

It is also possible to sign binary data. To do this you must make the binary data available through a URI. Oracle XML Security allows any URIs that can be resolved by the JDK, such as http:, file:, and zip: URIs.

You need to create a separate XML document which will hold the `Signature` element, and this signature will refer to the binary data using this URI.

You can sign XML data using this mechanism as well, provided your XML data can be accessed by a URI. But for XML you can decide to either treat it as binary data and sign as is, or apply canonicalization and sign as XML. To apply canonicalization you need to add a canonicalization transform.

If your binary data is present as a base64 encoded string in your XML document, you can use an ID-based or an Xpath-based reference to it, and then use a `Base64DecodeTransform` to decode the data and sign the binary.

```
<myDoc>
  <importantBinaryData  xml:id="foo1">
    XJELGHKLasNDE12KL=
  </importantBinaryData>
  <dsig:Signature>
    ...
      <dsig:Reference URI="#foo1">
          ...
          <Transform Algorithm="...base64">
          ...
      </dsig:Reference>
      ...
  </dsig:Signature>
</myDoc>
```

> **Note:**
>
> External URI dereferencing can be very insecure. For example, say you are running Oracle Security Developer Tools code inside a server, and you verify an incoming message; if this message has an external URI reference, it is essentially causing your server to read from the file or from external web sites. This can lead to denial of service attacks and cross-site scripting.
>
> This is why External URI dereferencing is disabled by default. You need to set the JVM property `osdt.allow.externalReferences` (or set `osdt.allow.all`) to allow external URI dereferencing.

7.3.1.3 Signing Multiple XML Fragments with a Signature

You can include multiple XML fragments into the same signature. For example, you can have two ID-based references, and include both of them in the same signature. Or you can use an Xpath expression which resolves to multiple subtrees.

You can also mix and match local ID-based references with remote URI references, and have all of them in the same signature.

In fact it is recommended that you include multiple parts into the same signature to cryptographically bind them together; for example, if you are using an XML signature to sign a purchase order approval, you must include the items that are being purchased, the user who approved it, and time it was approved, all in the same signature. If you forget to include the user, somebody can potentially steal this message, change the user name, resubmit it, and the signature will still verify.

7.3.1.4 Excluding Elements from a Signature

At times you may need to sign subtrees with exclusions, rather than signing complete subtrees; to achieve this you need to use an Xpath expression.

7.3.2 Deciding on a Signing Key

Once you have decided on what to sign, and how to reference it, you can decide on a signing key, by using a X509Certificate, a symmetric key, or a raw asymmetric signing key, like a DSA, RSA, or DH key.

These options are:

- Use a X509Certificate.

  This is the most common mechanism. You sign with the private key, and anybody who has your public key can verify with it.

- Use a raw asymmetric signing key, like a DSA, RSA, or DH key.

  When you are signing with an X509certificate, you are in fact signing with the DSA/RSA/DH signing key that is associated with the certificate. You can also sign with DSA/RSA/DH signing key that is not associated with any certificate, although there is no good reason for doing so.

- Use a symmetric key.

  You can also do HMAC signing with a symmetric key. This is useful when you and the verifier already share a symmetric key; it could be a key derived from a password, or it could be from a kerberos system which uses symmetric keys. The Oracle Security Developer Tools WS Security APIs provide explicit APIs for password-based keys and kerberos keys.

7.3.2.1 Setting Up Key Exchange

The key exchange needs to happen out of band. For example, if you signing with a certificate, the receiver should already be set up with the trust points, so that the receiver can verify your certificate. Or if you are signing with a symmetric key, the receiver should already know this symmetric key. The XML Signature specification does not define this initial key exchange mechanism.

7.3.2.2 Providing a Receiver Hint

You also need to provide a hint so that the receiver knows how to verify your signature. This will be in the `<dsig:KeyInfo>` tag inside the `<dsig:Signature>`. This can be accomplished in different ways:

- You can provide no hint at all. This perfectly acceptable, if you have already communicated the key to the receiver, and the receiver is expecting all signatures to be signed by this key. However this is not a likely situation.

- When signing with an `X509Certificate`, you can provide one or more of the following:

  – The entire `X509Certificate`. This is the most common usage.

  – The `Subject` DN of the certificate – This is useful when the receiver has access to a LDAP directory, and it can look up the certificate based on the DN.

  – The `SubjectKeyIdentifier` or the IssuerDN/Serial number pair – This is useful when the receiver is only expecting a signatures from a set of certificates, and it every time it has to verify a signature, it can loop over all the certificates and find the one with matching SKI or `IssuerSerial`.

- When signing with a raw asymmetric key, you can provide the actual values of the RSA/DSA/DH public key. This is not recommended as the receiver cannot verify the key; alternatively, if you include the certificate, the receiver can do PKIX processing and verify it; that is, the receiver can check for certificate validity and check against an OCSP or CRL.

- When signing with a symmetric key, you can provide a key name. This is just a string that conveys some information that the receiver can use to retrieve/construct the symmetric key.

7.4 Verifying XML Data

You can verify XML data by searching for the signature element and fetching the verification key.

This section explains the concepts behind data verification.

Once you understand how to create a signature, you can use similar steps to verify the signature. The basic steps are as follows:

1. Search for the signature element, and check what was signed

   When you first search for the signature element in the XML document. Oracle XML Security provides a method (put in link here) to list the elements included in this signature. Verify that those are the elements you were expecting to be signed.

2. Fetch the verification key

   Next identify the key with which the signature was signed. To do this, examine the `<dsig:KeyInfo>` for the certificate, raw public key, or symmetric key that should be used for verification.

> **✎ See Also:**
>
> For details of data verification with the Oracle XML Security APIs, see

7.5 Understanding how Data is Encrypted

You can encrypt an XML document, a fragment of an XML document or some binary data by applying an encryption key.

This section explains the concepts behind data encryption.

The basic steps are as follows:

- Identifying what to Encrypt
- Decide on the Encryption Key

> **✎ See Also:**
>
> For details of data encryption with the Oracle XML Security APIs, see
> Encrypting Data with the Oracle XML Security API.

7.5.1 Identifying what to Encrypt

The most common encryption scenario is to encrypt and replace. When you are encrypting a part of the document, replace the document with the encrypted bytes.

For example:

```
<myDoc>
  <importantInfo>
    ...
  </importantInfo>
</myDoc>
```

If you encrypt the importantInfo element, it will look like this:

```
<myDoc>
  <xenc:EncryptedData>
    ...
  </xenc:EncryptedData>
</myDoc>
```

Here the entire `<importantInfo>` and all its contents are replaced by an `EncryptedData` element which essentially contains a large base64 string, which is the base64 encoding of the encrypted `<importantInfo>` element.

In this mode the `<importantInfo>` element is completely hidden, and the receiver has no way of knowing the contents until it is decrypted.

7.5.1.1 Using the Content Only Encryption Mode

There is also a "Content only" encryption mode where the element tag itself is not encrypted, but all its contents are encrypted.

```
<myDoc>
  <importantInfo>
    <xenc:EncryptedData>
      ...
    </xenc:EncryptedData>
  </importantInfo>
</myDoc>
```

Use the "Content Only" mode if it is appropriate for everyone to know that the `<importantInfo>` exists; only the intended party will know how to decrypt and look at the contents of the `<importantInfo>` element.

7.5.1.2 Encrypting Binary Data

If you are encrypting binary data present as a base64 encoded string, you can encrypt it as if it were regular XML data.

However if you are encrypting external binary data (that is, data outside the XML document), your options depend on where you will store the encrypted data.

You can store the data externally or inside the encrypted data element.

One option is to store the encrypted data externally as well. For SOAP Attachments refer to the WS Security SOAP Attachments (insert link) which specifies a mechanism to encrypt attachments and store the encrypted data back as an attachment.

To store the encrypted data externally, you need to use a `xenc:CipherReference`, which is a subelement of `xencEncryptedData` and uses a URI to refer to the encrypted bytes.

The other option is to store the encrypted bytes inside the `EncryptedData`, just as you would with in-place XML encryption.

7.5.2 Decide on the Encryption Key

You can choose a random symmetric key and encrypt your data. Then you can encrypt this symmetric key with your asymmetric key.

This is very similar to the task of deciding the signing key (see section Deciding on a Signing Key) except that you never directly encrypt with an asymmetric key. Instead, you usually:

1. choose a random symmetric key,

2. encrypt your data with this key,

3. encrypt this random symmetric key with your asymmetric key, and

4. send both the encrypted data and encrypted key to the receiver.

Even with a symmetric key, you can still choose to:

1. generate a random symmetric key,

2. encrypt this random symmetric key with your symmetric key and

3. send both the encrypted data key and the encrypted key to the receiver

To use this encrypted key mechanism, you need to decide where to place the
`xenc:EncryptedKey` in your document.

- If you only have one `encryptedData` element, place the `EncryptedKey` in the
  `KeyInfo` of the `EncryptedData`.

- Otherwise, place them separately and have one refer to the other.

Use the `<dsig:KeyInfo>` inside the `EncryptedKey` to refer to the certificate,
asymmetric key, or key name that can be used to decrypt the `EncryptedKey`.

7.6 Understanding Data Decryption with Oracle XML Security

Data decryption follows the same process as for data encryption, but in reverse. You
need to decrypt the random symmetric key, and then use this key to decrypt the data.

The basic steps are as follows:

If the data was encrypted with a simple encryption in place, locate the `EncryptedData`
element and look at its `KeyInfo`.

If it is directly encrypted with a known symmetric key, decrypt it.

Otherwise if it is encrypted with a random symmetric key:

1. locate the corresponding `EncryptedKey`,

2. decrypt it first, and

3. use this decrypted random symmetric key to decrypt the `EncryptedData`.

> ✎ **See Also:**
>
> For details of data decryption with the Oracle XML Security APIs, see

7.7 Understanding and Using Element Wrappers in the OSDT XML APIs

All the XML-based Oracle Security Developer Tools APIs like Oracle XML Security
and Oracle Web Services Securityuse a wrapper concept, in which for each XML
element, there is a corresponding Java wrapper class.

For example, the `<dsig:Signature>` XML element corresponds to the `XSSignature`
class. All these wrapper classes inherit from `XMLElement`, and they contain only one
data member, which is the pointer to the corresponding DOM element.

This section shows how to work with wrapper objects in the Oracle Security Developer
Tools APIs. Topics include:

- Constructing the Wrapper Object
- Obtaining the DOM Element from the Wrapper Object
- Parsing Complex Elements
- Constructing Complex Elements

7.7.1 Constructing the Wrapper Object

You can invoke the constructor to construct a wrapper object from a DOM element. If the DOM element does not exist, either you can first create a DOM element, and then use the constructor, or you can use a newInstance method.

To construct a wrapper object from the DOM element, simply invoke the constructor.

For example:

```
Element sigElem =
    (Element)doc.getElementsByTagNameNS(XMLURI.ns_dsig, "Signature").item(0);
XSSignature sig = new XSSignature(sigElem);
```

To construct a Wrapper object when the DOM element does not exist, you can either:

- create a DOM element, and use the above method, or
- use a newInstance method

```
XSSignature sig = XSSignature.newInstance(doc, null);
```

This internally achieves the same ends, that is, it creates a `<dsig:Signature>` DOM element, without appending it anywhere, then creates a wrapper object on top of the element. You will need to append this element somewhere in your document.

For some wrapper classes, there is no `newInstance` method and you need to call a constructor that takes the document object.

```
XSSignedInfo sigInfo =  new XSSignedInfo(doc, null);
```

Another way to create the wrapper object from the element is to call the `XMLUtils.getInstance` method:

```
XSSignature sig = (XSSignature)XMLUtils.getInstance(sigElem);
```

The Oracle Security Developer Tools APIs internally maintain a table associating element names to wrapper class names. The `XMLUtils.getInstance` uses this table to invoke the appropriate constructor and return an instance of that wrapper class.

7.7.2 Obtaining the DOM Element from the Wrapper Object

You can use the method `XMLElement.getElement()` to get the underlying DOM element from the wrapper object.

The underlying DOM element is readily available. All wrapper classes extend from `XMLElement` which provides a method, `XMLElement.getElement()`, to get the underlying DOM element.

7.7.3 Parsing Complex Elements

For complex elements containing a hierarchy of subelements, there are an equivalent hierarchy of wrapper objects.

For example, suppose you have an incoming document containing a signature:

```
<dsig:Signature>
  <dsig:SignedInfo>
    <dsig:CanonicalizationMethod ... />
     ...
  <dsig:SignedInfo>
  <dsig:SignatureValue>..</dsig:SignatureValue>
  ...
</dsig:Signature>
```

Most of these elements have a corresponding wrapper class, such as `dsig:Signature` `-> XSSignature, dsig:SignedInfo -> XSSignedInfo, dsig:SignatureValue ->` `XSSignatureValue` and so on.

But when you construct the `XSSignedInfo` object from the `dsig:Signature` DOM element, it does not construct any of the child objects, in fact it does not even look at any of the child elements. The new `XSSignature(sigElem)` is a quick call which simply creates an object with the data member pointing to the `sigElem`. The child objects are created every time. So when you call `XSSignature.getSignedInfo()` it searches the child elements of `dsig:Signature` to find the `dsig:SignedInfo` element, constructs a wrapper object on that element, and returns it.

This wrapper object is not stored anywhere. So if you invoke `XSSignature.getSignedInfo()` again, it does the same thing, returning a different instance of the `SignedInfo` object; however both these objects point to the same DOM element, so they behave exactly the same way even though they are different instances.

> **Note:**
>
> Remember that the DOM is the source of truth, while the wrapper objects are throwaway objects. The `get` methods always create new wrapper objects, and if you modify the underlying DOM, the wrapper objects always see the most recent changes.

7.7.4 Constructing Complex Elements

You can create individual wrapper objects and assemble them by using the `set` methods to construct a complex element.

Consider the same example as before, but now instead of the signature present in an incoming document, you want to create a document containing a signature and send this document to someone.

```
<dsig:Signature>
  <dsig:SignedInfo>
     ...
```

```
    <dsig:SignedInfo>
    ...
</dsig:Signature>
```

To construct this complex element, you need to create individual wrapper objects and assemble them using `set` methods.

For example:

```
XSSignature sig = XSSignature.newInstance(doc, null);
XSSignedInfo sigInfo = new XSSignedInfo(doc, null);
sig.setSignedInfo(sigInfo);
```

Remember that the DOM is always the source of truth; the set methods do not store or copy the passed-in wrapper object, they just modify the underlying DOM.

So in this case the `setSignedInfo` gets the `dsig:SignedInfo` element, and makes that a child of the `dsig:Signature` element. So after invoking `setSignedInfo(sigInfo)`, if you do `sigInfo = null`, it will not affect anything.

Finally you need to insert the top-level object somewhere into your DOM:

```
elem.appendChild(sig.getElement());
```

7.8 Signing Data with the Oracle XML Security API

With Oracle XML Security APIs, you can create signatures for the XML data elements.

This section describes techniques for signing data with the Oracle XML Security APIs.

Topics include:

- Creating a Detached Signature, Basic Procedure
- Using Variations on the Basic Signing Procedure

7.8.1 Creating a Detached Signature, Basic Procedure

You can create a detached signature with an identified XML element, an ID attribute added to the element, and a signing key and certificate.

To create a detached signature like this:

```
<myDoc>
  <importantInfo  xml:id="foo1">
    ...
  </importantInfo>
  <dsig:Signature>
    ...
      <dsig:Reference URI="#foo1">
    ...
  </dsig:Signature>
</myDoc>
```

You need to do this:

```
// assume you have your data set up in doc
Document doc = ...
Element impElem = ...
```

```
// Now put an ID on the importantInfo element
impElem.setAttributeNS(XMLURI.ns_xml, "xml:id", "foo1");

// Then get the signing key and certificate from
// somewhere - e.g. you can load them from a keystore
PrivateKey signKey = ...
X509Certificate signCert = ...

// Create the Signature object
XSSignature sig = XSSignature.newInstance(doc, null);

// Create the SignedInfo object
// Normally you should use exclusive canonicalization
//     alg_exclusiveC14N
// Depending on the type of your private key DSA or RSA
//     use dsaWithSHA1 or rsaWithSHA1
XSSignedInfo sigInfo = sig.createSignedInfo(
  XMLURI.alg_exclusiveC14N, XMLURI.alg_rsaWithSHA1, null)
sig.setSignedInfo(sigInfo);


// Create a Reference object to the importantInfo element
// You need to specify the id which you set up earlier,
//  and also a digestMethod
XSReference ref = sig.createReference(null, "#foo1", null,
   XMLURI.alg_sha1);
sigInfo.addReference(ref);
// Create an exclusive c14n Transform object
// If you do not add this transform object, it will use
// inclusive by default
XSAlgorithmIdentifier transform =
  new XSAlgorithmIdentifier(doc, "Transform",
  XMLURI.alg_exclusiveC14n);
ref.addTransform(transform);


// Create a KeyInfo object
XSKeyInfo keyInfo = sig.createKeyInfo();
sig.setKeyInfo(keyInfo);

// Create an X509Data element for your signingCert, inside
//  this keyingo
X509Data x509 = keyInfo.createX509Data(signingCert);
keyInfo.addKeyInfoData(x509);

// Everything is setup, now do the actual signing
// This will actually do all the canonicalization,
// digesting, signing etc
sig.sign(signKey, null);

// Finally insert the signature somewhere in your document
doc.getDocumentElement().appendChild(sig.getElement());
```

> **✏️ Note:**
>
> After creating a child Wrapper object, you must call a set or add method to put it in its parent, and also remember to insert the top level `Signature` object into your document.

7.8.2 Using Variations on the Basic Signing Procedure

While creating a signature you can include multiple references, enveloped signatures, XPath expressions, certificate hints, and HMAC key signing.

The following topics explain it further:

- Including Multiple References
- Using an Enveloped Signature
- Using an XPath Expression
- Using a Certificate Hint
- Signing with an HMAC Key

7.8.2.1 Including Multiple References

To include multiple references in a signature, simply add more `XSReference` objects to the `XSSignedInfo` object. Each `XSReference` object needs its own list of transforms.

7.8.2.2 Using an Enveloped Signature

To use an enveloped signature, add the enveloped signature transform to the reference. This means inserting the following code just before the code that adds the exclusive transform:

```
XSAlgorithmIdentifier transform1 =
  new XSAlgorithmIdentifier(doc, "Transform",
  XMLURI.alg_envelopedSignature);
ref.addTransform(transform1);
```

7.8.2.3 Using an XPath Expression

To use an XPath expression instead of an ID-based reference, pass in an empty string instead of "#foo1" for the URI parameter of `createReference`, then add an XPath transform to the `Reference` as the first transform.

```
String xpathExpr = "ancestor-or-self:importantInfo";
Element xpathElem = doc.createElementNS(XMLURI.ns_dsig,
 "dsig:XPath");
xpathElem.appendChild(doc.createTextNode(xpathExpr);
XSAlgorithmIdentifier transform2 =
  new XSAlgorithmIdentifier(doc, "Transform",
  XMLURI.alg_xpath);
transform2.addParameter(xpathElem);
ref.addTransform(transform2);
```

7.8.2.4 Using a Certificate Hint

If you do not want to include the entire certificate in the key info, but only a hint to the certificate, use the no-argument form of `XSKeyInfo.createX509Data()` and call one of the methods `X509Data.addIssuerSerial`, `addSubjectName`, or `addSubjectKeyID`.

7.8.2.5 Signing with an HMAC Key

TO sign with an HMAC key, instead of signing with an RSA or DSA private key, use the `XSSignature.sign(byte[]` secret, `String sigValueId)` method, and pass your HMAC key as the first argument.

Also use a different kind of KeyInfo, such as a KeyName, by calling XSKeyInfo.createKeyName.

7.9 Verifying Signatures with the Oracle XML Security API

Using Oracle XML Security APIs, you can locate what is signed, fetch the `keyinfo` of the signature, and then verify the signature.

Signature verification topics include:

- Checking What is Signed, Basic Procedure
- Setting Up Callbacks
- Writing a Custom Key Retriever
- Checking What is Signed
- Verifying the Signature

7.9.1 Checking What is Signed, Basic Procedure

You can verify a signature by first locating the `<dsig:Signature>` element in your document, using it to construct the `XSSignature` wrapper object, and then fetching the `KeyInfo` of the signature.

```
Element sigElem = …
XSSignature sig = new XSSignature(sigElem);
```

Next, fetch the `KeyInfo` of the signature and examine the key to determine if you trust the signer. There are different ways to deal with the `KeyInfo`:

- For very simple cases, you may already know the verification key in advance, and you do not need to look at the `KeyInfo` at all.
- In most cases, however, you should look at the `KeyInfo`. One way is to set up callbacks, so when you call `XSSignature.verify()` you call it with no verification key. Internally, the Oracle Security Developer Tools look at the `KeyInfo` to see if it invokes a callback to fetch the key.
- The other option is to proactively look into the `KeyInfo` and determine the key yourself.

7.9.2 Setting Up Callbacks

If the KeyInfo contains the signing certificate, set a certificate validator callback. If the KeyInfo contains a hint, write a `KeyRetriever` to fetch a certificate from a certificate store.

If the KeyInfo Contains the Signing Certificate

If you expect the KeyInfo to contain the signing certificate, and you do not already have this certificate, but you have set up the trust points, you just need to set a certificate validator callback.

```
// Create your certificate validator
CertificateValidator myValidator
  = new CertificateValidator() {
  public void validateCert(CertPath cp) {
      // Code to validate the certificate
  }
};
KeyRetriever.setCertificateValidator(myValidator);
```

The Oracle Security Developer Tools API retrieves the certificate from the `KeyInfo` and invokes your callback; if the callback returns `true`, it will verify with that certificate.

If the KeyInfo Contains a Hint

If you expect the `KeyInfo` to contain only a hint to the signing certificate, that is, the `subjectDN` or Issuer `Serial` or subject key identifier, write a `KeyRetriever` to fetch a certificate from a certificate store given this hint.

If your certificate store is a keystore, a PKCS12 wallet, or a PKCS8 file, you can use one of the built-in retrievers for these types. These retrievers iterate through all the certificates in the keystore or Oracle wallet and find the one which matches the given subjectDN/issuerSerial or SubjectKey.

> **Note:**
>
> You can also use this mechanism also if your `KeyInfo` contains the entire certificate; the key retriever will simply match the entire certificate.

```
// Load your keystore
KeyStore ks =
// Set up a callback against this KeyStore
KeyRetriever.addKeyRetriever(
  new KeyStoreKeyRetriever(ks, passwd));
```

7.9.3 Writing a Custom Key Retriever

If these built-in retrievers are not suitable, you can write a custom `KeyRetriever` by deriving from the `KeyRetriever` class.

For example you could do this when you expect the `KeyInfo` to contain a `subjectDN`, and you will look up an LDAP directory to find the certificate for that DN.

```
KeyRetriever myRetriever = new KeyRetriever() {
   X509Certificate retrieveCertificate (KeyInfoData keyInfo) {
       // write code to fetch the certificate from
       // the certificate store based on keyInfo
   }

   PublicKey retrieveCertificate (KeyInfoData keyInfo) {
       // write code to fetch the PublicKey from
       // the certificate store based on keyInfo
   }
};
KeyRetriever.addKeyRetriever(myRetriever);
```

If the signature used the symmetric key, and the `KeyInfo` has the keyname of that key, write a custom key retriever which can fetch the symmetric key based on this key name.

7.9.4 Checking What is Signed

You can check if a signature really signs what you were expecting it to sign. The Oracle Security Developer Tools API provides methods to return this information.

```
// XSSignature has be created as mentioned before
XSSignature sig = ...

// at first locate the element that are expecting
// to be signed
Element impElem = ...

// Now check if the signature really signs this
List signedObjects = XMLUtils.resolveReferences(sig);
if (signedObjects.size() != 1 ||
   signedObjects.get(0) != impElem {
   // something is wrong - impElem is not signed by
   // this signature
}
```

7.9.5 Verifying the Signature

You can verify a signature by using the `sig.verify()` method and know whether the signature format is correct. You can also debug the failed signatures.

The last step is to actually verify the signature. The call protocol depends on whether callbacks are set up.

7.9.5.1 Verifying if Callbacks are Set Up

If you set up callbacks, then make this call:

```
boolean result = sig.verify();
```

You need to check for both a false result and an exception:

- `sig.verify()` returns `false` if the signature format is correct, but one of the reference digests does not match, or if the signature does not verify.

- `sig.verify()` throws an exception if there is something wrong in the construction of the signature; for example, if the algorithm names are wrong or signature bytes are not of the right size.

7.9.5.2 Verifying if Callbacks are Not Set Up

If you did not set up callbacks, and you determined the key by yourself, you must call:

- `sig.verify(byte[])` for HMAC keys or
- `sig.verify(PublicKey)` for DSA/RSA keys.

7.9.5.3 Debugging Verification

If you cannot determine why a particular signature does not verify, and you need to debug it, set the JVM property `-Dxml.debug.verify=1`. This flag instructs the Oracle Security Developer Tools to print diagnostic output to the `stderr` for failed signatures.

7.10 Encrypting Data with the Oracle XML Security API

You can encrypt data with a shared symmetric key or a random symmetric key.

The following topics explain it further:

- Encrypting with a Shared Symmetric Key
- Encrypting with a Random Symmetric Key

7.10.1 Encrypting with a Shared Symmetric Key

You can create a new XEEncryptedData instance and specify the encryption method. Then, create a Keyinfo with a hint to the symmetric key. You can use the utility method `XEncUtils.encryptElement` to perform all these steps.

To encrypt and replace the following `<importantInfo>` element:

```
<myDoc>
  <importantInfo>
    ...
  </importantInfo>
</myDoc>
```

you will need to take the following steps:

```
// Assuming there is a shared symmetric key
SecretKey dataEncKey = ...

// Create a new XEEncryptedData instance
// use either obj_Element or obj_Content depending
// on whether you want to encrypt the whole element
// or content only
XEEncryptedData ed = XEEncryptedData
    .newInstance(doc, null, XMLURI.obj_Element);

// Specify the data encryption method
XEEncryptionMethod em =
    ed.createEncryptionMethod(XMLURI.alg_aes128_CBC);
ed.setEncryptionMethod(em);
```

```
// Create a Keyinfo with a hint to the symmetric key
XEKeyInfo ki= ed.createKeyInfo();
ki.addKeyInfoData(ki.createKeyName("MyKey"));
ed.setKeyInfo(ki);

// Locate the importantInfo element
Element impElem = ...

// Encrypt the importantInfo element and replace
// it with the EncryptedData element
XEEncrytedData.encryptAndReplace(impElem, dataEncKey,
   null, ed);
```

There is a utility method which performs all these steps:

```
XEncUtils.encryptElement(
  impElem,  // element to be encrypted
  false,    // true = contentOnly, false = entire element
  XMLURI.alg_aes128_CBC, // data encryption alg
  "MyKey" // hint to data key
);
```

7.10.2 Encrypting with a Random Symmetric Key

Usually you need to generate a random symmetric key and encrypt the data with that key, and then encrypt this random symmetric key with the receiver's public key. The XEncUtils.encryptElement method performs all these steps.

Here is how you would do that:

```
// Load up the encryption certificate of the reciever
X509Certificate encCert = ...

// Get the reciever's public key from the cert
PublicKey keyEncKey = encCert.getPublicKey();

// Then generate a random symmetric key
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(128);
SecretKey dataEncKey = keyGen.generateKey();

// Now create an EncryptedKey object
XEEncryptedKey = new XEEncryptedKey(doc);

// set up the key encryption algorithm
XEEncryptionMethod em =
  ek.createEncryptionMethod(XMLURI.alg_rsaOAEP_MGF1);
em.setDigestMethod(XMLURI.alg_sha1);
ek.setEncryptionMethod(em);

// encrypt the random symmetric key with public key
byte[] cipherValue = ek.encrypt(dataEncKey, keyEncKey);

// store this cipherValue into ek
XECipherData cd = ek.createCipherData();
cd.setCipherValue(cipherValue);
ek.setCipherData(cd);

// decide on how you would let the receiver know the
```

```
// the key encryption key. We are putting in the
// entire reciever's certificate
XEKeyInfo kki = ek.createKeyInfo();
kki.addKeyInfoData(kki.createX509Data(encCert);

// Now the encrypted key has been set up, let us
// do the data encryption as before
XEncUtils.encryptElement(
  impElem,  // element to be encrypted
  false,     // true = contentOnly, false = entire element
  XMLURI.alg_aes128_CBC, // data encryption alg
  null // No hint to data key
);

// Finally we need to put the EncryptedKey inside the
// KeyInfo of the EncryptedData
ed.addKeyInfoData(ek);
```

There is a utility method which performs all these steps:

```
XEncUtils.encryptElement (
  impElem,  // element to be encrypted
  false,     // true = contentOnly, false = entire element
  XMLURI.alg_aes128_CBC, // data encryption alg
  dataEncKey, // the random symmetric key that we generated
  XMLURI.alg_rsaOAEP_MGF1, // key encryption alg
  KeyEncKey, // public key that we got from cert
  "RecieverCert" // A hint to the certificate
);
```

Notice that this utility method puts KeyName in the EncryptedKey's KeyInfo; if you want to pass X509Data instead, pass null for keyEncKeyName and then add the X509Data yourself:

```
// use utility method to create EncrytedData
XEEncryptedData ed = XEncUtils...

// no extract EncryptedKey from it
XEEncryptedKey ek = (XEEncryptedKey)ed.getKeyInfo()
 .getEncryptedKeys().elementAt(0);

// Set the keyInfo of the ek
XEKeyInfo kki = ek.createKeyInfo();
kki.addKeyInfoData(kki.createX509Data(encCert);
```

7.11 Decrypting Data with the Oracle XML Security API

Oracle XML Security API has different methods for decrypting data depending upon whether you have used a shared symmetric key or a random symmetric key.

The topics in this section explain it further.

7.11.1 Decrypting with a Shared Symmetric Key

You can search for the encrypted data element and decrypt the data by using the `XEEncrytedData.decryptAndReplace` method.

If you have a shared symmetric key, do the following:

```
// search for the EncryptedData element
Element edElem = ...

// decrypt the data
SecretKey dataDecKey = ...
XEEncrytedData.decryptAndReplace(dataDecKey, edElem, true);
```

7.11.2 Decrypting with a Random Symmetric Key

With a random symmetric key, you can decrypt the data by using the
`XEEncUtils.decryptElement` method.

If you expect to use a random symmetric key:

```
// search for the EncryptedData element
Element edElem = ...

// decrypt the data
PrivateKey keyDecKey = ...
XEEncUtils.decryptElement(edElem, keyDecKey);
```

7.12 About Supporting Classes and Interfaces

Oracle XML Security API contains supporting classes and interfaces. The
`oracle.security.xmlsec.util.XMLURI` interface defines URI string constants for
algorithms, namespaces, and objects. The `oracle.security.xmlsec.util.XMLUtils`
class contains static utility methods for XML and XML-DSIG.

It contains these topics:

- About the oracle.security.xmlsec.util.XMLURI Interface
- About the oracle.security.xmlsec.util.XMLUtils class

7.12.1 About the oracle.security.xmlsec.util.XMLURI Interface

The `oracle.security.xmlsec.util.XMLURI` interface defines URI string constants for
algorithms, namespaces, and objects.

It uses the following naming convention:

- Algorithm URIs begin with "alg_".
- Namespace URIs begin with "ns_".
- Object type URIs begin with "obj_".

7.12.2 About the oracle.security.xmlsec.util.XMLUtils class

The `oracle.security.xmlsec.util.XMLUtils` class contains static utility methods for
XML and XML-DSIG.

Methods frequently used in applications include the `createDocBuilder()`,
`createDocument()`, `toBytesXML()`, and `toStringXML()` methods.

7.13 Common XML Security Questions

Learn frequently asked questions about Oracle XML Security.

What is the DER format? The PEM format? How are these formats used?

DER is an abbreviation for ASN.1 Distinguished Encoding Rules. DER is a binary format that is used to encode certificates and private keys. Oracle XML Security SDK uses DER as its native format, as do most commercial products that use certificates and private keys.

Many other formats used to encode certificates and private keys, including PEM, PKCS #7, and PKCS #12, are transformations of DER encoding. For example, PEM (Privacy Enhanced Mail) is a text format that is the Base 64 encoding of the DER binary format. The PEM format also specifies the use of text `BEGIN` and `END` lines that indicate the type of content that is being encoded.

I received a certificate in my email in a text format. It has several lines of text characters that don't seem to mean anything. How do I convert it into the format that Oracle XML Security uses?

If you received the certificate in your email, it is in PEM format. You need to convert the certificate from PEM (Privacy-Enhanced Mail) format to ASN.1 DER (Distinguished Encoding Rules) format.

How do I use a certificate that is exported from a browser?

If you have exported the certificate from a browser, it is most likely in PKCS #12 format (*.p12 or *.pfx). You must parse the PKCS #12 object into its component parts.

7.14 Best Practices for Oracle XML Security

You can refer to discussions on best practices for implementors and users of the XML Signature specification.

See the best practices at:

http://www.w3.org/TR/xmldsig-bestpractices/

7.15 The Oracle XML Security Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes, interfaces, and methods used in Oracle XML Security API.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

8

Oracle SAML

Oracle SAML allows Java developers to develop cross-domain single sign-on and federated access control solutions that conform to the SAML 1.0/1.1 and SAML 2.0 specifications.

This chapter contains the following topics:

- Oracle SAML Features and Benefits
- Oracle SAML 1.0/1.1
- Oracle SAML 2.0

8.1 Oracle SAML Features and Benefits

The Oracle SAML SDK provides a Java API with supporting tools, documentation, and sample programs to assist developers of SAML-compliant Java security services. Oracle SAML can be integrated into existing Java solutions, including applets, applications, EJBs, servlets, and JSPs.

Oracle SAML provides the following features:

- Support for the SAML 1.0/1.1 and 2.0 specifications
- Support for SAML-based single sign-on (SSO), Attribute, Metadata, Enhanced Client Proxy, and federated identity profiles

> ✎ **See Also:**
>
> For more information and links to these specifications and related documents, see References.

8.2 Oracle SAML 1.0/1.1

Oracle SAML 1.0/1.1 conforms to the SAML 1.0/1.1 specifications. You can set up your environment for Oracle SAML 1.0/1.1 toolkit, and use its classes and interfaces.

It contains the following topics:

- Oracle SAML 1.0/1.1 Packages
- Setting Up Your Oracle SAML 1.0/1.1 Environment
- Classes and Interfaces of Oracle SAML 1.x
- The Oracle SAML 1.0/1.1 Java API Reference

8.2.1 Oracle SAML 1.0/1.1 Packages

The Oracle SAML Java API contains the following packages for creating SAML 1.0/1.1-compliant Java applications: `oracle.security.xmlsec.saml` and `oracle.security.xmlsec.samlp`

`oracle.security.xmlsec.saml`

This package contains classes that support SAML assertions.

`oracle.security.xmlsec.samlp`

This package contains classes that support the SAML request and response protocol (SAMLP).

8.2.2 Setting Up Your Oracle SAML 1.0/1.1 Environment

You can setup Oracle SAML environment by installing Oracle Security Developer Tools and Java Development Kit (JDK), and setting the `CLASSPATH` variable to all of the required jar and class files.

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`.

In order to use Oracle SAML, your system must have the Java Development Kit (JDK) version 1.6 or higher.

Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files. Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- `osdt_xmlsec.jar`
- `osdt_saml.jar`
- The `org.jaxen_1.1.1.jar` file (Jaxen XPath engine, included with your Oracle XML Security distribution)

See Setting the CLASSPATH Environment Variable for configuration details.

8.2.3 Classes and Interfaces of Oracle SAML 1.x

Oracle SAML 1.0/1/1 contains multiple core classes to create SAML assertions, requests, and responses. It contains supporting interfaces which define URI string constants for algorithms, namespaces, and objects. It also contains a supporting class that is base class for all the SAML and SAML extension messages.

This section provides information and code samples for using the classes and interfaces of Oracle SAML 1.0/1.1. It contains these topics:

- Core Classes of Oracle SAML 1.x
- Supporting Classes and Interfaces

8.2.3.1 Core Classes of Oracle SAML 1.x

SAML assertions, requests, and responses are created with the Oracle SAML API.

This section provides a brief overview of the core SAML and SAMLP 1.0/1.1 classes with some brief code examples.

Topics include:

- Using the oracle.security.xmlsec.saml.SAMLInitializer Class
- Using the oracle.security.xmlsec.saml.Assertion Class
- Using the oracle.security.xmlsec.samlp.Request Class
- Using the oracle.security.xmlsec.samlp.Response Class

8.2.3.1.1 Using the oracle.security.xmlsec.saml.SAMLInitializer Class

This class initializes the Oracle SAML toolkit. By default Oracle SAML is automatically initialized for SAML v1.0. You can also initialize Oracle SAML for a specific version of the SAML specification. When the `initialize` method is called for a specific version, previously initialized versions will remain initialized.

This example shows how to initialize the SAML toolkit for SAML v1.0 and SAML v1.1.

```
// initializes for SAML v1.1
SAMLInitializer.initialize(1, 1);
// initializes for SAML v1.0, done by default
SAMLInitializer.initialize(1, 0);
```

8.2.3.1.2 Using the oracle.security.xmlsec.saml.Assertion Class

This class represents the `Assertion` element of the SAML Assertion schema.

This example shows how to create a new `Assertion` element and append it to an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;
Assertion assertion = new Assertion(doc);
doc.getDocumentElement().appendChild(assertion);
```

This example shows how to obtain `Assertion` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get a list of all Assertion elements in the document

NodeList assrtList =
    doc.getElementsByTagNameNS(SAMLURI.ns_saml, "Assertion");
if (assrtList.getLength() == 0)
    System.err.println("No Assertion elements found.");

// Convert each org.w3c.dom.Node object to a
// oracle.security.xmlsec.saml.Assertion object and process

for (int s = 0, n = assrtList.getLength(); s < n; ++s)
{
    Assertion assertion = new Assertion((Element)assrtList.item(s));
    // Process Assertion element
```

```
        ...
    }
```

8.2.3.1.3 Using the oracle.security.xmlsec.samlp.Request Class

This class represents the `Request` element of the SAML Protocol schema.

This example shows how to create a new `Request` element and append it to an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;
Request request = new Request(doc);
doc.getDocumentElement().appendChild(request);
```

This example shows how to obtain `Request` elements from an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get a list of all Request elements in the document

NodeList reqList =
    doc.getElementsByTagNameNS(SAMLURI.ns_samlp, "Request");
if (reqList.getLength() == 0)
    System.err.println("No Request elements found.");

// Convert each org.w3c.dom.Node object to a
// oracle.security.xmlsec.samlp.Request object and process

for (int s = 0, n = reqList.getLength(); s < n; ++s)
{
    Request request = new Request((Element)reqList.item(s));
    // Process Request element
    ...
}
```

8.2.3.1.4 Using the oracle.security.xmlsec.samlp.Response Class

This class represents the `Response` element of the SAML Protocol schema.

This example shows how to create a `Response` element and append it to an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;
Response response = new Response(doc);
doc.getDocumentElement().appendChild(response);
```

This example shows how to obtain Response elements from an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get a list of all Response elements in the document

NodeList respList =
    doc.getElementsByTagNameNS(SAMLURI.ns_samlp, "Response");
if (respList.getLength() == 0)
    System.err.println("No Response elements found.");

// Convert each org.w3c.dom.Node object to a
// oracle.security.xmlsec.samlp.Response object and process
```

```
for (int s = 0, n = respList.getLength(); s < n; ++s)
{
    Response response = new Response((Element)respList.item(s));
    // Process Response element
    ...
}
```

8.2.3.2 Supporting Classes and Interfaces

This section provides an overview of the supporting classes and interfaces of Oracle SAML 1.0/1.1:

- Using the oracle.security.xmlsec.saml.SAMLURI Interface
- Using the oracle.security.xmlsec.saml.SAMLMessage Class

8.2.3.2.1 Using the oracle.security.xmlsec.saml.SAMLURI Interface

This interface defines URI string constants for algorithms, namespaces, and objects. The following naming conventions are used:

- Action Namespace URIs defined in the SAML 1.0 specifications begin with `action_`.
- Authentication Method Namespace URIs defined in the SAML 1.0 specifications begin with `authentication_method_`.
- Confirmation Method Namespace URIs defined in the SAML 1.0 specifications begin with `confirmation_method_`.
- Namespace URIs begin with `ns_`.

8.2.3.2.2 Using the oracle.security.xmlsec.saml.SAMLMessage Class

This is the base class for all the SAML and SAML extension messages that may be signed and contain an XML-DSIG (digital signature) structure.

8.2.4 The Oracle SAML 1.0/1.1 Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes, interfaces, and methods available in Oracle SAML 1.0/1.1 API.

You can access this guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

8.3 Oracle SAML 2.0

Oracle SAML 2.0 conforms to the SAML 2.0 specifications. You can set up your environment for Oracle SAML 2.0 toolkit, and use its classes and interfaces.

It contains the following topics:

- Oracle SAML 2.0 Packages
- Setting Up Your Oracle SAML 2.0 Environment
- Classes and Interfaces of Oracle SAML 2.0

- The Oracle SAML 2.0 Java API Reference

8.3.1 Oracle SAML 2.0 Packages

Oracle SAML 2.0 API contains multiple packages with classes to support SAML assertions, SAML request and response protocol (SAMLP), and SAML authentication.

The Oracle SAML Java API contains the following packages for creating SAML 2.0-compliant Java applications:

`oracle.security.xmlsec.saml2.core`

This package contains classes that support SAML assertions.

`oracle.security.xmlsec.saml2.protocol`

This package contains classes that support the SAML request and response protocol (SAMLP).

oracle.security.xmlsec.saml2.ac

This package contains classes that support the SAML authentication context basic types.

oracle.security.xmlsec.saml2.ac.classes

This package contains classes that support various SAML authentication context classes.

oracle.security.xmlsec.saml2.metadata

This package contains classes that support the SAML metadata.

oracle.security.xmlsec.saml2.profiles.attributes

This package contains classes that support various SAML attribute profiles.

oracle.security.xmlsec.saml2.profiles.sso.ecp

This package contains classes that support the SAML ECP SSO profile.

8.3.2 Setting Up Your Oracle SAML 2.0 Environment

You can setup Oracle SAML environment by installing Oracle Security Developer Tools and Java Development Kit (JDK), and setting the CLASSPATH variable to all of the required jar and class files.

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`.

In order to use Oracle SAML, your system must have the Java Development Kit (JDK) version 1.6 or higher.

Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files. Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`

- `osdt_cert.jar`

- `osdt_xmlsec.jar`

- `osdt_saml.jar`

- The `org.jaxen_1.1.1.jar` file (Jaxen XPath engine, included with your Oracle XML Security distribution)

For example, your `CLASSPATH` might look like this:

```
%CLASSPATH%;%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_xmlsec.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_saml.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_saml2.jar;
%ORACLE_HOME%\modules\org.jaxen_1.1.1.jar;
```

See Setting the CLASSPATH Environment Variable for configuration details.

8.3.3 Classes and Interfaces of Oracle SAML 2.0

Oracle SAML 2.0 contains multiple core classes to create SAML assertions, requests, and responses. It contains supporting interfaces which define URI string constants for algorithms, namespaces, and objects.

This section provides information and code samples for using the classes and interfaces of Oracle SAML 2.0. It contains these sections:

- Core Classes of Oracle SAML 2.0
- Supporting Classes and Interfaces

8.3.3.1 Core Classes of Oracle SAML 2.0

Core classes of the Oracle SAML 2.0 API enable you to create assertions, requests, and responses.

This section provides an overview of the core SAML and SAMLP classes with some brief code examples. Topics are:

- Using the oracle.security.xmlsec.saml2.core.Assertion Class
- Using the oracle.security.xmlsec.saml2.protocol.AuthnRequest Class
- Using the oracle.security.xmlsec.saml2.protocol.StatusResponseType Class

8.3.3.1.1 Using the oracle.security.xmlsec.saml2.core.Assertion Class

This class represents the Assertion element of the SAML Assertion schema.

This example shows how to create a new Assertion element and append it to an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;
Assertion assertion = new Assertion(doc);
doc.getDocumentElement().appendChild(assertion);
```

This example shows how to obtain Assertion elements from an XML document.

```
// Get a list of all Assertion elements in the document
```

```
NodeList assrtList =
    doc.getElementsByTagNameNS(SAML2URI.ns_saml, "Assertion");
if (assrtList.getLength() == 0)
    System.err.println("No Assertion elements found.");

// Convert each org.w3c.dom.Node object to a
// oracle.security.xmlsec.saml2.core.Assertion object and process

for (int s = 0, n = assrtList.getLength(); s < n; ++s)
{
    Assertion assertion = new Assertion((Element)assrtList.item(s));
    // Process Assertion element
    ...
}
```

8.3.3.1.2 Using the oracle.security.xmlsec.saml2.protocol.AuthnRequest Class

This class represents the `AuthnRequest` element of the SAML Protocol schema.

This example shows how to create a new `AuthnRequest` element and append it to an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;
AuthnRequest request = new AuthnRequest(doc);
doc.getDocumentElement().appendChild(response);
```

This example shows how to obtain `AuthnRequest` elements from an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get a list of all AuthnRequest elements in the document

NodeList reqList =
    doc.getElementsByTagNameNS(SAML2URI.ns_samlp, "AuthnRequest");
if (reqList.getLength() == 0)
    System.err.println("No Request elements found.");

// Convert each org.w3c.dom.Node object to a
// oracle.security.xmlsec.saml2.protocol.AuthnRequest
// object and process

for (int s = 0, n = reqList.getLength(); s < n; ++s)
{
    AuthnRequest request = new AuthnRequest((Element)reqList.item(s));
    // Process Request element
    ...
}
```

8.3.3.1.3 Using the oracle.security.xmlsec.saml2.protocol.StatusResponseType Class

This class represents the `Response` element of the SAML Protocol schema.

The `samlp:StatusResponseType` element is a base type representing an extension point for the SAML 2.0 protocols. The various protocols defined in the SAML 2.0 specification use sub-types such as `samlp:Response` or `samlp:LogoutResponse`.

This example shows how to create a `Response` element and append it to an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;
Response response = new Response(doc);
doc.getDocumentElement().appendChild(response);
```

This example shows how to obtain `Response` elements from an existing XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get a list of all Response elements in the document

NodeList respList =
    doc.getElementsByTagNameNS(SAML2URI.ns_samlp, "Response");
if (respList.getLength() == 0)
    System.err.println("No Response elements found.");

// Convert each org.w3c.dom.Node object to a
// oracle.security.xmlsec.saml2.protocol.Response object and process

for (int s = 0, n = respList.getLength(); s < n; ++s)
{
    Response response = new Response((Element)respList.item(s));
    // Process Response element
    ...
}
```

8.3.3.2 Supporting Classes and Interfaces

This section provides an overview of the supporting classes and interfaces of Oracle SAML 2.0. It includes:

• Using the oracle.security.xmlsec.saml2.util.SAML2URI Interface

8.3.3.2.1 Using the oracle.security.xmlsec.saml2.util.SAML2URI Interface

This interface defines URI string constants for algorithms, namespaces, and objects. The interface uses these naming conventions:

• Action namespace URIs defined in the SAML 1.0/1.1/2.0 specifications begin with `action_` .

• Authentication method namespace URIs defined in the SAML 1.0/1.1/2.0 specifications begin with `authentication_method_`.

• Confirmation method namespace URIs defined in the SAML 1.0/1.1/2.0 specifications begin with `confirmation_method_` .

• Namespace URIs begin with `ns_`.

8.3.4 The Oracle SAML 2.0 Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes, interfaces, and methods available in Oracle SAML 2.0 API.

The Oracle SAML Java API reference (Javadoc) is available at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

9

Oracle Web Services Security

Oracle Web Services Security provides a complete implementation of the OASIS WS Security 1.1 standard. It provides mechanisms to sign and encrypt messages, and security tokens to ascertain the sender's identity.

This chapter describes how to install and use the SDK. This chapter contains these topics:

- Setting Up Your Oracle Web Services Security Environment
- Classes and Interfaces of Oracle Web Services Security
- The Oracle Web Services Security Java API Reference

9.1 Setting Up Your Oracle Web Services Security Environment

You can setup Oracle Web Services Security environment by installing Oracle Security Developer Tools and Java Development Kit (JDK), and setting the `CLASSPATH` variable to all the required jar files.

The Oracle Security Developer Tools are installed with Oracle Application Server in `ORACLE_HOME`.

To use Oracle Web Services Security, you must have Development Kit (JDK) version 1.6 or higher.

Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- `osdt_xmlsec.jar` - This is the Oracle XML Security jar.
- `osdt_saml.jar` - This is the Oracle SAML 1.0 and 1.1 jar.
- `osdt_saml2.jar` - This is the Oracle SAML 2.0 jar.
- `org.jaxen_1.1.1.jar`, which is included in `$ORACLE_HOME/modules/`.
- `osdt_wss.jar` - This is the main jar containing Oracle Web Services Security.
- `saaj-api.jar` - This is the standard SAAJ API and is included in JDK6; for previous JDKs, you can obtain it from your JavaEE container.
- `mail.jar, activation.jar` - You can obtain these jars from your JavaEE container.

> **See Also:**
>
> Setting the CLASSPATH Environment Variable.

9.2 Classes and Interfaces of Oracle Web Services Security

Oracle Web Services Security provides classes, interfaces, and methods to sign and encrypt messages, and security tokens to ascertain the sender's identity.

> **Note:**
>
> Review Oracle XML Security before proceeding.

This section describes classes and interfaces in the Oracle Web Services Security API. It contains these topics:

- Element Wrappers in Oracle Web Services Security
- The <wsse:Security> header
- Security Tokens (ST) in Oracle Web Services Security
- Security Token References (STR)
- Signing and Verifying
- Encrypting and Decrypting

9.2.1 Element Wrappers in Oracle Web Services Security

Oracle Web Services Security provides element wrappers to all XML elements.

> **See Also:**
>
> Understanding and Using Element Wrappers in the OSDT XML APIs

Table 9-1 lists the element wrappers provided by Oracle Web Services Security.

Table 9-1   Element Wrappers for Oracle Web Services Security

XML Tag Name	Java Class Name
<wsse:Security>	oracle.security.xmlsec.wss.WSSecurity
<wsse:BinarySecurityToken>	oracle.security.xmlsec.wss.WSSBinarySecurityToken or one of its derived classes depending on the valueType attribute: oracle.security.xmlsec.wss.x509.X509BinarySecurityToken oracle.security.xmlsec.wss.kerberos.KerberosBinarySecurityToken

Table 9-1   (Cont.) Element Wrappers for Oracle Web Services Security

XML Tag Name	Java Class Name
<wsse: SecurityTokenReference>	oracle.security.xmlsec.wss.WSSecurityTokenReference
<wsse: Embedded>	oracle.security.xmlsec.wss.WSSEmbedded
<wsse11:EncryptedHeader>	oracle.security.xmlsec.wss.WSSEncryptedHeader
<wsse11:SignatureConfirmation>	oracle.security.xmlsec.wss.WSSignatureConfirmation
<wsse:KeyIdentifier>	oracle.security.xmlsec.wss.WSSKeyIdentifier or one of its derived classes depending on the valueType attribute: oracle.security.xmlsec.wss.x509.X509KeyIdentifier oracle.security.xmlsec.wss.saml.SAMLAssertionKeyIdentifier oracle.security.xmlsec.wss.saml2.SAML2AssertionKeyIdentifier oracle.security.xmlsec.wss.kerberos.KerberosKeyIdentifier oracle.security.xmlsec.wss.WSSEncryptedKeyIdentifier
<wsse:Reference>	oracle.security.xmlsec.wss.WSSReference
<wsu:Created>	oracle.security.xmlsec.wss.WSUCreated
<wsu:Expires>	oracle.security.xmlsec.wss.WSUExpires
<wsu:Timestamp>	oracle.security.xmlsec.wss.WSUTimestamp
<wsse:UsernameToken>	oracle.security.xmlsec.wss.username.UsernameToken oracle.security.xmlsec.wss. oracle.security.xmlsec.wss. oracle.security.xmlsec.wss. oracle.security.xmlsec.wss. oracle.security.xmlsec.wss. oracle.security.xmlsec.wss.

As explained in Understanding and Using Element Wrappers in the OSDT XML APIs, the java classes are only throwaway wrappers, while the DOM elements are the source of truth. You can create these wrapper classes using the appropriate constructor, which takes in the DOM element; you can get the underlying DOM element using the getElement method.

9.2.2 The <wsse:Security> header

The WS Security specification defines a new SOAP Header called <wsse:Security>. All security information, such as Security Tokens, Timestamp, Signatures, EncryptedKeys, and ReferenceList, are stored inside this header.

- Security Tokens - Contain user name tokens, certificates, SAML assertion and so on (see next section)

- Timestamp - The current time stamp is often included in the security header, and it is usually included in a signature to prevent replay attacks.

- Signatures - Any signatures are stored inside the header. Even though the signature is in the Security header, what it signs is often outside the header - for example, a single signature can sign the SOAP Body, some SOAP attachments, a UserName token inside the Security header, and a Timestamp token in the Security header.

- EncryptedKeys - Any encrypted session keys are stored here.

- ReferenceList - Contains a list of all the EncryptedData sections.

9.2.2.1 Handling Outgoing Messages

For outgoing messages, you need to create a new `<wsse:Security>` header, add security tokens and then encrypt and/or sign parts of the document. Here is how to accomplish this task:

```
// Assuming we the outgoing message has already been constructed into
// a SOAPMessage object (part of SAAJ API)
SOAPMessage msg = ...

// Now create a new <wsse:Security> Header
// newInstance will internally use SOAPHeader.addHeaderElement
SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
WSSecurity ws = WSSecurity.newInstance(env);

// Add required prefixes to this SOAP header

// Now add some security tokens (refer to the next section on
// how to create security tokens)
UsernameToken ut = ...
ws.addUsernameToken(ut);

// Create some security token references to this token
// (refer to following sections)
ws.createSTR...

// Now sign or encrypt some data (refer to following sections)
// These should use the above STRs
ws.sign(...);
ws.encryptWithEncKey(...);
ws.encryptNoEncKey(...);
```

9.2.2.2 Handling Incoming Messages

For incoming messages, you need to look for a particular `<wsse:Security>` header, inspect its contents, and verify or decrypt parts of the document. To accomplish this task:

```
// Assuming we the incoming message has already been constructed into
// a SOAPMessage object (part of SAAJ API)
SOAPMessage msg = ...
```

9.2.3 Security Tokens (ST) in Oracle Web Services Security

A security token represents an artifact such as a certificate, a kerberos ticket, a user name with password, a Single sign-on token and so on. Oracle Web Services Security contains different types security tokens, such as Username token, X509 certificate, Kerberos ticket, and SAML Assertion, with multiple variations.

Usually a key is derived/extracted from this token, and this key is used to encrypt/ decrypt sign/verify parts of the message. However, the security token can also be used just as a data object.

Table 9-2   Security Tokens for Oracle Web Services Security

Type of Token (Java Class)	Variations	Keys
Username token oracle.security.xmlsec.wss.username. UsernameToken	• With no password • With a SHA1 digest of the password • With the actual password, or a different kind of digest/derived password.	Symmetric key obtained by running KeyDerivation on user's password
X509 certificate oracle.security.xmlsec.wss.x509. X509BinarySecurityToken	• Single v3 certificate • Chain of certificates in PKIPath format • Chain of certificates in PKCS7 format	• Public key inside certificate • Private key associated with certificate
Kerberos ticket oracle.security.xmlsec.wss.kerberos. KerberosBinarySecurityToken	• AP_REQ packet • GSS-wrapped AP_REQ packet	Either the session key present in the ticket, or a subkey.
SAML Assertion 1.1 oracle.security.xmlsec.wss.saml.SAMLAssertionToken **SAML Assertion 2.0** oracle.security.xmlsec.wss.saml2. SAML2AssertionToken	• holder_of_key • sender_vouchers • bearer	For holder_of_key the subject's key is used – this is, the key inside the <saml:SubjectConfirmation> which is inside the <saml:Assertion>. For sender_vouches, the key of the attesting entity is used. Keys are not extracted from bearer tokens.

9.2.3.1 Creating a WSS Username Token

First, create a `UsernameToken` and place it inside your `WSSecurity` header. The only mandatory field in the `UsernameToken` is the username:

```
// create a Username token
WSSecurity ws = ...
UsernameToken ut = new UsernameToken(doc);
ut.setUserName("Zoe");

// remember to put this inside your WSSecurity header.
// addUserNameToken puts it at the beginning, you can also
// use a regular DOM method appendChild or insertChild to put it in.
ws.addUsernameToken(ut);

// optionally add an wsu:Id, so you can refer to it
ut.setWsuId("MyUser");
```

Next, decide how to put the password into this token. There are several choices:

1. Add a clear text password. Consider using this technique only when the whole message is being sent over a secure channel like SSL.

2. Add a digest of the password or some other kind of derived password. A digest is not necessarily more secure than a clear text password, as it can also be replayed unless it is protected by a nonce and time.

3. Add a digest of the password using the digest mechanism given in the WS Security specification. This uses the nonce and the `createdDate`.

4. Do not add the password or its digest at all. Instead derive a key from the password and use that to sign the message, to demonstrate knowledge of the key.

```
// For options 1 and 2, use the setPassword method
ut.setPassword("IloveDogs");

// With this mechanism, the reciever should simply call
// UsernameToken.getPassword to check if the password is as expected.

// For option 3, use the setPasswordDigest method, but before doing
// thatfor that you have to  at first set a nonce and a created date.
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
byte nonce[] = new byte[20];
random.nextBytes(nonce); // compute a 20 byte random nonce
ut.setNonce(nonce);
ut.setCreatedDate(new Date()); // Set the date to now
ut.setPasswordDigest("IloveDogs"); // will compute the digest from
                                   // this clear text password using
                                   // nonce and createdDate

// For this mechanism, the reciever should use the following
byte nonce[] = ut.getNonce();
.. check against the used nonces, to make sure this is a new nonce
Date createdDate = ut.getCreated();
.. check that this createdDate is within an expected clock skew
boolean valid = ut.isValid(userName, passwd),
// above call will recompute the digest from the passwd
// and the nonce and created date, and check if this digest matches
// the digest in the username token


// For option 4, set the salt and iteration count
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
byte salt[] = new byte[15];
random.nextBytes(salt); // compute a 15 byte random salt

ut.setSalt(1, salt);
ut.setIteration(1000);
SecretKey key = ut.deriveKey("IloveDogs");
```

Now you can use this secret key to sign or encrypt data.

9.2.3.2 Creating an X509 Token

You can either use the `X509BinarySecurityToken` constructor followed by the `setToken` method, or use the equivalent helper method `WSSecurity.createBST_X509`:

```
WSSecurity ws = ...
X509Certificate cert = ...
X509BinarySecurityToken x509token = WSSecurity.createBST_X509(cert);

// remember to put this inside your WSSecurity header.
// addX509CertificateToken puts it at the beginning, you can also
// use a regular DOM method appendChild or insertChild to put it in.
ws.addX509CertificateToken(x509Token);

// optionally add an wsu:Id, so you can refer to it
x509Token.setWsuId("MyCert");
```

You can also create an `X509BinarySecurityToken` from a `CertPath` object if you want to include an entire chain of certificates.

For encryption data with this certificate, you need the public key which you can obtain by using `cert.getPublicKey()`. For signing, however, you need the private key, which you should maintain in a keystore.

9.2.3.3 Creating a Client-Side Kerberos Token

Kerberos tokens are used, as a rule, in conjunction with the Java GSS-API.

This example shows how to create a client-side token:

```
//Use JAAS Authentication with Kerberos Login Module

// Set up the config files and then call  login()
// to login using this module. This will cause the client to contact
// the Kerberos Authentication-Service and get a ticket to talk to the
// Kerberos Ticket-Granting-Service
LoginContext lc = new LoginContext(...);
lc.login();

//Use JAAS Authorization to set the subject into the thread context
Subject.doAs(lc.getSubject(), action)

// The rest of the code should be executed as a Privileged action
// Create a GSSContext to talk to a particular server.

GSSManager gssManager = GSSManager.getInstance();
GSSName serviceName = gssManager.createName(svcPrincipalName, null);
GSSContext gssContext = gssManager.createContext(serviceName, null,
    null, GSSCredential.DEFAULT_LIFETIME);

// Then call initSecContext. this will cause the client to contact
// the Ticket-Granting-Service to obtain a ticket for talking to that
// particular server. The token that is returned by the initSecContext
// is a GSS wrapped AP_REQ packet.
byte[] token = new byte[1];
token = gssContext.initSecContext(token, 0, token.length);

// Create a Kerberos BST using this AP_REQ packet
WSSecurity ws = ...
KerberosBinarySecurityToken kbst = ws.createBST_Kerberos(token,
      WSSURI.vt_GSSKerberosv5);
ws.addKerberosToken(kbst);

// Get the sessionKey that is present inside the AP_REQ packet,
// this is the session that is generated by the TGT and returned
// to the client in the initSecContext class
//
// This getSessionKey call simply calls Subject.getPrivateCredentials
// to get a list of tickets associated with the subject, and then
// iterates through them to find the one to be used for
// for that particular server
SecretKey sessionKey =
   KerberosUtils.getSessionKey(lc.getSubject(),svcPrincipalName);
```

Now you can use this secret key to sign or encrypt data.

Server Side

9.2.3.4 Creating a Server-side Kerberos Token

Server-side kerberos tokens require creation of the GSSContext and extraction of the session key.

This example shows how to create a server-side kerberos token:

```
// Use JAAS Authentication and Authorization as for the client
// Create GSSContext will null credentials </b><br>
SSManager manager = GSSManager.getInstance();
GSSContext gssContext = manager.createContext((GSSCredential)null);


// Locate the KerberosBinarySecurityToken in the incoming WSSecurity
// header. You can do this by doing a DOM search
WSSecurity = ...
KerberosBinarySecurityToken kbst = ...


// Now extract the AP_REQ from the BST and call acceptSecContext
byte ap_req[] = kbst.getValue();
gssContext.acceptSecContext(ap_req);

// The context is now extablished. (Note Mutual authentication would
// need one more round trip)

// Now extract the session key
// KerberosUtils.getSession is an overloaded method, and this
// particular one is meant to be used by server. Internally
// it decrypts the ap_req packet using the server's key (or the
// tgtSession key) and extracts the key from the decrypted ap_req
// packet
Subject srvrSubject = ...
SecretKey sessionKey =
    KerberosUtils.getSessionKey(srvrSubject, ap_req);
```

Now you can decrypt or verify using this key.

9.2.3.5 Creating a SAML Assertion Token

Refer to Oracle XML Security for information on how to create `Assertion` objects. From the `Assertion` object you can create a SAML assertion token by simply invoking the `SAMLAssertionToken(Assertion assertion)` constructor.

9.2.4 Security Token References (STR)

The WS Security specifications defines the concept of Security Token Reference (STR), which is a mechanism to refer to a security token. A `Signature` or `Encryption` uses this STR mechanism to identify the key that was used to sign or encrypt. STR supports mechanisms such as Direct Reference, Key Identifier, and Embedded.

STR typically supports the following mechanisms:

- Direct Reference: The STR uses a URI to refer to the ST.

- Key Identifier: The STR does not use a URI, but instead uses some other mechanism to identify the token, such as the `Issuer` serial for X509 tokens and the assertion ID for SAML tokens. The token may not be in the message at all.
- Embedded: The token is directly embedded in the `KeyInfo`.

9.2.4.1 Creating a direct reference STR

STRs are created using a uniform procedure; the mechanism to pass in the STR depends on the type of token.

To create the STR:

1. Create the token as mentioned earlier.
2. Call `.setWsuId()` to set an ID on that token
3. Create the STR with the ID from Step 2
4. Pass in that STR in the `WSSSignatureParams` or `WSEncryptionParams`

Subsequent sections demonstrate how to pass in the STR for various tokens.

9.2.4.2 Creating a Reference STR for a username token

This example shows how to create a reference STR for a username token:

```
WSSecurity ws = ...
WSSecurityTokenReference str =
    ws.createSTR_Username_ref("#MyUser");
```

9.2.4.3 Creating a Reference STR for a X509 Token

This example shows how to create a reference STR for an X509 token:

```
WSSecurity ws = ...
WSSecurityTokenReference str =
    ws.createSTR_X509_Ref("#MyCert");
```

9.2.4.4 Creating a Reference STR for Kerberos Token

This example shows how to create a reference STR for a kerberos token:

```
WSSecurity ws = ...
// use the appropriate value type
String valueType = WSSURI.vt_GSSKerberosv5;
WSSecurityTokenReference str =
    ws.createSTR_KerberosKeyRef ( "#MyToken");
```

9.2.4.5 Creating a Reference STR for a SAML Assertion token

This example shows how to create a reference STR for a SAML assertion token:

```
WSSecurity ws = ...
WSSecurityTokenReference str =
    ws.createSTR_SAML_Assertion_Ref20("MySAMLAssertion")
```

9.2.4.6 Creating a Reference STR for an EncryptedKey

This example shows how to create a reference STR for an encrypted key:

```
WSSecurity ws = ...
WSSecurityTokenReference str =
  ws.createSTR_EncKeyRef("MyEncKey")
```

9.2.4.7 Creating a Reference STR for a generic token

Instead of using the `createSTR` methods you can also create the reference directly with the appropriate `valueType` and `tokenType`:

```
WSSecurity ws = ...
String uri = "#MyToken";
WSSReference ref = new WSSReference(doc, uri);
ref.setValueType(valueType); // set an optional valueType
WSSecurityTokenReference str = new WSSecurityTokenReference(doc);
str.setTokenType(tokenType); // set an optional tokenType
str.appendChild(ref);
```

9.2.4.8 Creating a Key Identifier STR

A `KeyIdentifier` is another way to refer to a security token that uses some intrinsic property of the token; for example, an `assertionID` for a SAML `Token` or a `Subject Key Identifier` for an X509 token.

`KeyIdentifers` are often used when the token itself is not present in the document. For example, an incoming message can be encrypted with a `X509Cert`, but instead of having that `X509Cert` in the message, it can have only a hint to it, in the form of a `SubjectKeyIdentifier`.

9.2.4.9 Creating a KeyIdentifier STR for an X509 Token

There are three different ways to identify an X509 Token:

1. Issuer Serial: A combination of Issuer DN and Serial number of the certificate

2. Subject Key Identifier : The subject key Identifier of the certificate

3. Thumbprint SHA1: SHA1 of the certificate.

```
X509Certificate cert = ...
WSSecurity ws = ...
WSSecurityTokenReference str =
   ws.createSTR_X509_IssuerSerial(cert);
// alternatively use ws.createSTR_X509_SKI(cert)
// or ws. createSTR_X509_ThumbprintSHA1(cert)
```

9.2.4.10 Creating a KeyIdentifier STR for a Kerberos Token

Kerberos tokens can be identified by the SHA1 of the AP_REQ packet or of the GSS wrapped AP_REQ packet.

```
byte ap_req[] = ...
WSSecurity ws = ...
String valueType = WSSURI.vt_GSSKerberosv5;
```

```
WSSecurityTokenReference str =
    ws.createSTR_KerberosKeyIdSHA1(ap_req, valueType);
```

9.2.4.11 Creating a KeyIdentifier STR for a SAML Assertion Token

SAML assertions can be identified by the Assertion ID.

For local SAML 1.1 assertions use:

```
WSSecurity.createSTR_SAML_AssertionIdv11(byte assertionId[])
```

For remote SAML 1.1 assertions use:

```
createSTR_SAML_AssertionIdv11(
    byte assertionId[], AuthorityBinding authorityBinding)
```

For local SAML 2.0 assertions use:

```
createSTR_SAML_AssertionIdv20(byte assertionId[])
```

For remote SAML 2.0 assertions use a reference URI:

```
createSTR_SAML_Assertion_Ref20("MySAMLAssertion")
```

9.2.4.12 Creating a KeyIdentifier STR for an EncryptedKey

Remote encrypted keys can be identified by their SHA1 hash. Use this function to create the `KeyIdentifier`:

```
createSTR_EncKeySHA1(byte sha1[])
```

9.2.4.13 Adding an STRTransform

An STRTransform is a very useful transform that you add to your signatures. This transform causes a temporary replacement of the STRs wth the corresponding STs while calculating the signature.

For example, you might include an X509 SKI based STR in your reference. Without the `STRTransform` this will result in only the STR reference being included in the signature,that is, only the SKI value. But if you add an `STRTransform`, during the signing and verifiing process the STR will be replaced by the actual X509 Certificate, that is, the entire X509 certificate will be included in the message.

9.2.5 Signing and Verifying

You can sign and verify SOAP messages, and confirm signatures.

This section contains a discussion of signing and verifying data.

Topics include:

- Signing SOAP Messages
- Verifying SOAP Messages
- Confirming Signatures

9.2.5.1 Signing SOAP Messages

Take these steps to sign a SOAP message:

1. Decide how you want to identify the data to be signed – the most common mechanism is to use an ID, but instead of an ID you can also use an XPath expression

2. Decide on additional transforms – exclusive c14n and STR transforms are two common transforms that you might add.

3. Decide on the signing key – you can either do HMAC signing with a symmetric key or do RSA/DSA signatures.

4. Decide on how to indicate this signing key to the receiver – for this you usually need to create an STR as mentioned earlier.

9.2.5.1.1 Adding IDs to elements

IDs may be added to DOM elements.

Use the function:

```
WSSUtils.addWsuIdToElement(String id, Element element)
```

to add a `wsu:Id` to the element to be signed. You can use this mechanism to add an ID to regular DOM element, or SAAJ objects which also derive from DOM Elements.

You must declare the wsu namespace prefix. For example, you can declare it at the SOAP Envelope level like this

```
SOAPEnvelope env = ...
env.addNamespaceDeclaration("wsu" , WSSURI.ns_wsu);
```

To sign attachments, you must assign a `ContentId` to each attachment. For this you need to use the following method:

```
setContentId(String contentId)
```

of the SAAJ AttachmentPart object.

9.2.5.1.2 Creating the WSSignatureParams object

A `WSSignatureParams` object must be created with all the signing parameters.

Use the following constructor to create the initial `WSSignatureParams` object. If you want to use HMAC signing, pass in a value for `hmacKey`, and null for the `signingKey`; to use asymmetric signing, pass in a value for the `signingKey` and null for `hmacKey`.

```
WSSignatureParams(byte[] hmacKey, PrivateKey signingKey);
```

This constructor assumes `c14nMethod=excC14N`, `digestMethod=SHA1` and `signMethod=hmacSHA/rsaSHA1/dsaSHA1` (depending on the key). If you want different algorithms use the following setters to set them:

```
setDigestMethod(String digestMethod)
setSignMethod(String signMethod)
setC14nMethod(String method)
```

You also need to set the STR that you have created earlier into this object; use the `setKeyInfoData` for setting the STR.

```
setKeyInfoData(KeyInfoData keyInfoData)
```

When signing attachments, you need to set the `SOAPMessage` into this `WSSignatureParams` object so that it can resolve the cid references by locating corresponding attachments.

```
setSOAPMessage(SOAPMessage msg)
```

9.2.5.1.3 Specifying Transforms

There are two ways to specify transforms - a simpler but limited way, and an advanced and flexible way.

For the simple way, you need to set the following parameters in the WSSignatureParams:

```
setAttachmentContentOnly(boolean)
```

In the simple mode, all `cid` references automatically get the `AttachmentContentOnly` transform, but if you call `setAttachmentContentOnly(false)` then the `cid` references will get an `AttachmentComplete` transform

- `setUsingSTRTransform(boolean)`

If you set this to true, each reference will be checked whether it points to an STR, if it does an `STRTransform` will we added to that reference. Note the `STRTransform` is only added if the reference directly points to an STR, not if the reference points to a an ancestor of an STR.

- `setC14Nmethod(String)`

This parameter defaults to exclusive `c14n`, and specifies both the canonicalization method for each of the references and the canonicalization method for the `SignedInfo` section.

- `setUsingDecryptTransfom(boolean)`

Set this to `true` if you want a decrypt transform to be added.

9.2.5.1.4 Calling the WSSecurity.sign method

The final step is to call the following method in WSSecurity to perform the actual signing.

```
XSSignature sign (String[] uris, WSSignatureParams sigParams, XSAlgorithmIdentifier[]
[] trans)
```

This method creates the `<Signature>` element, computes digests of each reference and finally computes the signature.

uris is an array of IDs to be signed. A separate `<Reference>` will be created for each element of this array.

As described earlier there are two ways to specify the transforms – a simple way in which the transform must be `null`, and the transformation information is specified throught the various set methods mentioned above (in `WSSignatureParams`). Or a more advanced way where the transform parameter must explicitly specify all the transforms for each reference, that is, `trans.length` must be equal to `uris.length`.

9.2.5.2 Verifying SOAP Messages

When verifying a signature you first need to locate the signature elements in the `<wsse:Security>` header; for this you can use the method

```
WSSecurity ws = ...
List<XSSignature>sigs = ws.getSignatures();
```

This method searches the DOM tree to find all immediate children of `<wsse:Security>` that are `<dsig:Signature>` and then creates `XSSignature` wrapper objects for each of those elements and returns them. (Note the namespace prefixes do not have to use wsse and dsig).

If you already have the verification key in hand, you can call the following method - either pass in an `hmacKey` for HMAC signatures or a `signingKey` for asymmetric key signatures. The `SOAPMessage` is only need when attachments are signed.

```
XSSignature sig = sigs[0];

byte [] hmacKey = ...
PublicKey signingKey = ... ; // Need either hmacKey of signingKey

SOAPMessage msg = null; // needed only for attachments
boolean res = WSSecurity.verify(sig, byte[] hmacKey, signingKey, msg);
```

However, if you do not have the verification key, you need to set up the following callbacks for resolving STR Key Identifiers. Recall that STR Key Identifiers are usually references to tokens outside the document, so Oracle Security Developer Tools cannot locate these tokens unless you explicitly set up these callbacks.

Table 9-3   Callbacks to Resolve STR Key Identifiers

Token Type	Implementation Interface and Registration	Notes
Username Token	Interface: PasswordRetriever Registration: UsernameToken.addPassword Retriever	This callback resolves the UsernameToken Reference STRs. In the getPassword() callback, return the password corresponding to the user. This secret key will be derived from password, iteration count and salt. login() and logout() callbacks are not used

Table 9-3   (Cont.) Callbacks to Resolve STR Key Identifiers

Token Type	Implementation Interface and Registration	Notes
Username Token	Interface: KeyDerivator Registration: UsernameToken.addKeyDerivator	This callback also resolves the UsernameToken Reference STRs. Use it when you want to use your own key derivation algorithm. In the resolve() callback, derive the key and return it.
X509	Interface: X509KeyIdentifierResolver Registration: X509KeyIdentifier.addResolver	This callback resolves Thumbprint and SKI Key Identifier STRs. Implement the resolve() and getPrivateKey() callbacks to return the certificate and the private key respectively. Note: The private key is not required for verification, but it is required for decryption. If you have an array of certificates, use the X509KeyIdentifier.matches() method to match each cerificate against the passed-in X509 KeyIdentifier.
X509	Interface: X509IssuerSerialResolver Registration: X509IssuerSerial.addResolver	This callback resolves Issuer Serial Key Identifier STRs. Implement the resolve() and getPrivateKey() callbacks as in the previous case.

Table 9-3   (Cont.) Callbacks to Resolve STR Key Identifiers

Token Type	Implementation Interface and Registration	Notes
Kerberos	Interface: KerberosKeyIdentifierResolver	This callback resolves Kerberos STRs.
	Registration: KerberosKeyIdentifier.addResolver	Implement the resolve() and resolveKey() method to return the ap_req packet and the session key/subkey
		which corresponds to the SHA1 value present in the KeyIdentifier.
		If you have an array of ap_req packets, calculate the SHA1 of each one of them, and find the one
		whose SHA1 matches the value returned by KerberosKeyIdentifier.getValue().
		Return this ap_req packet in the resolve() method.
		For the resolveKey() method you need to take one more step and return they key present inside
		the ap_Req packet, for this youe can use the KerberosUtils.getSessionKey(Subject, byte[]) method,
		which decrypts the ap_req packet using the Subject's key and extracts the session key/sub-key from it.
SAML Assertion v1.1	Interface: SAMLAssertionKeyIdentifierResolver	This callback resolves SAML Assertion KeyIdentifier STRs.
	Registration: SAMLAssertionKeyIdentifier.addResolver	Implement the resolve(), getPublicKey() and getPrivateKey() methods to return the SAML assertion, SAMLX509Cert, and private key respectively. (Note: The private key is required only for decryption, not for verification.)
SAML Assertion v 2.0	Interface: SAML2AssertionKeyIdentifierResolver	See previous notes for SAML Assertion v1.1.
	Registration: SAML2AssertionKeyIdentifier.addResolver	

For tokens that use symmetric keys - UserName Token, Kerberos, and EncryptedKey - you need to set up a resolver, because the document does not have this symmetric

key, and Oracle Security Developer Tools cannot verify (or decrypt) unless you set the resolvers.

For tokens that use asymmetric keys - SAML Assertions and X509 Tokens - you do not need to set up a resolver if it uses a direct URI reference STR or an embedded token, because in these cases Oracle Security Developer Tools can locate the certificate on its own. However you still need to set up the `CertificateValidator` callback because Oracle Security Developer Tools will not blindly use a certificate in the message unless you have validated the certificate in your callback.

> ✎ **See Also:**
>
> Oracle XML Security

After you have set up all the resolvers and the `CertificateValidator`, use the following method:

```
SOAPMessage msg = null; // needed only for attachments
boolean searchTokens = true;
boolean res = WSSecurity.verify(sig, searchTokens, msg);
```

This method inspects the Signature's `KeyInfo` and either searches for the certificate, or calls the appropriate resolvers to get the signing key.

You can also use the `WSSecurity.verifyAll` method which searches for signatures and verifies them one by one.

9.2.5.3 Confirming Signatures

You use the `WSSignatureConfirmation` wrapper class to contruct and process signature confirmation elements.

9.2.5.3.1 Signature Confirmation Response Generation

For response generation use the following function in WSSecurity:

```
List<WSSignatureConfirmation> createSignatureConfirmations(Document doc);
```

This looks at all the `Signatures` present in the current `WSSecurity` element, and constructs corresponding `SignatureConfirmation` elements in a new document. These could be put in the response's WSSecuirty header.

9.2.5.3.2 Signature Confirmation Response Processing

For response processing, first use this function (at request time) to save all the Signature values.

```
String [] getSignatureValues()
```

At response processing time, you can then use this saved list to compare against the incoming `SignatureConfirmations` as follows:

```
boolean verifySignatureConfirmations(String sigValue[])
```

9.2.6 Encrypting and Decrypting

You can encrypt or decrypt SOAP messages with or without an `EncryptedKey`.

There are two primary encryption methods:

1. With `EncryptedKey`: Encrypt the elements with a random session key, then encrypt this session key into an `<EncryptedKey>` element and place that element in the `<wsse:Security>` header.

2. Without `EncryptedKey`: Encrypt the elements with known symmetric keys, which may be different for each element; construct a `<ReferenceList>` element with references to each of these encrypted data sections, and place the `<ReferenceList>` in the `<wsse:Security>` header.

> **✎ Note:**
>
> While encrypting regular DOM elements is standard practice, you can also encrypt SOAP headers, the SOAP body, and attachments. Special considerations apply for encrypting these objects as explained later.

9.2.6.1 Encrypting SOAP messages with EncryptedKey

You can encrypt SOAP messages by means of `EncryptedKey`.

First decide on a key to use to encrypt this random session key, then create an STR with the information that the receiver will use to locate this decryption key:

```
Key keyEncKey = ... ; WSSecurityTokenReference str = ...
```

create a `WSSEncryptionParams` with this information:

```
// Choose a data encryption algorithm - say AES 128
String dataEncAlg = XMLURI.alg_aes128_CBC;

// Either generate a random session key yourself, or set this to
// null to indicate that OSDT should generate it
SecretKey dataEncKey = null;

// Depending on the KeyEncryptionKey that you have chosen choose
// either an RSA key wrap or a symmetric key wrap
String keyEncAlg = XMLURI.alg_rsaOAEP_MGF1;

// Now put all this information into a WSSEncryptionParams
WSSEncryptionParams eParam = new WSSEncryptionParams(
 dataEncAlg, dataEncKey, keyEncAlg, keyEncKey, str);
```

regular DOM element, SOAP headers, the SOAP Body or AttachmentParts:

```
Element elem1 = ... // one object to be encrypted
Element elem2 = … // another object to be encrypted
ArrayList objectList[] = new ArrayList();
objectList.add(elem1);
objectList.add(elem2);
```

Create two more arrays to indicate whether each object is to be encrypted content only, and what IDs will be assigned to the resulting `EncryptedData` objects:

> **✎ Note:**
>
> SOAP bodies are always encrypted content only, regardless of what you pass in this flag. For attachments, "not content only" means content plus mime headers.

```
// both these elements are not content only
boolean[] contentOnlys = { false, false };

// After encryption the EncryptedData elements will get these ids
String encDataIds[] = { "id1", "id2" };
```

Finally, call the `encryptWithEncKey` method:

```
WSSecurity ws = ...
XEEncryptedKey encKey = ws.encryptWithEncKey(objectList, contentOnlys,
 encDataIds, eParam);
```

9.2.6.2 Encrypting SOAP messages without EncryptedKey

Use these steps if you do not wish to use an `EncryptedKey`:

Decide on a data encryption key; you can either use the same one for all the EncryptedData sections or a different one for each. Also create an STR with the information that the receiver will use to locate this decryption key, and put into a `WSSEncryptionParams` object:

```
SecretKey dataEncKey = ... ; // assuming 128 bit AES key
String dataEncAlg = XMLURI.alg_aes128_CBC;
WSSecurityTokenReference str = ...

// Now put all this information into a WSSEncryptionParams
WSSEncryptionParams eParam = new WSSEncryptionParams(
 dataEncAlg, dataEncKey, null, null, str);
```

Now create a list of elements to be encrypted as before, along with the associated `contentOnly` and `encDataIds` array:

```
Element elem1 = ... // one object to be encrypted
Element elem2 = … // another object to be encrypted
ArrayList objectList[] = new ArrayList();
objectList.add(elem1);
objectList.add(elem2);

// both these elements are not content only
boolean[] contentOnlys = { false, false };

// After encryption the EncryptedData elements will get these ids
String encDataIds[] = { "id1", "id2" };
```

Finally, call the `encryptWithNoEncKey` method:

```
WSSecurity ws = ...
XEEncryptedKey encKey = ws.encryptWithNoEncKey(objectList,
 contentOnlys, encDataIds, new WSEncryptionParams[]{eParam, eParam});
```

In this example we used the same `encryptionParams` for both elements.

9.2.6.3 Encrypting SOAP Headers into an EncryptedHeader

When you call the encrypt methods on the SOAP header block , with content only set to `false`, the entire SOAP header block is encrypted into an `EncryptedData` element; this element is placed inside an `EncryptedHeader` element, which replaces the original SOAP header block.

The `mustUnderstand` and `actor` attributes are copied over from the current `wsse:Security` header.

9.2.6.4 Decrypting SOAP messages with EncryptedKey

To decrypt SOAP messages with `EncryptedKey`, use:

```
WSSecurity.decrypt(XEEncryptedKey, PrivateKey, SOAPMessage)
```

which first decrypts the `EncryptedKey` with the given `PrivateKey` to obtain a symmetric key, then uses this symmetric key to decrypt all the references inside the `EncrytedKey`.

If you do not know the `PrivateKey`, call:

```
decrypt(XEEncryptedKey, SOAPMessage)
```

which looks into the `KeyInfo` of the `EncryptedKey` and calls the registered callbacks to obtain the private key.

If you already know the decrypted form of the `EncryptedKey` then use:

```
decrypt(XEEncryptedKey, SecretKey, SOAPMessage)
```

which uses the given symmetric key to decrypt all the references inside the `EncryptedKey`.

9.2.6.5 Decrypting SOAP messages without EncryptedKey

When you wish to decrypt all the elements (or attachments) mentioned in a top level `ReferenceList`, use:

```
decrypt(XEReferenceList, SecretKey, SOAPMessage)
```

which uses the given symmetric key to decrypt all the references inside the `ReferenceList`. This functions assumes that all the references are encrypted with the same key.

If you do not know the `SecretKey`, or if all the references are not encrypted with the same key, send in a `null` for the `SecretKey`; `decrypt` then looks into the `KeyInfo` of each of the `EncrytedData` and calls the registered callbacks to obtain the symmetric key.

9.3 Additional Resources for Web Services Security

OASIS Specifications, such as OASIS WSS SOAP Message Security Specification and OASIS WSS Username Token Profile Specification, provide more information about Web Services Security.

The following resources provide more information about Web Services Security:

- OASIS WSS SOAP Message Security Specification
- OASIS WSS Username Token Profile Specification
- OASIS WSS X.509 Certificate Token Profile Specification
- OASIS WSS SAML Assertion Token Profile Specification
- OASIS WSS SWA Token Profile Specification 1.1

> ✎ **See Also:**
>
> Links to these documents are available in References.

9.4 The Oracle Web Services Security Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes, interfaces, and methods available in the Oracle Web Services Security API.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

10

Oracle Liberty SDK

Oracle Liberty SDK allows Java developers to design and develop single sign-on (SSO) and federated identity management (FIM) solutions. It aims to unify, simplify, and extend all aspects of development and integration of systems conforming to the Liberty Alliance ID-FF 1.1 and 1.2 specifications.

The Liberty Alliance was founded with the goal of allowing individuals and businesses to engage in virtually any transaction without compromising the privacy and security of vital identity information. Specifications issued by the Liberty Alliance are based on an open identity federation framework, allowing partner companies to form business relationships based on a cross-organizational, federated network identity model.

This chapter contains these topics:

- Oracle Liberty SDK Features and Benefits
- Oracle Liberty 1.1
- Oracle Liberty 1.2

10.1 Oracle Liberty SDK Features and Benefits

Oracle Liberty SDK 1.1 and 1.2 enable simplified software development through the use of an intuitive and straightforward Java API. The toolkits provide tools, information, and examples to help you develop solutions that conform to the Liberty Alliance specifications. The toolkits can also be seamlessly integrated into any existing Java solution, including applets, applications, EJBs, servlets, JSPs, and so on.

The Oracle Liberty SDK is a pure java solution which provides the following features:

- Support for the Liberty Alliance ID-FF version 1.1 and 1.2 specifications
- Support for Liberty-based Single Sign-on and Federated Identity protocols
- Support for the SAML 1.0/1.1 specifications

> ✎ **See Also:**
>
> You can find the Liberty Alliance specifications at `http://www.projectliberty.org/resources/specifications.php`.

10.2 Oracle Liberty 1.1

Oracle Liberty 1.1 conforms to the Liberty Alliance ID-FF 1.1 specifications. It contains classes, interfaces, and methods to provide functionality such as authentication request/response, logout request/response, and federation termination.

This section explains how to set up your environment for and use Oracle Liberty 1.1, and describes the classes and interfaces of Oracle Liberty 1.1. It contains the following topics:

- Setting Up Your Oracle Liberty 1.1 Environment
- Overview of Oracle Liberty 1.1 Classes and Interfaces
- The Oracle Liberty 1.1 API Reference

10.2.1 Setting Up Your Oracle Liberty 1.1 Environment

You can setup Oracle Liberty 1.1 environment by installing Oracle Security Developer Tools and Java Development Kit (JDK), and setting the CLASSPATH variable to all of the required jar and class files.

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`.

10.2.1.1 Understanding System Requirements for Oracle Liberty 1.1

In order to use Oracle Liberty 1.1, your system must have the Java Development Kit (JDK) version 1.6 or higher.

Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files. Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- `osdt_xmlsec.jar`
- `osdt_saml.jar`
- The `org.jaxen_1.1.1.jar` file (Jaxen XPath engine, included with your Oracle XML Security distribution)
- the `osdt_lib_v11.jar` file

For example, your `CLASSPATH` might look like this:

```
%CLASSPATH%;%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_xmlsec.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_saml.jar;
%ORACLE_HOME%\modules\org.jaxen_1.1.1.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_lib_v11.jar;
```

> ✎ **See Also:**
>
> Setting the CLASSPATH Environment Variable

10.2.2 Overview of Oracle Liberty 1.1 Classes and Interfaces

Oracle Liberty SDK v. 1.1 contains core and supporting classes and interfaces to provide functionality such as authentication request/response, logout request/ response, and federation termination

This section introduces some useful classes and interfaces of Oracle Liberty SDK v. 1.1. It contains these topics:

- Using Core Classes and Interfaces
- Using Supporting Classes and Interfaces

10.2.2.1 Using Core Classes and Interfaces

The core classes and interfaces of the Oracle Liberty SDK v. 1.1 enable you to create authentication request and response elements, logout request and response elements, and register name identifiers.

This section contains the topics:

- Using the oracle.security.xmlsec.liberty.v11.AuthnRequest Class
- Using the oracle.security.xmlsec.liberty.v11.AuthnResponse Class
- Using the oracle.security.xmlsec.liberty.v11.FederationTerminationNotification Class
- Using the oracle.security.xmlsec.liberty.v11.LogoutRequest Class
- Using the oracle.security.xmlsec.liberty.v11.LogoutResponse Class
- Using the oracle.security.xmlsec.liberty.v11.RegisterNameIdentifierRequest Class
- Using the oracle.security.xmlsec.liberty.v11.RegisterNameIdentifierResponse Class

10.2.2.1.1 Using the oracle.security.xmlsec.liberty.v11.AuthnRequest Class

This class represents the `AuthnRequest` element of the Liberty protocol schema.

This example shows how to create a new `AuthnRequest` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
AuthnRequest authnRequest = new AuthnRequest(doc);
doc.getDocumentElement().appendChild(authnRequest);
```

This example shows how to obtain `AuthnRequest` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all AuthnRequest elements in the document.
NodeList arList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty, "AuthnRequest");
if (arList.getLength() == 0)
    System.err.println("No AuthnRequest elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v11.AuthnRequest object and process
```

```
for (int s = 0, n = arList.getLength(); s < n; ++s)
{
    AuthnRequest authnRequest =
        new AuthnRequest((Element)arList.item(s));

    // Process AuthnRequest element
    ...
}
```

10.2.2.1.2 Using the oracle.security.xmlsec.liberty.v11.AuthnResponse Class

This class represents the `AuthnResponse` element of the Liberty protocol schema.

This example shows how to create a new `AuthnResponse` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
AuthnResponse authnResponse = new AuthnResponse(doc);
doc.getDocumentElement().appendChild(authnResponse);
```

This example shows how to obtain `AuthnResponse` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all AuthnResponse elements in the document.
NodeList arList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty, "AuthnResponse");
if (arList.getLength() == 0)
    System.err.println("No AuthnResponse elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v11.AuthnResponse object and process
for (int s = 0, n = arList.getLength(); s < n; ++s)
{
    AuthnResponse authnResponse =
        new AuthnResponse((Element)arList.item(s));
    // Process AuthnResponse element
    ...
}
```

10.2.2.1.3 Using the oracle.security.xmlsec.liberty.v11.FederationTerminationNotification Class

This class represents the `FederationTerminationNotification` element of the Liberty protocol schema.

This example shows how to create a new federation termination notification element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
FederationTerminationNotification ftn =
    new FederationTerminationNotification(doc);
doc.getDocumentElement().appendChild(ftn);
```

This example shows how to obtain federation termination notification elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;
```

```
// Get list of all FederationTerminationNotification elements in the document
    NodeList ftnList = doc.getElementsByTagNameNS(LibertyURI.ns_liberty,
    "FederationTerminationNotification");
if (ftnList.getLength() == 0)
    System.err.println("No FederationTerminationNotification elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v11.FederationTerminationNotification
// object and process
for (int s = 0, n = ftnList.getLength(); s < n; ++s)
{
    FederationTerminationNotification ftn =
        new FederationTerminationNotification((Element)ftnList.item(s));

    // Process FederationTerminationNotification element
    ...
}
```

10.2.2.1.4 Using the oracle.security.xmlsec.liberty.v11.LogoutRequest Class

This class represents the `LogoutRequest` element of the Liberty protocol schema.

This example shows how to create a new `LogoutRequest` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
LogoutRequest lr = new LogoutRequest(doc);
doc.getDocumentElement().appendChild(lr);
```

This example shows how to obtain `LogoutRequest` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all LogoutRequest elements in the document.
NodeList lrList = doc.getElementsByTagNameNS(LibertyURI.ns_liberty,
    "LogoutRequest");
if (lrList.getLength() == 0)
    System.err.println("No LogoutRequest elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v11.LogoutRequest
// object and process
for (int s = 0, n = lrList.getLength(); s < n; ++s)
{
    LogoutRequest lr = new LogoutRequest((Element)lrList.item(s));

    // Process LogoutRequest element
    ...
}
```

10.2.2.1.5 Using the oracle.security.xmlsec.liberty.v11.LogoutResponse Class

This class represents the `LogoutResponse` element of the Liberty protocol schema.

This example shows how to create a new `LogoutResponse` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
LogoutResponse lr = new LogoutResponse(doc);
doc.getDocumentElement().appendChild(lr);
```

This example shows how to obtain `LogoutResponse` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all LogoutResponse elements in the document.
NodeList lrList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty, "LogoutResponse");
if (lrList.getLength() == 0)
    System.err.println("No LogoutResponse elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v11.LogoutResponse
// object and process
for (int s = 0, n = lrList.getLength(); s < n; ++s)
{
    LogoutResponse lr = new LogoutResponse((Element)lrList.item(s));

    // Process LogoutResponse element
    ...
}
```

10.2.2.1.6 Using the oracle.security.xmlsec.liberty.v11.RegisterNameIdentifierRequest Class

This class represents the `RegisterNameIdentifierRequest` element of the Liberty protocol schema.

This example shows how to create a new `RegisterNameIdentifierRequest` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
RegisterNameIdentifierRequest rnir =
    new RegisterNameIdentifierRequest(doc);
doc.getDocumentElement().appendChild(rnir);
```

This example shows how to obtain `RegisterNameIdentifierRequest` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all RegisterNameIdentifierRequest elements in the document
NodeList rnirList = doc.getElementsByTagNameNS(LibertyURI.ns_liberty,
    "RegisterNameIdentifierRequest");
if (rnirList.getLength() == 0)
    System.err.println("No RegisterNameIdentifierRequest elements found.");

// Convert each org.w3c.dom.Node object to an
//oracle.security.xmlsec.liberty.v11.RegisterNameIdentifierRequest
// object and process
for (int s = 0, n = rnirList.getLength(); s < n; ++s)
{
    RegisterNameIdentifierRequest rnir = new
        RegisterNameIdentifierRequest((Element)rnirList.item(s));

    // Process RegisterNameIdentifierRequest element
```

```
            ...
        }
```

10.2.2.1.7 Using the oracle.security.xmlsec.liberty.v11.RegisterNameIdentifierResponse Class

This class represents the `RegisterNameIdentifierResponse` element of the Liberty protocol schema.

This example shows how to create a new `RegisterNameIdentifierResponse` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
RegisterNameIdentifierResponse rnir = new RegisterNameIdentifierResponse(doc);
doc.getDocumentElement().appendChild(rnir);
```

This example shows how to obtain `RegisterNameIdentifierResponse` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all RegisterNameIdentifierResponse elements in the document
NodeList rnirList = doc.getElementsByTagNameNS(LibertyURI.ns_liberty,
    "RegisterNameIdentifierResponse");
if (rnirList.getLength() == 0)
    System.err.println("No RegisterNameIdentifierResponse elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v11.RegisterNameIdentifierResponse
// object and process
for (int s = 0, n = rnirList.getLength(); s < n; ++s)
{
    RegisterNameIdentifierResponse rnir = new
        RegisterNameIdentifierResponse((Element)rnirList.item(s));

    // Process RegisterNameIdentifierResponse element
    ...
}
```

10.2.2.2 Using Supporting Classes and Interfaces

This section describes supporting classes and interfaces of Oracle Liberty SDK v. 1.1.

The supporting classes and interfaces are:

- Using the oracle.security.xmlsec.liberty.v11.LibertyInitializer class
- The oracle.security.xmlsec.liberty.v11.LibertyURI interface
- Using the oracle.security.xmlsec.liberty.v11.ac.AuthenticationContextURI interface
- The oracle.security.xmlsec.util.ac.AuthenticationContextStatement class
- The oracle.security.xmlsec.saml.SAMLURI Interface
- The oracle.security.xmlsec.saml.SAMLMessage class

10.2.2.2.1 Using the oracle.security.xmlsec.liberty.v11.LibertyInitializer class

The `oracle.security.xmlsec.liberty.v11.LibertyInitializer` class handles load-time initialization and configuration of the Oracle Liberty SDK library. You must call this class's static `initialize()` method before making any calls to the Oracle Liberty SDK API.

10.2.2.2.2 The oracle.security.xmlsec.liberty.v11.LibertyURI interface

The `oracle.security.xmlsec.liberty.v11.LibertyURI` interface defines URI string constants for algorithms, namespaces and objects. The following naming convention is used:

- Algorithm URIs begin with "`alg_`".
- Namespace URIs begin with "`ns_`".
- Object type URIs begin with "`obj_`".
- Liberty profile namespace URIs begin with "`prof_`".

10.2.2.2.3 Using the oracle.security.xmlsec.liberty.v11.ac.AuthenticationContextURI interface

The `oracle.security.xmlsec.liberty.v11.ac.AuthenticationContextURI` interface defines URI string constants for algorithms, namespaces and objects. The following naming convention is used:

- Algorithm URIs begin with "`alg_`".
- Namespace URIs begin with "`ns_`".
- Object type URIs begin with "`obj_`".

10.2.2.2.4 The oracle.security.xmlsec.util.ac.AuthenticationContextStatement class

The `oracle.security.xmlsec.util.ac.AuthenticationContextStatement` class is an abstract class representing the top-level `AuthenticationContextStatement` element of the Liberty authentication context schema. Each concrete implementation of this class represents a respective class defined in the Liberty Authentication Context Specification.

10.2.2.2.5 The oracle.security.xmlsec.saml.SAMLURI Interface

The `oracle.security.xmlsec.saml.SAMLURI` interface defines URI string constants for algorithms, namespaces and objects. The following naming convention is used:

- Action namespace URIs defined in the SAML 1.0 specifications begin with "`action_`"
- Authentication method namespace URIs defined in the SAML 1.0 specifications begin with "`authentication_method_`".
- Confirmation method namespace URIs defined in the SAML 1.0 specifications begin with "`confirmation_method_`".
- Namespace URIs begin with "`ns_`".

10.2.2.2.6 The oracle.security.xmlsec.saml.SAMLMessage class

The `oracle.security.xmlsec.saml.SAMLMessage` class is the base class for all the SAML and SAML extension messages that may be signed and contain an XML-DSIG structure.

10.2.3 The Oracle Liberty 1.1 API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains classes, interfaces, and the methods available in Oracle Liberty SDK v1.1.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

10.3 Oracle Liberty 1.2

Oracle Liberty 1.2 conforms to the Liberty Alliance ID-FF 1.2 specifications. You can setup your environment and use the classes and interfaces in Oracle Liberty 1.2 to provide functionality as per the specifications.

This section describes the classes and interfaces of Oracle Liberty 1.2, and explains how to set up your environment and use Oracle Liberty 1.2.

It contains these sections:

- Setting Up Your Oracle Liberty 1.2 Environment
- Overview of Oracle Liberty 1.2 Classes and Interfaces
- The Oracle Liberty SDK 1.2 API Reference

10.3.1 Setting Up Your Oracle Liberty 1.2 Environment

You can setup Oracle Liberty 1.2 environment by installing Oracle Security Developer Tools and Java Development Kit (JDK), and setting the CLASSPATH variable to all of the required jar and class files.

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`.

In order to use Oracle Liberty 1.2, your system must have the Java Development Kit (JDK) version 1.6 or higher. Also, make sure that your `PATH` environment variable includes the Java bin directory.

Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files. Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- `osdt_xmlsec.jar`
- `osdt_saml.jar`

- The `org.jaxen_1.1.1.jar` file (Jaxen XPath engine, included with your Oracle XML Security distribution)
- `osdt_lib_v12.jar`

For example your classpath may look like this:

```
setenv CLASSPATH $CLASSPATH:$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_core.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cert.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_xmlsec.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_saml.jar:
$ORACLE_HOME/modules/org.jaxen_1.1.1.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_lib_v12.jar
```

10.3.2 Overview of Oracle Liberty 1.2 Classes and Interfaces

Oracle Liberty 1.2 contains multiple core and supporting classes and interfaces. Useful classes of Liberty 1.2 include assertion, request, response, authentication request/response, and others.

This section introduces classes and interfaces of Oracle Liberty SDK v. 1.2. It contains these topics:

- Core Classes and Interfaces
- Supporting Classes and Interfaces

10.3.2.1 Core Classes and Interfaces

This section describes core classes and interfaces of the Oracle Liberty SDK, v. 1.2.

The core classes are:

- Using the oracle.security.xmlsec.saml.Assertion class
- Using the oracle.security.xmlsec.samlp.Request class
- Using the oracle.security.xmlsec.samlp.Response class
- Using the oracle.security.xmlsec.liberty.v12.AuthnRequest class
- Using the oracle.security.xmlsec.liberty.v12.AuthnResponse class
- Using the oracle.security.xmlsec.liberty.v12.FederationTerminationNotification class
- Using the oracle.security.xmlsec.liberty.v12.LogoutRequest class
- Using the oracle.security.xmlsec.liberty.v12.LogoutResponse class
- Using the oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierRequest class
- Using the oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierResponse class

10.3.2.1.1 Using the oracle.security.xmlsec.saml.Assertion class

The `oracle.security.xmlsec.saml.Assertion` class represents the Assertion element of the SAML Assertion schema.

This example shows how to create a new assertion element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
Assertion assertion = new Assertion(doc);
doc.getDocumentElement().appendChild(assertion);
```

This example shows how to obtain assertion elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all Assertion elements in the document
NodeList assrtList =
    doc.getElementsByTagNameNS(SAMLURI.ns_saml, "Assertion");
if (assrtList.getLength() == 0)
    System.err.println("No Assertion elements found.");

// Convert each org.w3c.dom.Node object to
// an oracle.security.xmlsec.saml.Assertion
// object and process
for (int s = 0, n = assrtList.getLength(); s < n; ++s)
{
    Assertion assertion = new Assertion((Element)assrtList.item(s));

    // Process Assertion element
    ...
}
```

10.3.2.1.2 Using the oracle.security.xmlsec.samlp.Request class

The `oracle.security.xmlsec.samlp.Request` class represents the `Request` element of the SAML Protocol schema.

This example shows how to create a new `Request` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
Request request = new Request(doc);
doc.getDocumentElement().appendChild(request);
```

This example shows how to obtain `Request` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all Request elements in the document
NodeList reqList =
    doc.getElementsByTagNameNS(SAMLURI.ns_samlp, "Request");
if (reqList.getLength() == 0)
    System.err.println("No Request elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.samlp.Request
// object and process
for (int s = 0, n = reqList.getLength(); s < n; ++s)
{
    Request request = new Request((Element)reqList.item(s));

    // Process Request element
    ...
}
```

10.3.2.1.3 Using the oracle.security.xmlsec.samlp.Response class

The `oracle.security.xmlsec.samlp.Response` class represents the `Response` element of the SAML Protocol schema.

This example shows how to create a new element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
Response response = new Response(doc);
doc.getDocumentElement().appendChild(response);
```

This example shows how to obtain `Response` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all Response elements in the document
NodeList respList =
    doc.getElementsByTagNameNS(SAMLURI.ns_samlp, "Response");
if (respList.getLength() == 0)
    System.err.println("No Response elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.samlp.Response
// object and process
for (int s = 0, n = respList.getLength(); s < n; ++s)
{
    Response response = new Response((Element)respList.item(s));

    // Process Response element
    ...
}
```

10.3.2.1.4 Using the oracle.security.xmlsec.liberty.v12.AuthnRequest class

The `oracle.security.xmlsec.liberty.v12.AuthnRequest` class represents the `AuthnRequest` element of the Liberty protocol schema.

This example shows how to create a new authorization request element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
AuthnRequest authnRequest = new AuthnRequest(doc);
doc.getDocumentElement().appendChild(authnRequest);
```

This example shows how to obtain `AuthnRequest` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all AuthnRequest elements in the document
NodeList arList = doc.getElementsByTagNameNS(LibertyURI.ns_liberty, "AuthnRequest");

if (arList.getLength() == 0)
    System.err.println("No AuthnRequest elements found.");

// Convert each org.w3c.dom.Node object to
// an oracle.security.xmlsec.liberty.v12.AuthnRequest
// object and process
for (int s = 0, n = arList.getLength(); s < n; ++s)
{
    AuthnRequest authnRequest = new AuthnRequest((Element)arList.item(s));

    // Process AuthnRequest element
```

```
        ...
    }
```

10.3.2.1.5 Using the oracle.security.xmlsec.liberty.v12.AuthnResponse class

The `oracle.security.xmlsec.liberty.v12.AuthnResponse` class represents the `AuthnResponse` element of the Liberty protocol schema.

This example shows how to create a new authorization response element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
AuthnResponse authnResponse = new AuthnResponse(doc);
doc.getDocumentElement().appendChild(authnResponse);
```

This example shows how to obtain `AuthnResponse` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all AuthnResponse elements in the document.
NodeList arList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty, "AuthnResponse");
if (arList.getLength() == 0)
    System.err.println("No AuthnResponse elements found.");

// Convert each org.w3c.dom.Node object to
// an oracle.security.xmlsec.liberty.v12.AuthnResponse
// object and process
for (int s = 0, n = arList.getLength(); s < n; ++s)
{
    AuthnResponse authnResponse =
        new AuthnResponse((Element)arList.item(s));

    // Process AuthnResponse element
    ...
}
```

10.3.2.1.6 Using the oracle.security.xmlsec.liberty.v12.FederationTerminationNotification class

The `oracle.security.xmlsec.liberty.v12.FederationTerminationNotification` class represents the `FederationTerminationNotification` element of the Liberty protocol schema.

This example shows how to create a new federation termination notification element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
FederationTerminationNotification ftn =
    new FederationTerminationNotification(doc);
doc.getDocumentElement().appendChild(ftn);
```

This example shows how to obtain federation termination notification elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all FederationTerminationNotification elements in the document
```

```
NodeList ftnList = doc.getElementsByTagNameNS(LibertyURI.ns_liberty,
    "FederationTerminationNotification");
if (ftnList.getLength() == 0)
    System.err.println("No FederationTerminationNotification elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v12.FederationTerminationNotification
// object and process
for (int s = 0, n = ftnList.getLength(); s < n; ++s)
{
    FederationTerminationNotification ftn = new
        FederationTerminationNotification((Element)ftnList.item(s));

    // Process FederationTerminationNotification element
    ...
}
```

10.3.2.1.7 Using the oracle.security.xmlsec.liberty.v12.LogoutRequest class

The `oracle.security.xmlsec.liberty.v12.LogoutRequest` class represents the `LogoutRequest` element of the Liberty protocol schema.

This example shows how to create a new element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
LogoutRequest lr = new LogoutRequest(doc);
doc.getDocumentElement().appendChild(lr);
```

This example shows how to obtain logout request elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all LogoutRequest elements in the document
NodeList lrList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty, "LogoutRequest");
if (lrList.getLength() == 0)
    System.err.println("No LogoutRequest elements found.");

// Convert each org.w3c.dom.Node object to
// an oracle.security.xmlsec.liberty.v12.LogoutRequest
// object and process
for (int s = 0, n = lrList.getLength(); s < n; ++s)
{
    LogoutRequest lr = new LogoutRequest((Element)lrList.item(s));

    // Process LogoutRequest element
    ...
}
```

10.3.2.1.8 Using the oracle.security.xmlsec.liberty.v12.LogoutResponse class

The `oracle.security.xmlsec.liberty.v12.LogoutResponse` class represents the `LogoutResponse` element of the Liberty protocol schema.

This example shows how to create a new logout response element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
LogoutResponse lr = new LogoutResponse(doc);
```

```
doc.getDocumentElement().appendChild(lr);
```

This example shows how to obtain logout response elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all LogoutResponse elements in the document
NodeList lrList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty, "LogoutResponse");
if (lrList.getLength() == 0)
    System.err.println("No LogoutResponse elements found.");

// Convert each org.w3c.dom.Node object to
// an oracle.security.xmlsec.liberty.v12.LogoutResponse
// object and process
for (int s = 0, n = lrList.getLength(); s < n; ++s)
{
    LogoutResponse lr = new LogoutResponse((Element)lrList.item(s));

    // Process LogoutResponse element
    ...
}
```

10.3.2.1.9 Using the oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierRequest class

The `oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierRequest` class represents the `RegisterNameIdentifierRequest` element of the Liberty protocol schema.

This example shows how to create a new `RegisterNameIdentifierRequest` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
RegisterNameIdentifierRequest rnir = new RegisterNameIdentifierRequest(doc);
doc.getDocumentElement().appendChild(rnir);
```

This example shows how to obtain `RegisterNameIdentifierRequest` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all
// RegisterNameIdentifierRequest elements
// in the document
NodeList rnirList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty,
    "RegisterNameIdentifierRequest");
if (rnirList.getLength() == 0)
    System.err.println("No RegisterNameIdentifierRequest elements found.");

// Convert each org.w3c.dom.Node object to a
// oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierRequest
// object and process
for (int s = 0, n = rnirList.getLength(); s < n; ++s)
{
    RegisterNameIdentifierRequest rnir =
        new RegisterNameIdentifierRequest((Element)rnirList.item(s));
```

```
            // Process RegisterNameIdentifierRequest element
            ...
    }
```

10.3.2.1.10 Using the oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierResponse class

The `oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierResponse` class represents the `RegisterNameIdentifierResponse` element of the Liberty protocol schema.

This example shows how to create a new `RegisterNameIdentifierResponse` element and append it to a document.

```
Document doc = Instance of org.w3c.dom.Document;
RegisterNameIdentifierResponse rnir =
    new RegisterNameIdentifierResponse(doc);
doc.getDocumentElement().appendChild(rnir);
```

This example shows how to obtain `RegisterNameIdentifierResponse` elements from an XML document.

```
Document doc = Instance of org.w3c.dom.Document;

// Get list of all RegisterNameIdentifierResponse elements in the document
NodeList rnirList =
    doc.getElementsByTagNameNS(LibertyURI.ns_liberty,
        "RegisterNameIdentifierResponse");

if (rnirList.getLength() == 0)
    System.err.println("No RegisterNameIdentifierResponse elements found.");

// Convert each org.w3c.dom.Node object to an
// oracle.security.xmlsec.liberty.v12.RegisterNameIdentifierResponse
// object and process
for (int s = 0, n = rnirList.getLength(); s < n; ++s)
{
    RegisterNameIdentifierResponse rnir = new
        RegisterNameIdentifierResponse((Element)rnirList.item(s));

    // Process RegisterNameIdentifierResponse element
    ...
}
```

10.3.2.2 Supporting Classes and Interfaces

This section describes supporting classes and interfaces of Oracle Liberty SDK v. 1.2:

- The `oracle.security.xmlsec.liberty.v12.LibertyInitializer` class
- The `oracle.security.xmlsec.liberty.v12.LibertyURI` interface
- The `oracle.security.xmlsec.util.ac.AuthenticationContextStatement` class
- The `oracle.security.xmlsec.saml.SAMLInitializer` class
- The `oracle.security.xmlsec.saml.SAMLURI` interface

10.3.2.2.1 The oracle.security.xmlsec.liberty.v12.LibertyInitializer class

This class handles load-time initialization and configuration of the Oracle Liberty SDK 1.2 library. You must call this class's static `initialize()` method before making any calls to the Oracle Liberty SDK 1.2 API.

10.3.2.2.2 The oracle.security.xmlsec.liberty.v12.LibertyURI interface

This interface defines URI string constants for algorithms, namespaces, and objects.

10.3.2.2.3 The oracle.security.xmlsec.util.ac.AuthenticationContextStatement class

This is an abstract class representing the top-level `AuthenticationContextStatement` element of the Liberty authentication context schema. Each concrete implementation of this class represents the respective class defined in the Liberty Authentication Context Specification.

10.3.2.2.4 The oracle.security.xmlsec.saml.SAMLInitializer class

This class handles load-time initialization and configuration of the Oracle SAML library. You should call this class's static `initialize(int major, int minor)` method, for version 1.1, before making any calls to the Oracle SAML Toolkit API for SAML 1.1.

10.3.2.2.5 The oracle.security.xmlsec.saml.SAMLURI Interface

The `oracle.security.xmlsec.saml.SAMLURI` interface defines URI string constants for algorithms, namespaces, and objects. The following naming convention is used:

- Action Namespace URIs defined in the SAML 1.1 specifications begin with "`action_`"

- Authentication Method Namespace URIs defined in the SAML 1.1 specifications begin with "`authentication_method_`"

- Confirmation Method Namespace URIs defined in the SAML 1.1 specifications begin with "`confirmation_method_`"

- Namespace URIs begin with "`ns_`"

10.3.2.2.6 The oracle.security.xmlsec.saml.SAMLMessage Class

`oracle.security.xmlsec.saml.SAMLMessage` is the base class for all the SAML and SAML extension messages that may be signed and contain an XML-DSIG structure.

10.3.3 The Oracle Liberty SDK 1.2 API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes, interfaces, and methods available in Oracle Liberty SDK v1.2 API.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

11

Oracle XKMS

XKMS (XML Key Management Specification) is a W3C specification for public key management. Oracle XKMS API conforms to this specification. It provides a convenient way to handle public key infrastructures by enabling developers to write XML transactions for digital signature processing.

This chapter contains these topics:

- Understanding Oracle XKMS Features and Benefits
- Setting Up Your Oracle XKMS Environment
- Core Classes and Interfaces
- The Oracle XKMS Java API Reference

11.1 Understanding Oracle XKMS Features and Benefits

Oracle XKMS is a pure Java solution which consists of a toolkit for locating keys and verifying user identities across businesses and applications. It supports the secure, trusted messaging required for web services, and provides a way to sidestep some of the costs and complexity associated with PKI.

Oracle XKMS provides the following features:

- Simplified access to PKI functionality - by implementing the W3C XKMS Standard, Oracle XKMS combines the simplicity of XML with the robustness of PKI. With this toolkit, developers can easily deploy robust application functionality by deploying secure, lightweight client software.
- Supports complete key/certificate life cycle - Oracle XKMS helps enterprise applications locate, retrieve, and validate signature and encryption keys using lightweight Web Services infrastructure.
- Secures XKMS messages using XML Signatures - requests and responses can be digitally signed using Oracle XML toolkit.
- 100% Java with no native methods
- Works with JAXP 1.1 compliant XML parsers

The Oracle XKMS library contains the following packages:

Table 11-1   Packages in the Oracle XKMS Library

Package	Description
`oracle.security.xmlsec.xkms`	Contains the main XKMS message elements
`oracle.security.xmlsec.xkms.xkiss`	Contains the classes for the Key Information Service Specification
`oracle.security.xmlsec.xkms.xkrss`	Contains the classes for the Key Registration Service Specification

Table 11-1   (Cont.) Packages in the Oracle XKMS Library

Package	Description
`oracle.security.xmlsec.xkms.util`	Contains constants and utility classes

11.2 Setting Up Your Oracle XKMS Environment

You can setup Oracle XKMS environment by installing Oracle Security Developer Tools and Java Development Kit (JDK), and setting the CLASSPATH variable to all of the required jar and class files.

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`.

In order to use Oracle XKMS, your system must have the following components installed:

- The Java Development Kit (JDK) version 1.6 or higher
- the Oracle XML Security toolkit

Your `CLASSPATH` environment variable must contain the full path and file names to the required jar and class files. Make sure that the following files are included in your `CLASSPATH`:

- `osdt_core.jar`
- `osdt_cert.jar`
- `osdt_xmlsec.jar`
- `org.jaxen_1.1.1.jar`, which is located in the `$ORACLE_HOME/modules/` directory of the security tools distribution. Oracle XML Security relies on the Jaxen XPath engine for XPath processing.

For example, your `CLASSPATH` might look like this:

```
C:%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_core.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_cert.jar;
%ORACLE_HOME%\modules\oracle.osdt_11.1.1\osdt_xmlsec.jar;
%ORACLE_HOME%\modules\org.jaxen_1.1.1.jar;
```

> **See Also:**
>
> Setting the CLASSPATH Environment Variable

11.3 Core Classes and Interfaces

Core classes of Oracle XKMS enable you to locate, validate, and recover message requests and results. You can refer the code samples to use the core classes and interfaces of Oracle XKMS.

The core classes are:

- oracle.security.xmlsec.xkms.xkiss.LocateRequest
- Using the oracle.security.xmlsec.xkms.xkiss.LocateResult Class
- Using the oracle.security.xmlsec.xkms.xkiss.ValidateRequest Class
- Using the oracle.security.xmlsec.xkms.xkiss.ValidateResult Class
- Using the oracle.security.xmlsec.xkms.xkrss.RecoverRequest Class
- Using the oracle.security.xmlsec.xkms.xkrss.RecoverResult Class

11.3.1 oracle.security.xmlsec.xkms.xkiss.LocateRequest

The `oracle.security.xmlsec.xkms.xkiss.LocateRequest` class represents the XKMS `LocateRequest` element. You can refer examples and create an instance of `LocateRequest`, and attach `RespondWith` attribute to `LocateRequest`.

```
// Parse the XML document containing the dsig:Signature.
Document sigDoc = //Instance of org.w3c.dom.Document;

//Create Query Key Binding
QueryKeyBinding queryKeyBinding = new QueryKeyBinding(sigDoc);
queryKeyBinding.setTimeInstant(new Date());

// Create the xkms:LocateRequest.
LocateRequest loc = new LocateRequest(sigDoc, queryKeyBinding);
```

Client requests of type `LocateRequest` must include an `xkms:RespondWith` attribute.

This example shows how `RespondWith` can be added to a `LocateRequest`:

```
//Add xkms:RespondWith as X.509 Certificate.
loc.addRespondWith(XKMSURI.respondWith_X509Cert);
```

11.3.2 Using the oracle.security.xmlsec.xkms.xkiss.LocateResult Class

`oracle.security.xmlsec.xkms.xkiss.LocateResult` class represents the `xkms:LocateResult` element. You can create an instance of `LocateResult` element. If the `LocateRequest` contains a `RespondWith` attribute of `X509Certificate`, you can add an X509 Certificate to the `LocateResult` element.

Example:

```
//Parse the XML document containin the dsig:Signature
Document sigDoc = //Instance of org.w3c.doc.Document;

// Create the xkms:LocateResult
LocateResult locRes = new LocateResult(sigDoc);

//Set ResultMajor to Success.
locRes.setResultCode(XKMSURI.result_major_success, null);
```

If the `LocateRequest` contained a `RespondWith` attribute of `X509Certificate`, use the following code to add an X509 Certificate to the `LocateResult`:

```
//Creating a signature and adding X509 certificate to the KeyInfo element.
X509Certificate userCert = // Instance of java.security.cert.X509Certificate
XSSignature Sig = XSSignature.newInstance(sigDoc, "MySignature");
```

```
XSKeyInfo xsInfo = sig.getKeyInfo();
X509Data xData = xsInfo.createX509Data(userCert);

//Add X509Data to the KeyInfo
xsInfo.addKeyInfoData(xData);

//Set Key Binding and add KeyInfo the the KeyBinding
UnverifiedKeyBinding keyBinding = new UnverifiedKeyBinding(sigDoc);
keyBinding.setKeyInfo(xsInfo);

//Add Key Binding to LocateResult
locRes.addKeyBinding(keyBinding);
```

11.3.3 Using the oracle.security.xmlsec.xkms.xkiss.ValidateRequest Class

The `oracle.security.xmlsec.xkms.xkiss.ValidateRequest` class represents the XKMS `xkms:ValidateRequest` element. With this class you can create an instance of `xkms:ValidateRequest` element.

This example shows how to create an instance of `xkms:ValidateRequest`:

```
// Parse the XML document containing the dsig:Signature.
Document sigDoc = //Instance of org.w3c.dom.Document;

//Create Query Key Binding
QueryKeyBinding queryKeyBinding = new QueryKeyBinding(sigDoc);
queryKeyBinding.setTimeInstant(new Date());

// Create the xkms:ValidateRequest.
ValidateRequest validateReq = new ValidateRequest(sigDoc, queryKeyBinding);
```

Requests of type `ValidateRequest` must include an `xkms:RespondWith` attribute. This example shows how to add `RespondWith` to a `ValidateRequest`:

```
//Add xkms:RespondWith as X.509 Certificate.
validateReq.addRespondWith(XKMSURI.respondWith_X509Cert);
```

11.3.4 Using the oracle.security.xmlsec.xkms.xkiss.ValidateResult Class

With the `oracle.security.xmlsec.xkms.xkiss.ValidateResult` class, you can create an instance of `ValidateResult`. You can also set a status in response to a `ValidateRequest`.

This example shows how to create an instance of `ValidateResult`:

```
//Parse the XML document containin the dsig:Signature
Document sigDoc = //Instance of org.w3c.doc.Document;

// Create the xkms:ValidateResult
ValidateResult valRes = new ValidateResult(sigDoc);

//Set ResultMajor to Success.
valRes.setResultCode(XKMSURI.result_major_success, null);
```

Use the following code to set a status in response to a `ValidateRequest`:

```
//Create a status element and add reasons.
Status responseStatus = new Status(sigDoc);
responseStatus.addValidReason(XKMSURI.reasonCode_IssuerTrust);
responseStatus.addValidReason(XKMSURI.reasonCode_RevocationStatus);
responseStatus.addValidReason(XKMSURI.reasonCode_ValidityInterval);
responseStatus.addValidReason(XKMSURI.reasonCode_Signature);

//Create a xkms:KeyBinding to add status and X509Data
XSKeyInfo xsInfo =
    // Instance of oracle.security.xmlsec.dsig.XSKeyInfo,
    // which contains X509Data
KeyBinding keyBinding = new KeyBinding(sigDoc);
keyBinding.setStatus(responseStatus);
keyBinding.setKeyInfo(xsInfo);

// Add the key binding to the ValidateResult.
valRes.addKeyBinding(keyBinding);
```

11.3.5 Using the oracle.security.xmlsec.xkms.xkrss.RecoverRequest Class

The `oracle.security.xmlsec.xkms.xkrss.RecoverRequest` class represents the XKMS `RecoverRequest` element. With this class, you can create an instance of `RecoverRequest`. You can also add the `Authentication` and `RecoverKeyBinding` elements to `RecoverRequest`.

This example shows how to create an instance of `RecoverRequest`:

```
// Parse the XML document containing the dsig:Signature.
Document sigDoc = //Instance of org.w3c.dom.Document;

// Create the xkms:RecoverRequest
RecoverRequest recReq = new RecoverRequest(sigDoc);

//Set RespondWith to PrivateKey, so that the RecoverResult
contains the private key.
recReq.addRespondWith(XKMSURI.respondWith_PrivateKey);
```

A `RecoverRequest` must include the `Authentication` and `RecoverKeyBinding` elements. These can be added with the following code:

```
//Create an instance of XSSignature.
XSSignature sig =
    //Instance of oracle.security.xmlsec.dsig.XSSignature

//Create an instance of Authentication element.
Authentication auth = new Authentication(sigDoc);

//Set key binding authentication.
auth.setKeyBindingAuthentication(sig);

//Set Authentication for the RecoverRequest.
recReq.setAuthentication(auth);

//Add RecoverKeyBinding to RecoverRequest.
RecoverKeyBinding recKeyBind = new RecoverKeyBinding(sigDoc);

//Add Key Info on the key to be recovered.
XSKeyInfo xsInfo =
```

```
      //Instance of oracle.security.xmlsec.dsig.XSKeyInfo
recKeyBind.setKeyInfo(xsInfo);

//Adding status, as known to the key holder, to the KeyBinding
Status keyStatus = new Status(sigDoc);
keyStatus.setStatusValue(XKMSURI.kbs_Indeterminate);
recKeyBind.setStatus(keyStatus);

//Adding RecoverKeyBinding to RecoverRequest.
recReq.setKeyBinding(recKeyBind);
```

11.3.6 Using the oracle.security.xmlsec.xkms.xkrss.RecoverResult Class

The `oracle.security.xmlsec.xkms.xkrss.RecoverResult` class represents the `xkms:RecoverResult` element. With this class, you can create an instance of `RecoverResult`. You can set `KeyBinding` for `RecoverResult`. You can also set the recovered `PrivateKey` into the `RecoverResult`.

This example shows how to create an instance of `RecoverResult`:

```
// Parse the XML document containing the dsig:Signature.
Document sigDoc = //Instance of org.w3c.dom.Document;

// Create the xkms:RecoverResult
RecoverResult recResult = new RecoverResult(sigDoc);

//Set ResultMajor to Success.
recResult.setResultCode(XKMSURI.result_major_success, null);
```

The `KeyBinding` needs to be set for a `RecoverResult`. You can accomplish this with the following code:

```
//Create a xkms:KeyBinding to add status and X509Data
XSKeyInfo xsInfo =
    //Instance of oracle.security.xmlsec.dsig.XSKeyInfo,
    //which contains X509Data
KeyBinding keyBinding = new KeyBinding(sigDoc);
keyBinding.setKeyInfo(xsInfo);

//Create a status element and add reasons.
//Status is set to Invalid because the service can decide
//to revoke the key binding in the case of recovery.

Status responseStatus = new Status(sigDoc);
responseStatus.addInvalidReason(XKMSURI.reasonCode_IssuerTrust);
responseStatus.addInvalidReason(XKMSURI.reasonCode_RevocationStatus);
responseStatus.addInvalidReason(XKMSURI.reasonCode_ValidityInterval);
responseStatus.addInvalidReason(XKMSURI.reasonCode_Signature);
responseStatus.setStatusValue(XKMSURI.kbs_Invalid);

keyBinding.setStatus(responseStatus);

//Set KeyBinding into RecoverResult
recResult.addKeyBinding(keyBinding);
```

Finally, this example shows how to set the recovered `PrivateKey` into the `RecoverResult`:

```
//Create an Instance of dsig:XEEncryptedData
XEEncryptedData encryptedData = //Instance of
oracle.security.xmlsec.enc.XEEncryptedData

//Create an instance of oracle.security.xmlsec.xkms.xkrss.PrivateKey
PrivateKey privKey = new PrivateKey(sigDoc);
privKey.setEncryptedData(encryptedData);

//Add PrivateKey to RecoverResult
recResult.setPrivateKey(privKey);
```

11.4 The Oracle XKMS Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes, interfaces, and methods available in Oracle XKMS API.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

12

Oracle JSON Web Token

Oracle JSON Web Token API is a full Java solution that provides extensive support for JSON Web Token (JWT). You can use the API to construct Base64url encoded tokens and set the token's header and claim parameter values, parse and verify tokens, and sign and serialize tokens.

Oracle JSON Web Token, introduced in Release 11*g*, provides support for the JSON Web Token (JWT) standard.

- Oracle JSON Web Token Features and Benefits
- Setting Up Your Oracle JSON Web Token Environment
- Using Core Classes and Interfaces
- Examples of Oracle JSON Web Token Usage
- The Oracle JSON Web Token Java API Reference

12.1 Oracle JSON Web Token Features and Benefits

JSON Web Token (JWT) represents claims to be transferred between two parties. JWT is a compact token format intended for space- constrained environments such as HTTP Authorization headers and URI query parameters. You can use the API to construct Base64url encoded tokens and set the token's header and claim parameter values, parse and verify tokens, and sign and serialize tokens.

This section introduces JWT concepts and key features of Oracle JSON Web Token.

- About JSON Web Token
- Oracle JSON Web Token Features

12.1.1 About JSON Web Token

JSON Web Token (JWT) represents claims to be transferred between two parties. JWT is a compact token format intended for space- constrained environments such as HTTP Authorization headers and URI query parameters. A JSON object is digitally signed using a JSON Web Signature (JWS) and optionally encrypted using JSON Web Encryption (JWE).

The claims in a JWT are encoded as a JSON object that is base64url encoded and consists of zero or more name/value pairs (or members), where the names are strings and the values are arbitrary JSON values. Each member is a claim represented by the JWT.

The JWT is represented as the concatenation of three segments:

- JWT Header Segment describes the cryptographic operations applied to the token.
- JWT Claim Segment encodes the claims contained in the JWT.

- JWT Crypto Segment contains the cryptographic material that secures the contents of the token.

The segments are separated by period ('.') characters. All three segments are always Base64url encoded values.

> **See Also:**
>
> JSON Web Token IETF draft document at `http://tools.ietf.org/html/draft-jones-json-web-token-05`.

12.1.2 Oracle JSON Web Token Features

You can use the API to construct Base64url encoded tokens and set the token's header and claim parameter values, parse and verify tokens, and sign and serialize tokens.

Oracle JSON Web Token is a full Java solution that provides extensive support for JWT tokens. You can use the API to:

- construct Base64url encoded tokens and set the token's header and claim parameter values, including user-defined headers
- parse and verify tokens
- sign and serialize tokens

The oracle.security.jwt.JwtToken class represents the JSON Web Token (JWT). Representative methods of oracle.security.jwt.JwtToken include:

- setAlgorithm(String), getAlgorithm()
- signAndSerialize(PrivateKey)
- serializeUnsigned()
- claim methods such as setPrincipal(String), getPrincipal(), getIssuer()

For details, see the tables of header and claim parameter names and corresponding get/set methods in the Javadoc.

> **See Also:**
>
> The Oracle JSON Web Token Java API Reference.

12.2 Setting Up Your Oracle JSON Web Token Environment

You can setup Oracle JSON Web Token environment by installing Oracle Security Developer Tools and Java Development Kit (JDK), and setting the CLASSPATH variable to all of the required jar and class files.

The Oracle Security Developer Tools are installed with Oracle WebLogic Server in `ORACLE_HOME`.

In order to use Oracle JSON Web Token, your system must have the Java Development Kit (JDK) version 1.6 or higher.

Your `CLASSPATH` environment variable must contain the full path and file names to all of the required jar and class files. Make sure the following items are included in your `CLASSPATH`:

- `osdt_core.jar` file
- `osdt_cert.jar` file
- `jackson-core-1.1.1.jar` file
- `jackson-mapper-1.1.1.jar` file

For example, your `CLASSPATH` might look like this:

```
setenv CLASSPATH $CLASSPATH:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_core.jar:
$ORACLE_HOME/modules/oracle.osdt_11.1.1/osdt_cert.jar:
$Jackson.library.path/jackson-core-1.1.1.jar
$Jackson.library.path/jackson-mapper-1.1.1.jar
```

At run-time, the following locations are searched for the Jackson jars:

1. If present, the jars are loaded from the system class path.

2. If the jars are not present in the system class path, the system property `Jackson.library.path` is examined. If present, the jars are loaded from that location for both Java SE and Java EE clients.

3. If the system property `Jackson.library.path` is not set or the Jackson jars are not found there, they are picked up from the predefined location `$ORACLE_HOME/ modules` (for Java EE environment) and from the present directory (for Java SE client).

> ✏️ **See Also:**
>
> Setting the CLASSPATH Environment Variable

12.3 Using Core Classes and Interfaces

The Oracle JSON Web Token consists of the oracle.security.restsec.jwt.JwtToken class. Key functions by this class include constructing a JWT token, setting the parameter values of the JWT token, signing the token, verifying the token, and token serialization.

Examples of Oracle JSON Web Token Usage demonstrates how to use Oracle JSON Web Token.

12.4 Examples of Oracle JSON Web Token Usage

You can refer the examples of constructing a JWT token, signing the token, verifying the token, and serializing the token without signing to know how to use Oracle JSON Web Token.

This section provides some examples of using Oracle JSON Web Token.

- Creating the JWT Token
- Signing the JWT Token
- Verifying the JWT Token
- Serializing the JWT Token without Signing

> **Note:**
>
> These are specific examples to demonstrate how to use Oracle JSON Web Token. For details and other options for using the methods described here, see the JWT javadoc (The Oracle JSON Web Token Java API Reference).

12.4.1 Creating the JWT Token

Creating the JWT token involves creating the object itself, then setting header and claim parameters as needed.

The steps are as follows:

1. To create a JWT token, begin by using the constructor method JwtToken() to create a JwtToken object.

   ```
   JwtToken jwtToken = new JwtToken();
   ```

   You can use various setter methods to set the parameter values of the JWT token.

2. To set header parameters, the header parameter `alg` must first be set; use the setAlgorithm(String) and getAlgorithm() methods, respectively, to set and get this parameter. By default, the `alg` parameter is set to "`none`" implying that you do not want to sign the token.

   Use the `setHeaderParameter(String, Object)` method to set a user-defined header parameter in the JWT header segment.

3. Oracle JSON Web Token provides methods to set claim parameters exp, iat, iss, aud, prn. All the claim parameters are optional.

   Use the setClaimParameter(String, Object) method to set the user-defined claim parameter in the JWT claim segment.

12.4.2 Signing the JWT Token

Signing a token involves actions such as creating a token instance, setting token parameters, and finally signing the token.

The steps are as follows:

1. Create and sign the JWT token, by first creating the instance of the JwtToken class:

   ```
   JwtToken jwtToken = new JwtToken(String);
   ```

2. Next set the parameters like algorithm, issuer, expiry time, other claims and so on:

```
jwtToken.setAlgorithm(JwtToken.SIGN_ALGORITHM.HS256.toString());
jwtToken.setType(JwtToken.JWT);
jwtToken.setIssuer("my.company.com");
jwtToken.setPrincipal("john.doe");
```

3. Finally obtain the private key and sign the token with a secret key or private key:

```
PrivateKey privateKey ;
String jwtString = jwtToken.signAndSerialize(privateKey);
```

12.4.3 Verifying the JWT Token

Verifying a token involves actions such as reading the token from the HTTP header, checking the token issuer, and so on.

This example code verifies the expiry date and token issuer:

```
// Read the JWT token as a String from HTTP header
String jwtStr = "eyJ.eyJp.dB";
JwtToken token = new JwtToken(jwtStr);

// Validate the issued and expiry time stamp.
if (token.getExpiryTime().after(new Date())) {
...
...
}

// Get the issuer from the token
String issuer = token.getIssuer();
```

12.4.4 Serializing the JWT Token without Signing

If the JWT token is not required to be digitally signed, you can serialize the token without signing.

Example:

```
JwtToken jwtToken = new JwtToken();
jwtToken.setType(JwtToken.JWT);
jwtToken.setIssuer("my.example.com");
jwtToken.setPrincipal("john.doe");
String jwtString = jwtToken.serializeUnsigned();
```

12.5 The Oracle JSON Web Token Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods available in the Oracle JSON Web Token API.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

A

Migrating to the JCE Framework

The Oracle Security Developer Tools framework introduced changes to low-level libraries starting in 11*g* Release 1 to comply with the Java Cryptography Extension (JCE) framework. The changes affected both client programs and higher-level libraries of the Oracle Security Developer Tools. You can migrate your legacy programs to leverage the JCE functions.

This chapter describes how the changes affected the toolkit architecture, and explain how you can migrate your legacy programs to leverage the JCE functions. It contains these topics:

- About The JCE Framework
- Understanding JCE Keys
- Converting Between OSDT Key Objects and JCE Key Objects
- Working with JCE Certificates
- Working with JCE Certificate Revocation Lists (CRLs)
- Using JCE Keystores
- The Oracle JCE Java API Reference

A.1 About The JCE Framework

Prior to Oracle Fusion Middleware 11*g*, Oracle Security Developer Tools used a cryptographic engine that was developed prior to the adoption of JCE in the market. To enable applications (including Oracle WebLogic Server) to continue their move to adopt JCE, the Oracle Security Developer Tools have standardized on low-level libraries that are compliant with the Java Cryptography Extension (JCE) framework with Oracle Fusion Middleware 11*g*.

Benefits of the JCE framework include:

- standards-based implementations of cryptographic and certificate management engines
- a pluggable JCE provider architecture that enables you to leverage third-party JCE provider implementations
- the ability to use third-party providers as the cryptographic engine

Additional Reading

This chapter's primary focus is on the changes to the Oracle Security Developer Tools for the JCE framework, and how to migrate your existing security artifacts to JCE objects.

A.2 Understanding JCE Keys

As of Release 11*g*R1, the higher level toolkits (Oracle XML Security, Oracle Web Services Security, Oracle CMS, Oracle S/MIME, Oracle XKMS) have changed so that instead of taking Oracle cryptographic keys and certificates, they take standard JCE keys and certificates.

Thus, APIs that were taking:

```
oracle.security.crypto.core.PublicKey
```

now take a:

```
java.security.PublicKey
```

> **Note:**
>
> This discussion highlights changes in the Oracle Security Developer Tools in support of JCE. For fuller details of all the available cryptographic functions, see the API documentation.

- `oracle.security.crypto.core.PublicKey` changed to `java.security.PublicKey`
- `oracle.security.crypto.core.PrivateKey` changed to `java.security.PrivateKey`
- `oracle.security.crypto.core.SymmetricKey` changed to `javax.crypto.SecretKey`

A.3 Converting Between OSDT Key Objects and JCE Key Objects

You can convert keys from Oracle Security Developer Tools (OSDT) objects to JCE objects and vice versa.

If you are using a java.security.KeyStore to store your keys, you will directly get a java.security.PrivateKey object from it, so you do not need to do any conversion.

However if you are using a oracle.security.crypto.cert.PKCS12 object to store your keys, you will get an oracle.security.crypto.core.PrivateKey from it, and then you need to convert to a java.security.PrivateKey object.

A.3.1 Converting a Private Key from OSDT to JCE Object

You can convert keys in Oracle Security Developer Tools (OSDT) format to JCE objects.

Here is an example:

```
//***** Conversion of PrivateKeys from OSDT -> JCE *******
{
```

```
                    // Example code to convert an RSAPrivateKey (non CRT) to JCE
                    oracle.security.crypto.core.RSAPrivateKey osdtKey = null;
                    RSAPrivateKeySpec keySpec = new RSAPrivateKeySpec(
                    osdtKey.getModulus(), osdtKey.getExponent());
                    KeyFactory kf = KeyFactory.getInstance("RSA");
                    RSAPrivateKey jceKey = (RSAPrivateKey)kf.generatePrivate(keySpec);
                    }

                    {
                    // Example code to convert an RSAPrivateKey (CRT) to JCE
                    oracle.security.crypto.core.RSAPrivateKey osdtKey = null;
                    RSAPrivateKeySpec keySpec = new RSAPrivateCrtKeySpec(
                    osdtKey.getModulus(),
                    osdtKey.getPublicExponent(),
                    osdtKey.getExponent(),
                    osdtKey.getPrimeP(),
                    osdtKey.getPrimeQ(),
                    osdtKey.getPrimeExponentP(),
                    osdtKey.getPrimeExponentQ(),
                    osdtKey.getCrtCoefficient());
                    KeyFactory kf = KeyFactory.getInstance("RSA");
                    RSAPrivateCrtKey jceKey = (RSAPrivateCrtKey)kf.generatePrivate(keySpec);

                    }

                    {
                    // Example code to convert a DSAPrivateKey to JCE
                    oracle.security.crypto.core.DSAPrivateKey osdtKey = null;
                    DSAPrivateKeySpec keySpec = new DSAPrivateKeySpec(
                    osdtKey.getX(),
                    osdtKey.getParams().getP(),
                    osdtKey.getParams().getQ(),
                    osdtKey.getParams().getG());

                    KeyFactory kf = KeyFactory.getInstance("DSA");
                    DSAPrivateKey jceKey = (DSAPrivateKey)kf.generatePrivate(keySpec);

                    }

                    {
                    // Example code to convert a DHPrivateKey to JCE
                    oracle.security.crypto.core.DHPrivateKey osdtKey = null;

                    // Note q is assumed to be (p-1)/2
                    DHPrivateKeySpec keySpec = new DHPrivateKeySpec(
                    osdtKey.getX(),
                    osdtKey.getParams().getP(),
                    osdtKey.getParams().getG());

                    KeyFactory kf = KeyFactory.getInstance("DiffieHelman");
                    DHPrivateKey jceKey = (DHPrivateKey)kf.generatePrivate(keySpec);

                    }
```

A.3.2 Converting a Private Key from JCE Object to OSDT Object

You can convert private key objects from JCE to OSDT format.

Here is an example:

```
//***** Conversion or Private Keys from JCE -> OSDT *******
{
// Example code to convert an RSAPrivateKey (non CRT) to OSDT
RSAPrivateKey jceKey = null;
oracle.security.crypto.core.RSAPrivateKey osdtKey =
new oracle.security.crypto.core.RSAPrivateKey(
jceKey.getModulus(),
jceKey.getPrivateExponent());
}

{
// Example code to convert an RSAPrivateKey (CRT) to OSDT
RSAPrivateCrtKey jceKey = null;
oracle.security.crypto.core.RSAPrivateKey osdtKey =
new oracle.security.crypto.core.RSAPrivateKey(
jceKey.getModulus(),
jceKey.getPrivateExponent(),
jceKey.getPublicExponent(),
jceKey.getPrimeP(),
jceKey.getPrimeQ(),
jceKey.getPrimeExponentP(),
jceKey.getPrimeExponentQ(),
jceKey.getCrtCoefficient());
}

{
// Example code to convert an DSAPrivateKey to OSDT
DSAPrivateKey jceKey = null;
oracle.security.crypto.core.DSAPrivateKey osdtKey =
new oracle.security.crypto.core.DSAPrivateKey(
jceKey.getX(),
new oracle.security.crypto.core.DSAParams(
jceKey.getParams().getP(),
jceKey.getParams().getQ(),
jceKey.getParams().getG())));
}

{
// Example code to convert an DHPrivateKey to OSDT
DHPrivateKey jceKey = null;

// Note calculate q = (p-1)/2
oracle.security.crypto.core.DHPrivateKey osdtKey =
new oracle.security.crypto.core.DHPrivateKey(
jceKey.getX(),
new oracle.security.crypto.core.DHParams(
jceKey.getParams().getP(),
jceKey.getParams().getG(),
jceKey.getParams().getP().subtract(new BigInteger("1")).divide(new
BigInteger("2"))));
}
```

A.4 Working with JCE Certificates

As of Release 11*g*R1, `oracle.security.crypto.cert.X509` is changed to `java.security.cert.X509Certificate`. Several utility methods are available for creating and working with JCE certificates. An X509Certificate object can be created from an input stream using java.security.cert.CertificateFactory.

The input stream can be one of the following:

- • a FileInputSream, if the certificate is stored in a file, or

- • a ByteArrayInputStream, if we got the encoded bytes from an old X509 object, or

- • any other sources.

For example, the following code converts an Oracle Security Developer Tools certificate to a JCE certificate:

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");

X509Certificate cert = (X509Certificate)cf.generateCertificate(
    new FileInputStream(certFileName);
```

where `certFileName` is the name of the certificate file.

A.5 Working with JCE Certificate Revocation Lists (CRLs)

In Release 11*g*R1, `oracle.security.crypto.cert.CRL` is replaced by `java.security.cert.CRL`. You can create the `java.security.cert.CRL` object from an input stream by using `java.security.cert.CertificateFactory`.

The input stream can be one of the following:

- • `FileInputSream`, if the CRL is stored in a file

- • `ByteArrayInputStream`, if the encoded bytes were obtained from an old `oracle.security.crypto.cert.CRL` object

- • any other source

Here is an example of a CRL object creation:

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");

509Certificate cert = (X509Certificate)cf.generateCRL(
    new FileInputStream(crlFileName));
```

where the `crlFileName` is the name of the CRL file.

A.6 Using JCE Keystores

Oracle Security Developer Tools provide four types of keystore: JKS keystore, Oracle wallet, PKCS12 wallet, and PKCS8 wallet.

These are:

1. the JKS keystore, which is Oracle's implementation of the java.security.KeyStore interface

2. the Oracle wallet, which is Oracle's implementation of the java.security.KeyStore interface

3. the PKCS12 wallet, which is a proprietary Oracle interface/implementation of PKCS12

4. the PKCS8 wallet, which is a proprietary Oracle interface/implementation of PKCS8

A.6.1 Working with standard KeyStore-type Wallets

You can instantiate a Keystore object using an Oracle provider, load a Keystore file, and retrieve a certificate.

Creating a PKCS12 Wallet

This example instantiates a PKCS12 wallet for the Oracle provider:

```
java.security.KeyStore keystore = KeyStore.getInstance("PKCS12", "OraclePKI");
```

Loading a Keystore File

You perform this task with the `keystore.load` method:

```
keystore.load(new FileInputStream(walletFile), pass);
```

Retrieving a Certificate

To retrieve a certificate and private key using an alias:

```
Key key = keystore.getKey(alias);

Certificate cert = keystore.getCert(alias);
```

If the alias is not known in advance, you can list all aliases by calling:

```
keystore.aliases();
```

A.6.2 Working with PKCS12 and PKCS8 Wallets

If you maintain keystores in the PKCS12 or PKCS8 oracle wallet format, you can retrieve keys, certificates or CRLs from those stores in Oracle Security Developer Tools format.

- Retrieving a PKCS Object
- Retrieving a Certificate
- Retrieving CRLs

A.6.2.1 Retrieving a PKCS Object

In Oracle wallets, the key is found in oracle.security.crypto.core.PrivateKey.

After retrieval, you can convert the keys into the JCE key format, using the utility class PhaosJCEKeyTranslator.

For more information, see Converting Between OSDT Key Objects and JCE Key Objects.

A.6.2.2 Retrieving a Certificate

In Oracle wallets, the certificate is found in oracle.security.crypto.cert.X509.

After retrieval, you can:

1. get the encoded value of the X509 certificate, for example X509.getEncoded();

2. use the CertificateFactory to create a X509Certificate instance, based on the encoded bytes value.

For more information, see Working with JCE Certificates.

A.6.2.3 Retrieving CRLs

In Oracle wallets, the CRL is found in oracle.security.crypto.cert.CRL.

After retrieval, you can:

1. get the encoded value of the CRL, for example CRL.getEncoded();

2. use the CertificateFactory to create a java.security.cert.CRL instance, based on the encoded bytes value.

For more information, see Working with JCE Certificate Revocation Lists (CRLs).

A.7 The Oracle JCE Java API Reference

The Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools guide explains the classes and methods available in the Oracle JCE framework.

You can access the guide at:

Oracle Fusion Middleware Java API Reference for Oracle Security Developer Tools

B

References

A list of standards forms the basis of Oracle Security Developer Tools. You can refer these standards and protocols referenced in this document.

Table B-1   Security Standards and Protocols

Document	Reference
[AES-128]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *Block Encryption Algorithms*,http://www.w3.org/2001/04/xmlenc#aes128-cbc and http://www.w3.org/2001/04/xmlenc#kw-aes128
[AES-192]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *Block Encryption Algorithms*,http://www.w3.org/2001/04/xmlenc#aes192-cbc and http://www.w3.org/2001/04/xmlenc#kw-aes192
[AES-256]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *Block Encryption Algorithms*,http://www.w3.org/2001/04/xmlenc#aes256-cbc and http://www.w3.org/2001/04/xmlenc#kw-aes256
Cryptography	Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C (2nd Edition)*, John Wiley and Sons, 1996.
Cryptography	William Stallings, *Cryptography and Network Security: Principles and Practice (3rd Edition)*, Prentice Hall, 2002.
[DES-EDE]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *Block Encryption Algorithms*,http://www.w3.org/2001/04/xmlenc#aes128-cbc and http://www.w3.org/2001/04/xmlenc#kw-tripledes
Diffie-Hellman Key Agreement	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *Diffie-Hellman Key Agreement*, http://www.w3.org/2001/04/xmlenc#dh
[DSA-SHA]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *DSA*, http://www.w3.org/TR/xmlenc-core/
JSON Web Token	JSON Web Token (JWT) Draft. See http://tools.ietf.org/html/draft-jones-json-web-token-05
Liberty Alliance	Liberty Alliance Project ID-FF 1.2 and ID-WSF 2.0 Specifications, http://www.projectliberty.org/resources/specifications.php
[PKCS]	RSA Laboratories, "Public-Key Cryptography Standards (PKCS)", http://www.rsasecurity.com/rsalabs/node.asp?id=2125
[PKCS1]	RSA Laboratories, "PKCS #1: RSA Cryptography Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=2125
[PKCS3]	RSA Laboratories, "PKCS #3: Diffie-Hellman Key Agreement Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=2126

Table B-1　(Cont.) Security Standards and Protocols

Document	Reference
[PKCS5]	RSA Laboratories, "PKCS #5: Password-Based Cryptography Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=2127
[PKCS6]	RSA Laboratories, "PKCS #6: Extended-Certificate Syntax Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=2128
[PKCS7]	RSA Laboratories, "PKCS #7: Cryptographic Message Syntax Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=21299
[PKCS8]	RSA Laboratories, "PKCS #8: Private-Key Information Syntax Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=2130
[PKCS9]	RSA Laboratories, "PKCS #9: Selected Attribute Types", http://www.rsasecurity.com/rsalabs/node.asp?id=2131
[PKCS10]	RSA Laboratories, "PKCS #10: Certification Request Syntax Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=2132
[PKCS11]	RSA Laboratories, "PKCS #11: Cryptographic Token Interface Standard", http://www.rsasecurity.com/rsalabs/node.asp?id=2133
[RFC2311]	S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, L. Repka, "S/MIME Version 2 Message Specification". March 1998, http://www.ietf.org/rfc/rfc2311.txt
[RFC2459]	R. Housley, W. Ford, W. Polk, D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile". January 1999, http://www.ietf.org/rfc/rfc2459.txt
[RFC2510]	C. Adams, S. Farrell, "Internet X.509 Public Key Infrastructure Certificate Management Protocols". March 1999, http://www.ietf.org/rfc/rfc2510.txt
[RFC2511]	M. Myers, C. Adams, D. Solo, D. Kemp, "Internet X.509 Certificate Request Message Format". March 1999, http://www.ietf.org/rfc/rfc2511.txt
[RFC2560]	M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP". June 1999, http://www.ietf.org/rfc/rfc2560.txt
[RFC2630]	R. Housley, "Cryptographic Message Syntax". June 1999, http://www.ietf.org/rfc/rfc2630.txt
[RFC2634]	P. Hoffman, Editor, "Enhanced Security Services for S/MIME". June 1999, http://www.ietf.org/rfc/rfc2634.txt
[RFC3161]	C. Adams, P. Cain, D. Pinkas, R. Zuccherato, "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)". August 2001, http://www.ietf.org/rfc/rfc3161.txt
[RFC3274]	P. Gutmann, "Compressed Data Content Type for Cryptographic Message Syntax (CMS)". June 2002, http://www.ietf.org/rfc/rfc3274.txt
[RFC3275]	D. Eastlake, J. Reagle, D. Solo, "(Extensible Markup Language) XML-Signature Syntax and Processing". March 2002, http://www.ietf.org/rfc/rfc3275.txt

Table B-1   (Cont.) Security Standards and Protocols

Document	Reference
[RFC3280]	R. Housley, W. Polk, W. Ford, D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile". April 2002, http://www.ietf.org/rfc/rfc3280.txt
[RSA-OAEP]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *RSA-OAEP*, http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
[RSA-SHA]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *PKCS1 (RSA-SHA1)*, http://www.w3.org/TR/xmlenc-core/
[RSAES-OAEP]	R. Housley. "RFC 3560 - Use of the RSAES-OAEP Key Transport Algorithm in Cryptographic Message Syntax (CMS)," http://www.faqs.org/rfcs/rfc3560.html
[RSAES-PKCS1-v1_5]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. See *RSA Version 1.5*, http://www.w3.org/2001/04/xmlenc#rsa-1_5
[SAML]	OASIS Security Services (SAML) TC, http://www.oasis-open.org/committees/security/
[WSS]	OASIS Web Services Security (WSS) TC, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
[WSS v1.0]	OASIS Standards and Other Approved Work, http://www.oasis-open.org/specs/index.php#wssv1.0. This OASIS standard contains the following: 1. OASIS WSS SOAP Message Security Specification 2. OASIS WSS Username Token Profile Specification 3. OASIS WSS X.509 Certificate Token Profile Specification 4. OASIS WSS SAML Assertion Token Profile Specification 5. OASIS WSS REL Token Profile Specification
[XKMS 2.0]	W. Ford, P. Hallam-Baker, B. Fox, B. Dillaway, B. LaMacchia, J. Epstein, J. Lapp, "XML Key Management Specification", 30 March 2001, http://www.w3.org/TR/xkms/.
[xml.com]	O'Reilly xml.com, http://www.xml.com/
[XML 1.0]	W3C Recommendation XML 1.0: Extensible Markup Language (XML) 1.0 (Third Edition), 04 February 2004. http://www.w3.org/TR/REC-xml/
[XML Canonicalization]	W3C Recommendation Canonical XML: Canonical XML Version 1.0, 15 March 2001. http://www.w3.org/TR/xml-c14n
[Exclusive XML Canonicalization]	W3C Recommendation Exclusive XML Canonicalization: Exclusive XML Canonicalization Version 1.0, 15 March 2001. http://www.w3.org/TR/xml-exc-c14n/
[XML Decryption Transform]	W3C Recommendation XML Decryption Transform: Decryption Transform for XML Signature, 10 December 2002. http://www.w3.org/TR/xmlenc-decrypt

Table B-1   (Cont.) Security Standards and Protocols

Document	Reference
[XML Encryption]	W3C Recommendation XML Encryption: XML Encryption Syntax and Processing, 10 December 2002. `http://www.w3.org/TR/xmlenc-core/`
[XML FAQ]	Java Technology and XML FAQs, `http://www.oracle.com/technetwork/java/faq-141681.html`
[XML Signatures]	W3C Recommendation XML Signature: XML-Signature Syntax and Processing, 12 February 2002. `http://www.w3.org/TR/xmldsig-core/`