Oracle® Application Server

Web Services Security Guide

10*g* Release 3 (10.1.3)

B15979-01

January 2006

ORACLE®

Oracle Application Server Web Services Security Guide 10*g* Release 3 (10.1.3)

B15979-01

Primary Author: Thomas Pfaeffle

Contributor: Moushmi Banerjee, William Bathurst, Anirban Chattergee, Dheeraj Goswami, Ganesh Kirti, Ramana Turlapati

Contents

2 Configuring Web Service Security

3 Administering Web Services Security

4 Building Secure Web Services

Index

Preface

This book describes the different security strategies that can be applied to a Web service in Oracle Application Server Web Services. The strategies that can be employed are username token, X.509 token, SAML token, XML encryption, and XML signature. The book describes the configuration options available for the client and the service, for inbound messages and outbound messages. It also describes how to configure these options for a number of different scenarios.

Intended Audience

This book is intended for software developers and architects who want to add security to Web services. It is expected that the reader has some experience with Web technology, OracleAS Web Services, the J2EE environment, and Java and XML programming principles.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information on OC4J, Web services, and security, see the following manuals:

- *Oracle Application Server Web Services Developer's Guide*

  This book describes how to use the WebServicesAssembler tool to assemble Web services from a variety of resources: Java classes, EJBs, database resources, JMS destinations and J2SE 5.0 Annotations. You can also assemble REST-style Web services. The *Developers Guide* also describes how to assemble J2SE and J2EE clients to access these services. This book includes descriptions of the message formats and datatypes supported by OracleAS Web Services.

- *Oracle Application Server Advanced Web Services Developer's Guide*

  This book describes topics beyond basic Web service assembly. For example, it describes how to diagnose common interoperability problems, how to enable Web service management features (such as reliability, auditing, and logging), and how to use custom serialization of Java value types.

  This book also describes how to employ the Web Service Invocation Framework (WSIF), the Web Service Provider API, message attachments, and management features (reliability, logging, and auditing). It also describes alternative Web service strategies, such as using JMS as a transport mechanism.

- *Oracle Containers for J2EE Security Guide*

  This book (not to be confused with the *Oracle Application Server 10g Security Guide*), describes security features and implementations particular to OC4J. This includes information about using JAAS, the Java Authentication and Authorization Service, and other Java security technologies.

- *Oracle Containers for J2EE Services Guide*

  This book provides information about standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS, and the Oracle Application Server Java Object Cache.

- *Oracle Containers for J2EE Configuration and Administration Guide*

  This book discusses how to configure and administer applications for OC4J, including use of the Oracle Enterprise Manager 10g Application Server Control Console, use of standards-compliant MBeans provided with OC4J, and, where appropriate, direct use of OC4J-specific XML configuration files.

- *Oracle Containers for J2EE Deployment Guide*

  This book covers information and procedures for deploying an application to an OC4J environment. This includes discussion of the deployment plan editor that comes with Oracle Enterprise Manager 10g.

- *Oracle Containers for J2EE Developer's Guide*

  This discusses items of general interest to developers writing an application to run on OC4J—issues that are not specific to a particular container such as the servlet, EJB, or JSP container. (An example is class loading.)

Available from the Oracle Server Technologies group:

- *Oracle Database Advanced Security Administrator's Guide*

From the Oracle Application Server core documentation group:

- *Oracle Application Server Security Guide*
- *Oracle Application Server Administrator's Guide*
- *Oracle Application Server Certificate Authority Administrator's Guide*
- *Oracle Application Server Single Sign-On Administrator's Guide*
- *Oracle Application Server Enterprise Deployment Guide*

For Oracle Identity Management and Oracle COREid:

- *Oracle Identity Management Administrator's Guide*
- *Oracle Identity Management Guide to Delegated Administration*
- *Oracle Identity Management Application Developer's Guide*
- *Oracle COREid Access and Identity Administration Guide*
- *Oracle COREid Access and Identity Customization Guide*
- *Oracle COREid Access and Identity Deployment Guide*
- *Oracle COREid Access and Identity Developer Guide*
- *Oracle COREid Access and Identity Integration Guide*
- *Oracle COREid Access and Identity Installation Guide*
- *Oracle COREid Access and Identity Introduction*
- *Oracle COREid Access and Identity Schema Description*
- *Oracle COREid Access and Identity Upgrade Guide*

For Oracle Web Services Manager:

- *Oracle Web Services Manager User and Administrator Guide*
- *Oracle Web Services Manager Extensibility Guide*
- *Oracle Web Services Manager Installation and Deployment Guide*
- *Oracle Web Services Manager Upgrade Guide*

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.

1 Introduction

This chapter introduces essential Web service security concepts, standards, and specifications. It is divided into the following sections:

- Web Service Security Concepts
- Web Services Security Support in OracleAS Web Services
- Tool Support for Web Service Security

Historically, the majority of Web services have been based on the same enabling technology that underlies the Web, namely HTTP. As a result, common technologies that secure Web applications, such as basic authentication and SSL, work equally well with Web services. These security technologies have been effective for years for all sorts of online business transactions, and they work equally well for Web services.

SSL, however, does have limitations. SSL for Web services is an all-or-nothing proposition: it often secures the entire wire protocol rather than just the SOAP message sent over the protocol. It doesn't let developers apply different levels of security to different parts of a document. Because of its point-to-point structure, SSL doesn't support chained services or workflow applications where user credentials can be passed through each stop in a transaction chain. This leaves the messages unsecured at each intermediary checkpoint. SSL also does not support the concept of an audit trail.

The answer to providing both message- and transport-level security lies with the OASIS standard, WS-Security, released as a full industry-recognized recommendation in April 2004. WS-Security defines a mechanism for adding transport independence and different levels of security to SOAP messages.

- WS-Security offers multiple ways to authenticate. In WS-Security, it is easy to associate different identities with service requests. These identities can be used to enforce authorization after authentication.
- WS-Security offers support for SOAP traffic involving intermediaries.
- WS-Security is transport-independent, which gives greater transport flexibility.
- WS-Security is targeted security. For example, you can sign or encrypt the whole message body, or just a single XML element of the body payload.

If there is a need to apply integrity and confidentiality at a fine-grained level instead of applying to the entire SOAP message, XML signature and encryption can be used to protect the SOAP body, header block, or portions of either. If the SOAP message needs to be protected beyond the transport session, message-level security can be used. If there is a need to use different forms of authentication, then message-level security authentication tokens can be used, such as username token, X.509 token, or SAML token.

Web Service Security Concepts

The concepts fundamental to security are *authentication*, *message integrity*, and *message confidentiality*.

- *Authentication* — "Who is trying to access my services?" Applications need to know whom they're serving.
- *Message integrity* —"Was the message lost, destroyed, or modified in transit, either accidentally or deliberately?" Applications must validate messages.
- *Message confidentiality*—"Can anybody else read this message?" Applications may need to encrypt data between endpoints, revealing data to only applications that need to see it.

A number of Web services and security-related terms and concepts are used throughout this book. They are described in the following sections.

SOAP

The SOAP protocol is an XML-based standard for Web service request and response messages, which are normally transmitted over the HTTP or HTTPS (HTTP with SSL) protocols. WS-Security is an XML-based extension to the SOAP description, designed to fit within the SOAP message context. To see this, one must first understand the anatomy of a SOAP message.

In the XML schema definition for SOAP, the `<Envelope>` element is the root element of a SOAP message. A SOAP message can also have headers, a body, or a fault element. Example 1–1 illustrates the contents the SOAP `<Envelope>` element.

Example 1–1 Contents of the SOAP <Envelope> Element

```
<env:Envelope>
    <env:Header>
    </env:Header>
  <env:Body>
        .
        .
        .
  </env:Body>
<env:Envelope>
```

Note: This example illustrates a SOAP request envelope. There is also a response envelope, which can contain a SOAP fault element.

Although Web services developers are primarily concerned with the contents of the SOAP body, most security-related elements appear in the SOAP header. The body itself can be encrypted, in which case it contains ciphertext.

Security Policies

A Web service security policy determines the security mechanism that is applied to a SOAP message. Security policies can be applied to the Web service on the server side and to the Web service client. Oracle Application Server Web Services Security recognizes the following types and levels of security policies.

- Inbound Policy
- Outbound Policy
- Global Level Policy
- Port-Level Policy
- Operation-Level Policy

Inbound Policy

An inbound policy determines the security mechanisms applied to the inbound SOAP message. The `decrypt` value and the `verify-username-token` value are two examples of inbound policies. The decrypt value represents a policy which is used for decrypting the inbound SOAP message. The `verify-username-token` value represents a policy; this indicates that the incoming SOAP request is carrying a username token which must be verified and authenticated.

To configure an inbound policy, you can use Application Server Control, JDeveloper, or create the configuration manually. The security configuration appears as subelements beneath an `<inbound>` element. The `<inbound>` element and its configuration can appear in the `oracle-webservcies.xml` file for a Web service application and `<generated_name>_Stub.xml` file for a Web service client. For more information about the contents of the `<inbound>` element, see "Security Elements for Inbound Messages" on page 2-7.

Outbound Policy

An outbound policy determines the security mechanisms applied to the outbound SOAP message. The `<encrypt>` value and the `<username-token>` value are two examples of outbound policies. The `<encrypt>` value represents a policy, which is used for encrypting the outbound SOAP message. The `<username-token>` value represents a policy, which indicates that a username token must be included in the outgoing SOAP message.

To configure an outbound policy, you can use Application Server Control, JDeveloper, or create the configuration manually. The security configuration appears as subelements beneath an `<outbound>` element. The `<outbound>` element and its configuration can be used in the `oracle-webservcies.xml` file for a Web service application and `<generated_name>_Stub.xml` file for a Web service client. For more information about the contents of the `<outbound>` element, see "Security Elements for Outbound Messages" on page 2-11.

Global Level Policy

A global-level security policy is a security policy that is applied to all of the Web services deployed in an OC4J instance. Global-level policies can be configured only on the server side.

You can configure the following items at the global level:

- Keystore, signature and encryption keys
- Nonce configuration
- Inbound policy
- Outbound policy

Port-Level Policy

In the Web service context, a port is analogous to an application. A port-level security policy is a security policy that is applied to all operations defined in that Web service. Port-level policies can be configured on the server and the client side.

You can configure the following items at the port level:

- Keystore, signature and encryption keys
- Nonce configuration
- Inbound policy
- Outbound policy

Operation-Level Policy

An operation-level security policy is a security policy that is applied to a particular operation of a Web service. Operation-level policies can be configured on the server and the client side.

You can configure the following items at the operation level:

- Inbound policy
- Outbound policy

The Request Envelope

A request envelope contains the information required to process a remote call. Each request message contains a message header and a message body. The header stores processing information such as message routing, requirements information, and security information. The body of the message stores the information required to process the call; this information is also known as the *payload*.

The payload contains the following items:

- Name of the service called
- Location of the service
- Parameter name and data passed to the service

The Response Envelope

SOAP response envelope format-type is identical to the request. The only difference is that the `IN` parameter is replaced by an `OUT` parameter. The application's output is included in the SOAP body.

Note: The structure of the request and response envelopes might differ based on whether the service supports the RPC-encoded and/or document-literal message format. For more information on supported message formats, see "Oracle Application Server Web Services Messages" in the *Oracle Application Server Web Services Developer's Guide*.

XML Digital Signatures

The XML signature specification describes digital signature processing rules and syntax. XML signatures provide integrity, message authentication, and/or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.

An XML signature contains the basic hash of the signed document, along with information that tells the recipient of the document what data was signed and which algorithms were used. An XML signature can be included inside the document to which the signature applies, or it can exist in a separate document. An XML signature can be applied to document subsets, or even to non-XML data.

XML Signature supports various types of signature, including:

- enveloped—the signature contains the data that is being signed; the signature and element being signed are embedded in the same element.
- enveloping—the signature containing the signed data is embedded in itself.
- detached—the signature corresponds to data external to the signature element.

For more information about XML Signature, see the specification at the following Web site.

http://www.w3.org/TR/xmldsig-core

XML Signatures and Web Services

For a Web service, XML digital signatures ensure the validity, and identity of SOAP messages sent between Web service clients and services. They are similar in concept to a handwritten signature, where only one person, the signer, can produce the signature. Signature validation is performed in two steps.

1. Compute the message digests of all SOAP message elements to be signed, then place the digests in the `<SignedInfo>` element of the message signature.
2. Compute the message digest of the `<SignedInfo>` element, then sign this digest with the signer's private key. Place the key used for signing in the `<KeyInfo>` element.

The signed SOAP message then is transmitted and verified at the Web service endpoint by the signer's public key. The following steps illustrate how verification works.

1. Recalculate the value of the `<SignedInfo>` element using the digest algorithm specified in the `<SignatureMethod>` element. Use the public key to verify that the value of the `<SignatureValue>` element is correct for the digest of the `<SignedInfo>` element.
2. Recalculate the digests of the references contained within the `<SignedInfo>` element and compare them to the digest values expressed in each `<Reference>` element's corresponding `<DigestValue>` element.

Example 1–2 illustrates the XML representation of signed data. The body of the SOAP message is signed with the RSA-SHA1 algorithm.

Example 1–2 XML Representation of Signed Data

```
<dsig:Signature xmlns="http://www.w3.org/2000/09/xmldsig#"
                       xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
  <dsig:SignedInfo>
    <dsig:CanonicalizationMethod
         Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    <dsig:SignatureMethod
         Algorithm="http://www.w3.org/2000/09/xmldsig#*rsa-sha1*"/>
    <dsig:Reference URI="*#body*">
      <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <dsig:DigestValue>p5vhdagV0tjJafczbLB/I4aonlg=</dsig:DigestValue>
    </dsig:Reference>

  </dsig:SignedInfo>
        <dsig:SignatureValue>QKEJpRGwwRApPFWfA1R/6K4JFwCxyH2Ur0mdzTnzmpf
                             8DNvDVB9xdF9PVsQ68vEey8afbrL/Qwujghoq3gF22VlEBPj
                             TDrRTZ9JgnRVbOt/M/SacHP/BZn9gSfLySpJaQIj7x
                             MUbm5s29JiUvjQ0oR0/Skn+8f+KeQD0QEmbWX8=
  </dsig:SignatureValue>
  <dsig:KeyInfo>
    <wsse:SecurityTokenReference xmlns="http://docs.oasis-open.org/wss/2004/01/
                                  oasis-200401-wss-wssecurity-secext-1.0.xsd"
                                 xmlns:wsse="http://docs.oasis-open.org/wss/2004/
                                  01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
                                  wsu:Id="_str"
                                 xmlns:wsu="http://docs.oasis-open.org/wss/2004/
                                  01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
      <wsse:Reference xmlns="http://docs.oasis-open.org/wss/2004/
                                   01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
                                 xmlns:wsse="http://docs.oasis-open.org/wss/2004/
                                   01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
                                   URI="#_bst"/>
    </wsse:SecurityTokenReference>
  </dsig:KeyInfo>
</dsig:Signature>
```

XML Encryption

The XML encryption specification describes a process for encrypting data and representing the result in XML. The data can be arbitrary; for example, it can be an XML document, an XML element, or XML element content. The result of encrypting data is an XML encryption element which contains (by using one of its children's content) or identifies (by using a URI reference) the cipher data. The standard allows selected parts of a document to be encrypted, while the document as a whole remains valid XML.

XML encryption supports different encryption types such as:

- symmetric key encryption
- public key encryption

For more information about XML Encryption, see the specification at:

http://www.w3.org/TR/xmlenc-core

XML Encryption and Web Services

For a Web service, XML encryption provides security for applications that require a secure exchange of data. While SSL was considered the standard way to secure data exchanges, it has limitations. For example, assume that a document visits several web services before hitting its eventual endpoint. By using XML encryption, the document can be encrypted while at rest, or in transport. It is also possible to encrypt only portions of a document, instead of the whole document.

Example 1–3 illustrates credit card data represented in XML.

Example 1–3 XML Representation of Credit Card Data

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Example 1–4 illustrates the same XML snippet with the credit card number, spending limit, issuing bank, and expiration date encrypted and represented by a cipher value.

Example 1–4 XML Representation of Encrypted Credit Card Data

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
   xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>A23B45C56</CipherValue>
</CipherData>
  </EncryptedData>
</PaymentInfo>
```

SAML

The Security Assertion Markup Language (SAML) is an XML-based framework for exchanging security information. This security information is expressed in the form of assertions about subjects, either human or computer, that have identities in some security domain. One example of a subject is a person, who is identified by his or her email address in a particular Internet DNS domain. Assertions convey information about authentication acts performed by subjects, attributes of subjects, as well as authorization decisions about whether subjects are allowed to access certain resources.

SAML provides a flexible and extensible framework for business and Web services to exchange security information, such as authentication, authorization, and attributes, about their users. For example, SAML allows one service to vouch for a user's authentication to another service. This allows useful activities, such as single sign-on to multiple services.

SAML allows one service to inform another service whether a user is allowed to access a given resource. This allows policy enforcement to be decoupled from resource services.

SAML allows attribute information about a user, such as work-site address, to be communicated between services as the user moves from one site to another.

Message-Level Security

Oracle's Web Services Security model supports end-to-end security via message-level security. This means that in addition to the connection, the SOAP message itself is secure even when it is transferred across multiple intermediaries. The SOAP message is digitally signed and encrypted. You can apply signature and encryption at a granular level by specifying which elements you want to sign and encrypt. The client and the application can authenticate each other by passing various authentication tokens such as username token, X.509 token, or SAML token.

Transport-Level Security

Existing technologies such as SSL/TLS can be used to secure the transport channel. SSL allows two applications to securely connect over a network and authenticate each other. It also allows you to encrypt the data exchanged between the applications. In Oracle's Web Services Security model, this transport security mechanism can be used to provide point-to-point security, data integrity, and data confidentiality.

WS-Security

The Web Services Security (WS-Security) specification describes enhancements to SOAP 1.1 that increase the protection and confidentiality of messages. These enhancements include functionality to secure Simple Object Access Protocol (SOAP) messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens. The goal of the enhancements is to provide greater overall protection for communications over Web services.

WS-Security provides this protection by defining mechanisms for associating tokens with messages. It is completely extensible in that it can support multiple token formats.

The specification includes descriptions of how to include the following security tokens and keys:

- binary security tokens using X.509 and Kerberos ticket
- text-based tokens, such as the username token
- signature and encryption keys

There are no restrictions on the types of credentials that you can include with a message.

Oracle Web Services Security provides support for WS-Security, including XML signature and encryption, and credential propagation through username, X.509, and SAML token profiles. For more information on WS-Security, see the specification at:

`http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

Security Tokens

OracleAS Web Services Security supports the use of the following security tokens.

- Username Token
- X.509 Token
- SAML Token

Username Token

The usename token carries basic authentication information. The `username-token` element propagates user name and password information to authenticate the message. The information provided in the token and the trust relationship provide the basis for establishing the identity of the user.

For more information on the username token profile, see the following Web address.

`http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`

X.509 Token

X.509 certificate with the user's credentials can be passed in the SOAP message. The X.509 token can be used to authenticate the user based on the trust relationship. The X.509 token can also be used to sign the SOAP message.

For more information on the X.509 token profile, see the following Web address.

`http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf`

SAML Token

SAML security tokens are composed of assertions: one or more statements about a user, such as an authentication or attribute statement. SAML tokens are attached to SOAP messages by placing assertion elements inside the header.

For more information on the SAML token profile, see the following Web address.

`http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf`

Keystore

A keystore is used to store certificates, including the certificates of all trusted parties, for use by a program. Through its keystore, an entity, such as OC4J (for example) can authenticate other parties, as well as authenticate itself to other parties. Oracle HTTP Server has what is called a *wallet* for the same purpose.

OracleAS Web Services Security implementation supports the Java Key Store (JKS) and the Oracle Wallet and PKCS#12 keystore types. You can use Oracle Wallet or JKS as your keystore for storing private keys, public keys, and trusted CA certificates.

Oracle Wallet contains a user private key, a user certificate and a set of trust points (that is, the list of root certificates the user trusts). It implements the storage and retrieval of credentials for use with various cryptographic services. Oracle Wallet files are actually PKCS#12 files, which can be parsed, created, and otherwise, manipulated with the `orapki` tool, Oracle Wallet Manager, and Open SSL.

Web Services Security Support in OracleAS Web Services

This section contains the following topics.

Standards Supported by OracleAS Web Services Security

OracleAS Web Services Security is defined by the following specifications and standards.

- Web Services Security (**WS-Security**) defines an open security standard.

  http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

- **XML Digital Signature** defines how to verify the integrity of an XML object.

  http://www.w3.org/TR/xmldsig-core

- **XML Encryption** defines encrypting XML messages.

  http://www.w3.org/TR/xmlenc-core

- **SAML tokens** assert that the SAML user or subject has already been authenticated.

  http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0.pdf

- **X.509 tokens** can be used to authenticate the user based on the trust relationship.

  http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf

- **Username tokens** propagates user name and password information.

  http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf

- Web Service Interoperability (**WS-I**) is a Web service organization that creates, promotes, and supports generic protocols for the interoperable exchange of messages between Web services.

  http://www.ws-i.org

  The following Web Services Interoperability Organization (WS-I) document presents a full discussion of security challenges and threats in today's Web services environment.

  http://www.ws-i.org/Profiles/BasicSecurity/SecurityChallenges-1.0.pdf.

Interceptor Framework

In OracleAS Web Services, all WS-Security features, including digital signatures, encryption, and authentication, are implemented using a prebuilt JAX-RPC handler called an *interceptor*. Whenever a Web service client requests that a message be sent, the SOAP message is "intercepted" by the JAX-RPC handler. The interceptor adds the authentication, signature, and encryption WS-Security elements to the SOAP message, then forwards the message to the receiving Web service.

Each receiving Web service also has interceptors that can decrypt, verify signatures, and authenticate the incoming message.

Figure 1–1 illustrates the actions of the framework. The numbers in the illustration correspond to the following steps.

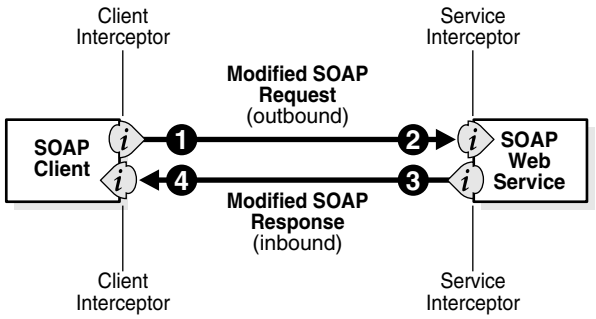
1. A Web service client sends a message to a Web service. The client interceptor adds the authentication, signature, and encryption WS-Security elements to the

outbound SOAP message, then forwards the modified message to the receiving Web service.

2. The service interceptor intercepts the message and decrypts, verifies, and authenticates the message.
3. When sending a response message, the service interceptor in the Web service adds a WS-Security header with integrity and confidentiality.
4. The client interceptor interprets the header and delivers it to the client application.

For more information on common use cases, see Chapter 5, "Secure Web Service Usage Scenarios".

Figure 1–1 Interceptor Framework



Service Security Interceptor

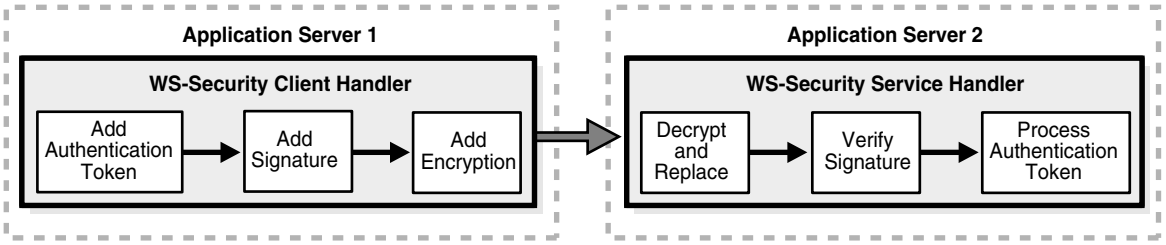
The service security interceptor acts as a filter on incoming SOAP messages. The security settings in the deployment descriptor describes the configuration that drives the interceptor. The service interceptor is responsible for decrypting, verifying, and authenticating (in that order) the incoming request message. When sending a response message, the service interceptor embeds a WS-Security header and provides for integrity and confidentiality of the outbound response message. The main intent of the service interceptor is to establish a security context of execution for SOAP operations

Client Security Interceptor

Client handlers act as peers for the service handlers on the Web service client side. The client interceptor follows a set order of processing when it comes to adding authentication tokens to, signing and encrypting a SOAP message.

Figure 1–2 illustrates the movement of data through the client and service handlers. The client handler first collects authentication information and inserts it as WS-Security token. This is followed by signature computation and encryption of the message. Configuration that resides in a Stub deployment descriptor determines what parts of the message must be integrity/confidentiality protected.

Figure 1–2 Data Flow Through Security Handlers



When the Oracle WS-Security runtime is involved in both the client and server, security interceptors are available at both sides, and these intercepts load security policies specified in configuration files. Many other types of clients (for example, .NET clients) understand and generate WS-Security headers within SOAP envelopes. A receiving interceptor enforces security policies; a sending interceptor intercepts a message and modifies it based on security requirements of the target receiver. For example, if the target receiver requires a message to be signed, the sender signs the message.

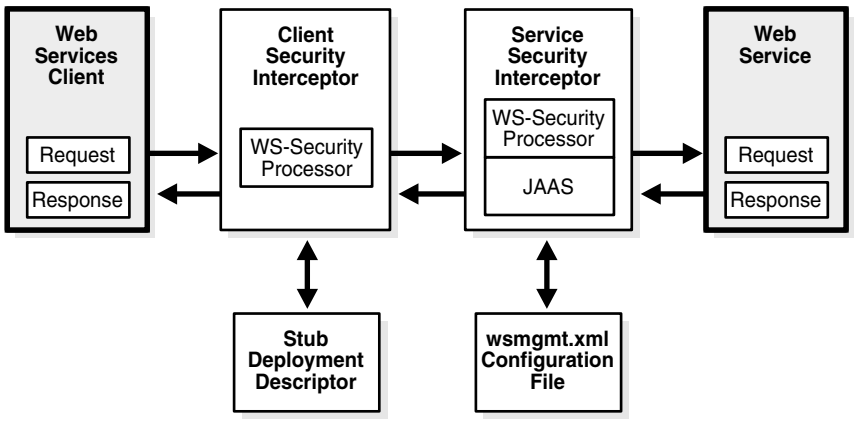
The Oracle WS-Security component is fully integrated with the OracleAS Web Services security infrastructure, which is based on the J2EE security model. For instance, when a request message contains a user name and password, the security handler authenticates by leveraging the underlying OracleAS JAAS Provider infrastructure. If authentication succeeds, a user identity (JAAS Subject) is established in the execution context, and further resource access is authorized by the container based on this identity. It should be noted that the security interceptor helps establish the identity against which the operation can be authorized. Server-side handlers leverage the OracleAS JAAS Provider for the formulation of Subjects and associate the correct Principal instances (users and/or roles) with each Subject during the commit phase of JAAS authentication. After that point, the asserted or authenticated Principal instances contained in Subject are used as a basis for J2EE and JAAS authorization.

Architecture

The OracleAS Web Services architecture is built to support secure communications using a declarative model that requires a minimum of developer effort. An interceptor framework automates common security tasks, such as encrypting portions of the message body.

Figure 1–3 illustrates the overall architecture of WS-Security architecture for the Oracle Web service stack.

Figure 1–3 Web Services Security Architecture



The major components of the Oracle WS-Security architecture are the client and service security interceptors. These interceptors plug into the OracleAS Web Services management framework and work in tandem with other interceptors, including those that implement auditing, logging, and reliability.

Web Service Security Integration

The following sections described how the Web service security interceptor integrates with the OracleAS Web Services Security framework.

- Integration with JAAS
- Integration with Oracle Identity Management
- Integration with External LDAP Servers
- Integration with COREid Access and Identity

Integration with JAAS

OracleAS Web Services provides an implementation of Java Authentication and Authorization Service (JAAS) for J2EE applications that is fully integrated with J2EE declarative security. This allows J2EE applications to take advantage of the JAAS constructs such as principal-based security and pluggable login modules. OracleAS Web Services Security provides out-of-the-box JASS authentication login modules that allow J2EE applications running on OracleAS Web Services to leverage the central security services of Oracle Identity Management.

The JAAS Provider ensures secure access to and execution of Java applications, and integration of Java-based applications with Oracle Application Server Single Sign-On.

Integration with Oracle Identity Management

Oracle Identity Management provides an enterprise infrastructure for securing distributed enterprise applications. Oracle Identity Management is an integrated package of the Oracle Internet Directory, an LDAP server, Single Sign-On, Security and User Management functionality. For more information on these components, see the books *Oracle Identity Management* and the *Oracle Identity Management Administrator's Guide*.

Security in OracleAS Web Services is integrated with Oracle Identity Management. With this integration it is possible to perform the following services:

- Secure Web services and authenticate against OID servers.

- Propagate the Oracle Single Sign-On server authenticated user identity to remote Web services using standards based SAML token.
- Implement fine-grained JAAS authorization for Web services. In this case, JAAS authorization policies are stored in the OID server.

Oracle Identity Management Support for Web Services Authentication Mechanisms Oracle Identity Management integration supports these Web services authentication mechanisms: HTTP basic, digest, username token, X.509 token, and SAML token.

Application Server Web Services Security Integration with Oracle Identity Management The current release of Application Server Web Services Security is integrated with these releases of Oracle Identity Management: 9.0.4, 10.1.2.0.1, and 10.1.2.0.2.

For more information on integrating Oracle Identity Management with OC4J Security, see the *Oracle Containers for J2EE Security Guide*.

Integration with External LDAP Servers

Web services security in OracleAS Web Services is integrated with external (non-Oracle) LDAP servers such as Active Directory. With this integration it is possible to:

- Secure Web services and authenticate against LDAP servers.
- Implement fine-grained JAAS authorization for Web services. In this case, JAAS authorization policies are stored in the XML file `system-jazn-data.xml`.

External LDAP Server Integration Support for Web Services Authentication Mechanisms External LDAP server integration supports these Web services authentication mechanisms: HTTP basic, username token, X.509 token, and SAML token.

Application Server Web Services Security Integration with External LDAP Servers The current release of Application Server Web Services Security can be integrated with these external LDAP servers:

- 10.1.3 Application Server Web Services Security is integrated with Windows 2000 and 2003 Active Directory.
- 10.1.3 Application Server Web Services Security is integrated with Sun Java System Application Server (formerly known as iPlanet).

For more information on integrating Oracle Identity Management with OC4J security, see the *Oracle Containers for J2EE Security Guide*.

Integration with COREid Access and Identity

Oracle COREid Access and Identity is a complete solution for user identity and profile management, single sign-on, and access control. By integrating COREid with OC4J, a single COREid instance allows you to centralize authentication and authorization for one or more instances of OC4J. This single instance allows you to access single sign-on, centralize auditing, and provide stronger authentication options.

The COREid security provider for OC4J can be used to configure authentication and authorization for Web-based applications and single sign-on. It can also be used to configure EJB authentication and Web Service authentication schemes such as username token, X509 and SAML. See the *Oracle Containers for J2EE Security Guide* for more information on Oracle COREid Access and Identity.

Tool Support for Web Service Security

This section provides an overview of the parts of the Web service security configuration that can be set by the JDeveloper and Application Server Control tools. For detailed information of the individual security options that can be controlled by these tools, see the on-line help for Application Server Control and JDeveloper.

This section contains these sub-sections:

- Application Server Control Support for Web Service Security
- JDeveloper Support for Web Service Security
- Oracle Web Services Manager

Application Server Control Support for Web Service Security

Application Server Control can read and modify the security configuration of a deployed Web service. Once the configuration values have been modified and applied, the Web service can be restarted and run with the new values. Application Server Control can be used to set Web service security options on the port and operation level.

At the global level, you can use Application Server Control to set the keystore configuration and the signature and encryption keys. Like the security configuration, if you change the keystore, signature, or encryption values, you must restart the application for the new values to take effect.

For more information on how to use Application Server Control to read and modify the configuration of a deployed Web service, see the Application Server Control on-line help.

Global- and Port-Level Keystore and Identity Certificates

You can choose to employ a global keystore or an application-specific keystore for the Web service port. If you choose a global keystore, it applies to all applications deployed in that OC4J instance. If you choose an port-specific keystore, it must be deployed with your application. For a global keystore, you choose a keystore name, path, and password. You also choose identity certificates for message signature and encryption. For more information on the elements and attributes that allow you to specify a global keystore, see "Keystore Elements" on page 2-5.

If you choose an application-specific keystore, you must also specify the identity certificates for signature and encryption to be used by all operations exposed by this Web service port. For more information on the elements and attributes that allow you to specify a port signature and encryption keys, see "Signature and Encryption Key Elements" on page 2-6.

Note: Both server and trusted certificates are stored in the same keystore. This is different from how Secure Sockets Layer (SSL) or Oracle Remote Method Invocation (ORMI) is used in OC4J. You can use either Oracle Wallet or Java Keystore (JKS) as a keystore. "Using Keystores" on page 3-1 provides more information on how to use these keystores.

Port- and Operation-Level Security Configuration

A security configuration for inbound and outbound SOAP messages can be set on the port-level and operation level. The same security operations (except keystore settings) are available at operation level.

A port-level setting applies to a Web service application. A port-level security configuration is used by all of the operations exposed by the Web service port. An operation-level setting is for a particular operation of the Web service application. A configuration setting made on the operation level overrides the setting made on the port level.

At the port-level you also choose keystore and identity certificates that will apply to the all of the operations on the Web service. These values cannot be overridden on the operation level.

Port-Level and Operation-Level Inbound Policy Configuration

The port and operation level share the same security options for inbound SOAP messages. This section summarizes the options that Application Server Control can set.

- Authentication for inbound messages—lets you specify whether the Web service should expect the message to authenticate by using a username/password (with or without a nonce or timestamp), a X.509 certificate, or a SAML token. "Security Elements for Inbound Messages" on page 2-7 provides more information on the authentication elements that Application Server Control can set.
- Signature verification for inbound messages—lets you specify whether to require the message body to be signed, whether a timestamp is present and a set of acceptable signature algorithms. "Signature Verification Elements for Inbound Messages" on page 2-9 provides more information on the verification elements that Application Server Control can set.
- Decryption for inbound messages—lets you specify whether the message body should be encrypted and a set of acceptable encryption algorithms. "Decryption Elements for Inbound Messages" on page 2-10 provides more information on the decryption elements that Application Server Control can set.

Port- and Operation-Level Outbound Policy Configuration

The port and operation level share the same security options for outbound messages. This section summarizes the options that Application Server Control can set.

- Signing for outbound messages—lets you specify whether to sign the message, add a timestamp, and an acceptable signature algorithm. "Signature Elements for Outbound Messages" on page 2-16 provides more information on the signing elements that Application Server Control can set.
- Encryption for outbound messages—lets you specify to encrypt the body element of the outbound SOAP message by using either a request certificate or a public key by specifying an alias. "Encryption Elements for Outbound Messages" on page 2-17 provides more information on the verification elements that Application Server Control can set.

JDeveloper Support for Web Service Security

JDeveloper can be used to develop OracleAS Web Services and client Web service management configuration files. JDeveloper can aid you in the initial creation of these files or it can be used to add management configuration to existing files.

Wizards in JDeveloper help you configure port and operation level security for inbound and outbound SOAP messages. They cannot be used to set security on the global level.

The port and operation level share the same security options for inbound and outbound messages. This section summarizes the options that JDeveloper wizards can set. For more information on these options, see the JDeveloper on-line help.

- Web service authentication—lets you specify whether the Web service should expect to be accessed via a username and password, a X.509 certificate, or a SAML token.
- signature validation for inbound messages—lets you specify that inbound messages must be signed, and the signature algorithm.
- decryption for inbound messages—lets you specify that inbound messages will be decrypted and the decryption algorithm.
- signing outbound messages—lets you specify whether to add signatures and timestamps to outbound messages and a signing algorithm.
- encryption for outbound messages—lets you specify whether to use a public key or the client-supplied certificate from a previously sent signed message, and the encryption algorithm.
- keystore path—lets you specify credential store locations of the various keys required to implement the chosen security policies.
- signature key—lets you specify the alias and password to access the required key for signing messages.
- encryption key—lets you specify the alias and password to access the required key for decrypting messages.

Oracle Web Services Manager

Oracle Web Services Manager (OWSM) is a Web services security and management solution that provides the visibility and control required to deploy Web services into production. With OWSM, organizations can enjoy a common security infrastructure for all Web Service applications. This allows "best practice" security policies and monitoring to be deployed across existing or new services.

For information on integrating OWSM with OC4J, see the *Oracle Web Services Manager Installation and Deployment Guide*.

With OWSM, an administrator creates security and management policies using a browser-based tool. A typical Web Service security policy could include the following items:

- Decrypt the incoming XML message
- Extract the user's credentials
- Perform an authentication for this user
- Perform an authorization check for this user and this Web Service
- Write a log record of the above information
- If all steps are successful, pass the message to the intended Web Service
- If all of the steps are not successful, return an error and write an exception record

To apply the security policy, OWSM intercepts every incoming request to a Web service and apply any one of the policy items listed above. As the policy is executed,

OWSM collects statistics about its operations and sends them to a monitoring server. The monitor displays errors, service availability data, and so on. As a result, each Web service in an enterprise network can automatically gain security and management control, without the service developer coding extra logic.

For more information on using Oracle Web Services Manager, see the *Oracle Web Services Manager User and Administrator Guide*.

The following sections describe the motivations for using OWSM.

Web Services Access Control and Single Sign-On

OWSM supports single sign-on, including authentication, authorization, and auditing for Web services. For example, a browser-based single sign-on cookie could be used for authentication and authorization in a Web service. OWSM has tight integration with products such as Siteminder and COREid to support enterprise single sign-on solutions. It also supports authentication and authorization through Microsoft Active Directory, and LDAP.

Centralized Security Policy Management with Localized Enforcement

OWSM allows organizations to minimize the duplication of effort required to build security into each service by leveraging a centralized security infrastructure. The Oracle Web service management solution provides the ability to add "best practice" security to an existing Web service, without requiring you to re-work the Service's code.

Place Security Policies in the hands of Security Professionals instead of the Developer

Companies are struggling to manage their security policies. They want to be able to change, develop, and manage their security policies from a central location. This reduces cost, and allows an organization to quickly determine their security policies throughout their applications.

When to Use OWSM to Secure Web Services

Both OWSM and OracleAS Web Services Security provide the ability to secure Web services. The WS-Security implementation within OracleAS Web Services was developed so that Web services can be secured by anybody who uses them. OracleAS Web Services Security is bundled with Oracle 10.1.3 Application Server release, and supports the WS-Security 1.0 specification. It is ideal for deployments where Web services are deployed within the OC4J instance, and security is enforced locally by OC4J.

For large or enterprise-wide deployments where a number of Web services must be centrally managed and secured, then OWSM is recommended.

2

Configuring Web Service Security

This chapter describes the client- and server-side configurations for Web services security. These security configurations are stored in XML files. On the server, this configuration is stored in the `oracle-webservices.xml` deployment descriptor file. On a client, it is stored in the `<generated_name>_Stub.xml` deployment descriptor file.

You can create these files and configure the security elements in any of the following ways:

- JDeveloper IDE—Use the JDeveloper IDE to create and configure Web service and client security. This tool provides wizards that help you create a security configuration. It stores the results in an `oracle-webservices.xml` deployment descriptor for the server and a `<generated_name>_Stub.xml` deployment descriptor for the client.
- Application Server Control—Use Application Server Control to configure security for all of the operations in a particular Web services port, or configure specific security settings for individual operations in the port. For more information, see the topic *Configuring Security for a Web Service* in the Application Server Control on-line help.
- WebServicesAssembler—Use this command line tool to configure and assemble the security elements in the Web service and client.
- Manual configuration—To manually configure security, enter values for the security elements directly in the server and client deployment descriptor files.

The Web services security configuration for the client and server is based on inbound and outbound policies. These policies are defined in "Security Policies" on page 1-3.

Figure 2–1 illustrates the deployment descriptors that are generated when you configure Web service security and how information is passed to the client and service. The following steps correspond to the numbers in the figure.

1. The security configuration for a Web service client. Configuring security for a client will generate the `<generated_name>_Stub.xml` deployment descriptor with a security configuration. The client interceptor uses the `<generated_name>_Stub.xml` deployment descriptor at runtime to generate the security header (if an `<outbound>` element is configured) and enforce the security policy (if an `<inbound>` element is configured).
2. The security configuration for a Web service (server side). Configuring security for the Web service application will generate an `oracle-webservices.xml` deployment descriptor with the security configuration.
3. When you deploy the Web service, the `wsmgmt.xml` file will be updated with the Web service security configured in the `oracle-webservices.xml` deployment

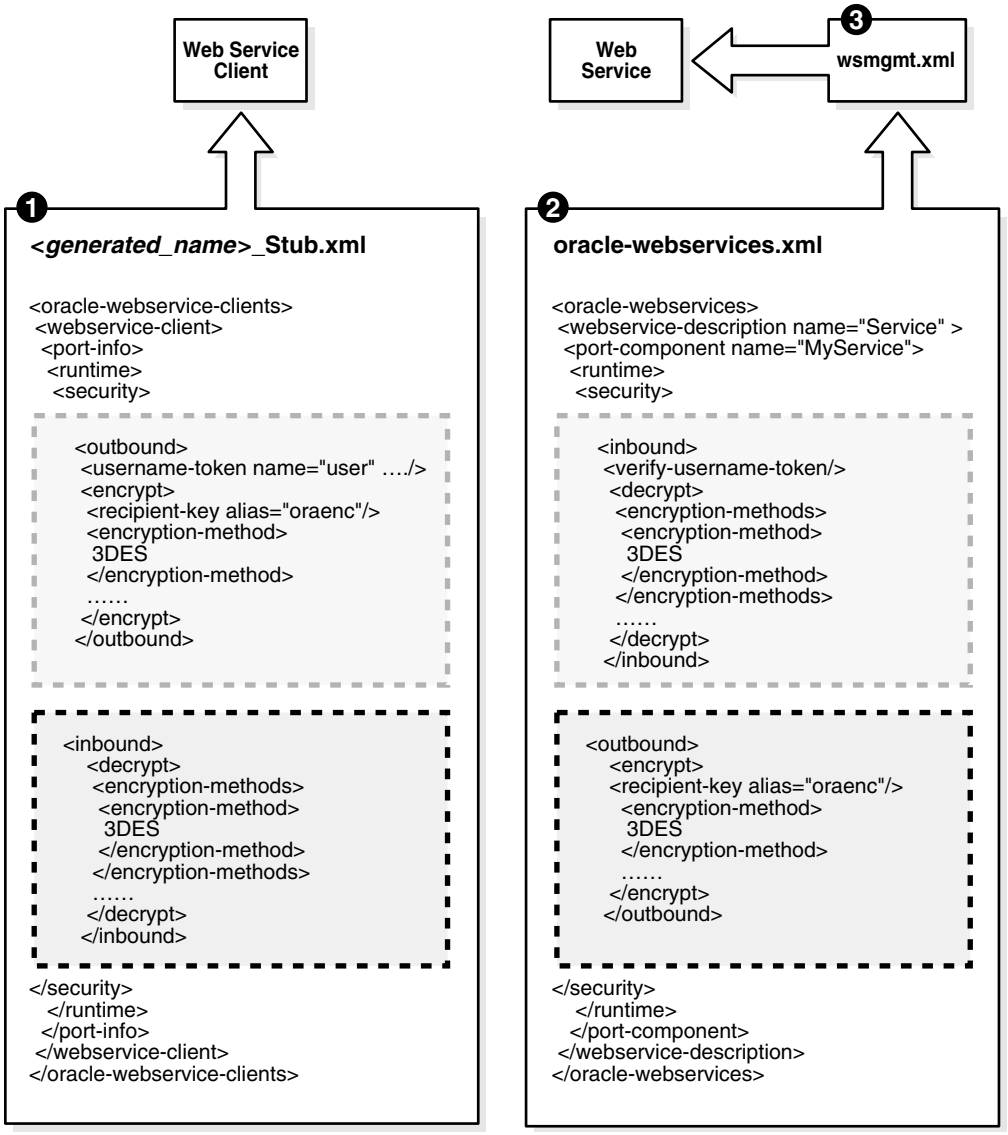
descriptor as described in Step 2. The `wsmgmt.xml` file is an instance-level configuration file, which holds the entire security configuration for the Web services deployed in an OC4J instance. The server interceptor uses the `wsmgmt.xml` file at runtime to enforce the security policy (if an inbound element is configured) or generate the security header (if an outbound element is configured).

Note: Oracle Application Server Web Services Security recommends that you do not directly edit the `wsmgmt.xml` file.

In the diagram, the dashed boxes indicate that for a given `<outbound>` element in the Web service client configuration, there is a corresponding `<inbound>` element in the Web service configuration. Similarly for a given `<outbound>` element in the Web service there is a corresponding `<inbound>` element in the client configuration file.

For example in the client `<outbound>` element there is a `<username-token>` element that indicates that client will be sending the username token. In the Web service `<inbound>` element there is corresponding `<verify-username-token>` element which indicates that the Web service must verify the username token.

Figure 2–1 How Policies Work Together for Client- and Server-side Web Services



Security Configuration Elements

This section provides definitions of the security configuration elements. The same set of security elements can appear at the global, port, and operation level. The values for security elements set at the operation level override settings made at the port and global levels. Port-level settings override global-level settings.

In general, OracleAS Web Services Security elements appear in the header or body of SOAP messages. The client uses the same set of security elements as the server. Example 2–1 illustrates the security elements as they are used in the server-side `oracle-webservices.xml` deployment descriptor.

Example 2–1 Security Elements in the Server-Side Configuration File

```
<oracle-webservices xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/oracle-webs
ervices-10_0.xsd">
 <webservice-description name="XYZService ">
<port-component name="XYZPort">
  <runtime enabled="security">
    <security>
      <key-store store-pass="abc" path="./mykeystore.jks"/>
      <signature-key alias="signkey" key-pass="signkeypass"/>
      <encryption-key alias="enckey" key-pass="enckeypass"/>
    </security>
  </runtime>

  <operations>
    <operation name="sayHello" input="{http://tempuri.org/}/sayHello">
      <runtime>
        <security>
          <inbound>
            <verify-username-token />
            <verify-x509-token />
            <verify-saml-token/>
            <verify-signature>
              <signature-methods>
                <signature-method>
                  RSA-SHA1
                </signature-method>
              </signature-methods>
              <tbs-elements>
                <element name-space="http://schemas.xmlsoap.org/soap/envelope"
                   local-part="Body"/>
              </tbs-elements>
              <verify-timestamp expiry="28800" created="true"/>
            </verify-signature>
            <decrypt>
              <encryption-methods>
                <encryption-method>
                  3DES
                </encryption-method>
              </encryption-methods>
              <keytransport-methods>
                <keytransport-method>
                  RSA-1_5
                </keytransport-method>
              </keytransport-methods>
              <tbe-elements>
                <element name-space="http://schemas.xmlsoap.org/soap/envelope"
                      local-part="Body" mode="CONTENT"/>
              </tbe-elements>
            </decrypt>
          </inbound>

          <outbound>
            <signature>
              <tbs-elements>
                <tbs-element
name-space="http://schemas.xmlsoap.org/soap/envelope/" local-part="Body"/>
              </tbs-elements>
            </signature>
            <encrypt>
              <recipient-key alias="enckey"/>
```

```
              <tbe-elements>
                <tbe-element
name-space="http://schemas.xmlsoap.org/soap/envelope/" local-part="Body" />
              </tbe-elements>
            </encrypt>
          </outbound>
        </security>
      </runtime>
    </operation>
  </operations>
</port-component>
</webservice-description>
</oracle-webservices>
```

Example 2–2 illustrates the security elements as they are used in the client-side *<generated_name>*_Stub.xml deployment descriptor.

Example 2–2 Security Elements in the Client-Side Configuration File

```
<oracle-webservice-clients>
<webservice-client>
<port-info>
<runtime enabled="security">
 <key-store path="mykeystore.jks" store-pass="password"/>
          <signature-key alias="signkey" key-pass="signkeypass"/>
 <security>
  <outbound>
   <username-token name="SCOTT" password="TIGER"/>
<signature>
            <tbs-elements>
             <tbs-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
            </tbs-elements>
            <add-timestamp created="true" expiry="28800"/>
           </signature>
           <encrypt>
            <recipient-key alias="reckey"/>
            <tbe-elements>
             <tbe-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
            </tbe-elements>
           </encrypt>
  </outbound>
 </security>
</runtime>
<operations>
 <operation name="sayHello"/>
</operations>
</port-info>
</webservice-client>
</oracle-webservice-clients>
```

Keystore Elements

The <key-store> element is required and can occur at both global and port levels on the server, and at port level on the client. A global keystore setting applies to all applications deployed within the instance; it can be overridden by a specific port-level keystore. If you make any changes to the value of the <key-store> element, then you must restart the application to enable the new values.

Table 2–1 Keystore Settings

Element Name	Description
<key-store>	Identifies the path to the keystore. This element can appear at global or port level. This element has these attributes: ■ `path`—File system path to the keystore. This can be an absolute or relative path. On server, the relative path is relative to the application root directory: `J2EE_HOME/applications/application_name`. On the client, the path can be either absolute or relative to `J2EE_HOME` when the client application is deployed on OracleAS Web Services. ■ `type`—Keystore type. The default type is `JKS`. Other supported types are `PKCS12` and `ORACLE_WALLET`. ■ `store-pass`—Password to access the keystore.

Signature and Encryption Key Elements

The `<signature-key>` and `<encryption-key>` are required at port level if a port level keystore is specified or when selecting keys from global keystore. If these keys are not configured at the port level, then the global-level values are used.

If you make any changes to the values of the `<signature-key>` or `<encryption-key>` elements, then you must restart the application to enable the new values.

Table 2–2 General Security Settings

Element Name	Description
<signature-key>	Points to the key required by `<signature>` and `<verify-signature>`. This element has these attributes: ■ `alias`—Alias for the key. ■ `key-pass`—Password to access the key.
<encryption-key>	Points to the key required for decrypting the message. This element has these attributes: ■ `alias`—Alias for the key. ■ `key-pass`—Password to access the key. The `<encryption-key>` element is configured as part of the `alias` attribute of the `<recipient-key>` subelement of the `<encrypt>` element. For more information on the `alias` attribute and `<recipient-key>`, see Table 2–18, " Subelements of the <encrypt> Element" on page 2-17.

Nonce Configuration Elements

A nonce is a random value that can be included in the username token to prevent replay attacks. The nonce is cached by the server. OracleAS Web Services Security lets you configure a nonce value that can be inserted into the username token. For more information on configuring the nonce, see "Configure the Nonce Cache with a Digest Password" on page 3-8.

Table 2–3 Nonce Configuration Settings

Element Name	Description
<nonce-config>	The `nonce-config` element allows you to configure the cache. ■ `clock-skew`—The amount of clock skew, in seconds, that is allowed between the client and server if the creation time is included in the username token. The default is 300 seconds. ■ `cache-ttl` (cache time to live)—Indicates how long the nonces remain valid in the cache. Expired nonces are removed from the cache. Default is 300 seconds.

Security Elements for Inbound Messages

The following sections describe the security elements that can be set for inbound messages.

- Username Token Elements for Inbound Messages
- X.509 Token Elements for Inbound Messages
- SAML Token Elements for Inbound Messages
- Signature Verification Elements for Inbound Messages
- Decryption Elements for Inbound Messages

The inbound message section in the `oracle-webservices.xml` and `<generated_name>_Stub.xml` deployment descriptors are delimited with `<inbound>` elements. The `<inbound>` element encapsulates the security configuration policy with respect to incoming messages. The `<inbound>` element can occur as a subelement of `<security>` at the global, port, and operation level.

Inbound security defines the context-specific security policy for the incoming messages. In the case of a client, it corresponds to the security policy associated with receiving a response. In the case of a service, it corresponds to the security policy associated with receiving a request.

The `<verify-username-token>`, `<verify-x509-token>`, and `<verify-saml-token>` elements are the authentication elements for inbound messages. Authentication elements are optional and can be configured for the server side. The Web service application can choose to allow a username token, an X.509 token, and a SAML token, in any combination.

A Web service client is not required to send an authentication token. If an authentication token is required, only one can be sent. The user will be authenticated based on the token(s) sent in the SOAP request.

Table 2–4 summarizes the security tokens that can be inserted into an inbound message configuration.

Table 2–4 Authentication Elements for Inbound Messages

Element Name	Description
<verify-username-token>	Specifies the security policy for username tokens. See "Username Token Elements for Inbound Messages" for more information on this security token.
<verify-x509-token>	Specifies the authentication policy with respect to X.509 tokens. See "X.509 Token Elements for Inbound Messages" for more information on this security token.

Table 2–4 (Cont.) Authentication Elements for Inbound Messages

Element Name	Description
<verify-saml-token>	Specifies whether the incoming SOAP message carrying a SAML assertion should be verified. See "SAML Token Elements for Inbound Messages" on page 2-9 for more information on this security token.

Username Token Elements for Inbound Messages

The `<verify-username-token>` element specifies the security policy for username tokens. This is an optional subelement of the `<inbound>` element and can occur only once within the element. This subelement has these attributes:

- `password-type`—Type of password authentication: `plaintext` or `digest`. Default is `plaintext`.
- `require-nonce`—Specifies whether a nonce must be included in the username token. This attribute is required for digest authentication. Default is `false`.
- `require-created`—Specifies whether the creation time must be included in the username token. This attribute can be used with either plain text or digest password authentication. However, it is required for digest authentication. Default is `false`.

X.509 Token Elements for Inbound Messages

The `<verify-x509-token>` element specifies the authentication policy with respect to X.509 tokens. It is an optional subelement of the `<inbound>` element.

SAML Token Elements for Inbound Messages

The `<verify-saml-token>` element is an optional subelement of the `<inbound>` element. It specifies whether the incoming SOAP message carrying a SAML assertion should be verified.

Table 2–5 lists the subelements of the `<verify-saml-token>` element. All of the subelements are optional.

Table 2–5 Subelements of the <verify saml-token> Element

Element Name	Description
<subject-confirmation-method>	This is an optional element for inbound policy. When used as part of the inbound `<verify-saml-token>` policy, it refers to the confirmation method used for propagating the identity in the incoming SOAP message.
<confirmation-method>	This is an optional element for the `<subject-confirmation-method>`. The possible values for `<confirmation-method>` are: ▪ `Sender-Vouches`—(default) The incoming SAML token must supply a sender-vouches confirmation method and the reference to the token must be signed. ▪ `Sender-Vouches-Unsigned`—The incoming SAML token must supply a sender-vouches confirmation method and the token must not be signed. ▪ `Holder-Of-Key`—The incoming SAML token must supply a holder-of-key confirmation method. The assertion must have the public key of the user.

Signature Verification Elements for Inbound Messages

The `<verify-signature>` element is an optional subelement of the `<inbound>` element. It specifies the integrity or signature requirements of the receiver. These requirements include the name of the signature verification algorithm and the message parts to be verified. The `<verify-signature>` element occurs only once within the `<inbound>` element. Table 2–6 lists the subelements of the `<verify-signature>` element. All of the subelements are optional.

Table 2–6 Subelements of the <verify-signature> Element

Element Name	Description
<signature-methods>	Collection of `<signature-method>` elements. The `<signature-method>` is used to specify the acceptable signing algorithms. Algorithm names are specified using their short names instead of URIs. The default value is `RSA-SHA1`. Table 2–17, " Signature Algorithms and Short Names" on page 2-16 lists the algorithm URIs and corresponding short names that are recognized by Web service security.
<tbs-elements>	List of elements that are expected to be signed in the incoming request. This element has these attributes: ■ `local-part`—The actual element name. ■ `name-space`—The actual name space of the element in the SOAP message. This attribute can be omitted if there is only one element with this name in the namespace.
<verify-timestamp>	Verifies the timestamp in the incoming SOAP message. (This timestamp is configured with the `<add-timestamp>` element described in Table 2–16). The `created` attribute is used to indicate whether a timestamp was created for the message. Incoming SOAP messages with a timestamp that has expired are rejected by the server. ■ `expiry`—Expiration time, in seconds, on the signature. Default is 28800 seconds. ■ `created`—Indicates whether the timestamp includes the creation time. Default is `true`.

<property>	Properties that can be set on the `<verify-signature>` element. The `<property>` subelement has this format:
	`<property name="property_name" value="property_value"/>`
	OracleAS Web Services Security defines the following property on `<verify-signature>`.
	■ `clock-skew`—Configures the clock difference between the client and the server. The client that is sending the SOAP message (signing and adding a timestamp) and the Web service application (receiving the SOAP message and verifying the signature and the timestamp) may be running on two separate machines. If the clocks on the machines are not in sync, then `clock-skew` is configured to sync-up the time between them.
	The default value of `clock-skew` is `0` and the units are measured in milliseconds. The following example sets the clock skew to three seconds.
	`<property name="clock-skew" value="3000"/>`
	This property can be set in either the `oracle-webservices.xml` or `<generated_name>_Stub.xml` deployment descriptor. There is no tool support for adding this property; you must manually edit the files.

Decryption Elements for Inbound Messages

The <decrypt> element is an optional subelement of the <inbound> element. It specifies the confidentiality requirements of the receiver. The <decrypt> element occurs only once within an <inbound> element.

Table 2–7 describes the elements that are available to set decryption details for inbound messages. All of the subelements are optional.

Table 2–7 Subelements of the <decrypt> Element

Element Name	Description
<encryption-methods>	Encryption methods as part of decrypt element refer to the encryption methods accepted by the Web service application. A Web service application can accept multiple encryption methods. If the incoming SOAP message is encrypted the server interceptor checks the confidentiality policy and rejects the request if the encryption algorithms do not match. The valid options are: ■ 3DES ■ AES-128 (default) ■ AES-256 Table 2–8 lists the encryption algorithm URIs and corresponding short names recognized by Web services security.
<keytransport-methods>	Collection of <keytransport-method> elements. Each <keytransport-method> specifies the acceptable key transport algorithms. Multiple <keytransport-method> elements can be specified for inbound messages. Algorithm names are specified using their short names instead of URIs. Acceptable values are: ■ RSA-1_5 (default) ■ RSA-OAEP-MGF1P Table 2–9 lists the algorithm URIs and corresponding short names recognized by Web services security.
<tbe-elements>	Indicates the elements that are encrypted in the incoming SOAP message. ■ name-space—The actual name space of the element in the SOAP message. This attribute can be omitted if there is only one element with this name in the namespace. ■ local-part—The actual element name. ■ mode—An additional attribute that is used to specify whether the ELEMENT or the CONTENT is expected to be encrypted. If mode is ELEMENT then the entire element is expected to be encrypted, if mode is CONTENT then the content of the element is expected to be encrypted. Default is CONTENT.

Table 2–7 (Cont.) Subelements of the <decrypt> Element

Element Name	Description

<property>	Properties that can be set on the `<decrypt>` element. The `<property>` subelement has this format:
	`<property name="property_name" value="property_value"/>`
	OracleAS Web Services Security defines the following property on `<decrypt>`.
	▪ `oracle.security.wss.decryptusingski`
	When set to `true`, the subject key identifier in the encrypted key tag is resolved to a private key in the keystore. By default, this property is set to `false`. For example:
	`<property name="oracle.security.wss.decryptusingski" value="true"/>`
	Note: This property can be set in either the `oracle-webservices.xml` or `<generated_name>_Stub.xml` deployment descriptor. There is no tool support for adding this property; you must manually edit the files.
	The decryption key alias and password is stored in the `system-jazn-data.xml` file using password indirection. See "Replacing Cleartext Passwords by Using Password Indirection" on page 3-6 for more information on the password indirection mechanism.

Table 2–8 lists the URIs of the encryption algorithms recognized by Web service security and their short names.

Table 2–8 URIs and Short Names for Encryption Algorithms

URI of the Algorithm	Short Name
http://www.w3.org/2001/04/xmlenc#3des-cbc	3DES
http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES-128 (default)
http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES-256

Table 2–9 lists the URIs of the key transport algorithms recognized by Web service security and their short names.

Table 2–9 URIs and Short Names for Key Transport Algorithms

URI of the Algorithm	Short Name
http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA-1_5 (default)
http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	RSA-OAEP-MGF1P

Security Elements for Outbound Messages

The following sections describe the security elements that can be set for outbound messages.

- Username Token Elements for Outbound Messages
- X.509 Token Elements for Outbound Messages
- SAML Token Elements for Outbound Messages

- Elements for Retrieving SAML Tokens from an External SAML Authority
- Signature Elements for Outbound Messages
- Encryption Elements for Outbound Messages

The outbound message section in the `oracle-webservices.xml` and `<generated_name>_Stub.xml` deployment descriptors are delimited with `<outbound>` elements. Outbound security defines the context-specific security policy for the outgoing messages. In the case of a client, it corresponds to the security policy associated with sending a request. In the case of a service, it corresponds to the security policy associated with sending a response.

Example 2–10 summarizes the security tokens that can be inserted into an outbound message configuration.

Table 2–10 Outbound Security Tokens

Element Name	Description
<username-token>	Specifies a username token that must be inserted into the security header block. See "Username Token Elements for Outbound Messages" on page 2-12 for more information on this security token.
<x509-token>	If this element is present, an X.509 certificate is inserted into the security header block. See "X.509 Token Elements for Outbound Messages" on page 2-13.
<saml-token>	The client interceptor uses the contents of this element to create the SAML assertion for the user identity. See "SAML Token Elements for Outbound Messages" on page 2-14 for more information on this security token.
<signature>	Specifies the algorithm for signing outgoing messages or individual message elements. See "Signature Elements for Outbound Messages" on page 2-16 for more information on this security token.
<encryption>	Specifies the encryption algorithm, key transport algorithm, and recipient key for encrypting outgoing messages or message elements. See "Encryption Elements for Outbound Messages" on page 2-17 for more information on this security token.

Username Token Elements for Outbound Messages

The `<username-token>` element is an optional element of the outbound policy. This element specifies the username token that must be inserted into the security header block. Only one instance of the element is permitted.

A client can pass a username and password by using a callback handler. The callback handler is a user-defined class that handles `javax.security.auth.Namecallback` and `Passwordcallback`. The name of the class can be specified in the `<username-token>` element's `cbhandler-name` attribute.

If this element is not present, the client runtime can build the username token using the `Stub.USERNAME_PROPERTY` and `Stub.PASSWORD_PROPERTY` supported by JAX-RPC. The Web service client can either use the `Stub` properties or use a static configuration by specifying the user name and password in a deployment descriptor. For more information on using the `Stub` properties, see "Pass the User Name and Password with Stub Properties" on page 3-11.

Table 2–11 describes the `<username-token>` attributes. All of the attributes are optional.

Table 2–11 Attributes of the <username-token> Element

Attribute Name	Description
name	The username to be inserted into the token.
password	The actual password of the user.
password-type	Type of password: `plaintext` or `digest`. Default is `plaintext`. Note that if you set `password-type` to `digest`, then `add-nonce` and `add-created` will be set to `true` by default.
cbhandler-name	The name of the callback handler that inserts the username token into the SOAP message. The callback handler is a user-defined callback handler class that handles `NameCallback` and `PasswordCallback`. The value of the `cbhandler-name` attribute is the name of the user-defined implementation class.
add-nonce	Specifies whether a nonce should be added to the request. For digest authentication, this attribute is required and must be set to `true`. This attribute is optional for plain text password authentication. The default value is `false`.
add-created	Specifies whether a creation time should be added to the request. For digest password authentication, this attribute is required and must be set to `true`. This attribute is optional for plain text password authentication. The default value is `false`.

X.509 Token Elements for Outbound Messages

The `<x509-token>` element is an optional element of the `<outbound>` configuration. This element indicates that an X.509 signing certificate will be inserted into the request. A direct reference to the X.509 certificate (signer's certificate) is added. You must have the signature key configured for this configuration to work.

Instead of passing a certificate (using direct reference), you could also pass the subject key identifier of the certificate. "Using the Subject Key Identifier for Signing" on page 3-40 provides more information on how to pass the subject key identifier.

Table 2–12 describes the subelement of the `<x509-token>` element.

Table 2–12 Subelement of the <x509-token> Element

Element Name	Description

<property>	Properties that can be set on the `<x509-token>` element. The `<property>` subelement has this format:
	`<property name="property_name" value="property_value"/>`
	OracleAS Web Services Security defines the following property on `<x509-token>`.
	▪ `oracle.security.wss.signX509token`—This property is applicable only when the `<x509-token>` is used with signature `<signature>`. If set to `true` (default), the Binary Security Token (BST) that contains the X.509 token will be signed by default. If set to `false`, the Binary Security Token will not be signed. For example:
	`<property name="oracle.security.wss.signX509token" value="false"/>`

SAML Token Elements for Outbound Messages

The `<saml-token>` element is an optional element of the `<outbound>` policy. The client interceptor refers to the `<saml-token>` element in the outbound policy for creating the actual SAML assertion for the user identity. "How to Configure a SAML Token for the Client-Side" on page 3-24 provides information on how to provide dynamic and static configuration of SAML tokens. Table 2–13 describes the attributes of the `<saml-token>` element. All of the attributes are optional.

Table 2–13 Attributes of the <saml-token> Element

Attribute	Description
name	You can choose a name for the assertion subject by providing a value for the `name` attribute of the `<saml-token>` element. The `name` attribute has the following format: [*realm-name/*]*name* The *name* represents the name of the assertion. The assertion name can be prefixed with the assertion's *realm-name*. If the realm name is already present, then it is set as the name qualifier. The `name` attribute contains the actual name of the user identity that is being propagated. For example, `name="jdoe"`. The value of the `name` attribute is inserted in the `<name-identifier>` element of the SAML assertion. The default name identifier format is `UNSPECIFIED`.
name-format	Specifies the format of the assertion subject name. This element can have any of the following values: ■ `UNSPECIFIED` (default)— can be any value. ■ `EMAIL`—an email address, such as `abc@myCompany.com`. ■ `X509-SUBJECT-NAME`—an X.509 subject name (an X.509 subject name translates to `DN`, a distinguished name). For example: `CN="abc", OU="Security", O="Oracle", C="US"`. ■ `WINDOWS-DOMAIN-NAME`—the name of a Windows domain. For example: `abc`.
cbhandler-name	Identifies the name of the user-defined class that will handle the `SAMLTokenCallback` call back handler. This class is used to pass SAML assertions to the interceptor. The callback handler must be able to handle SAML token callback. "Writing a SAML Token Callback Handler" on page 3-28 provides more information on how to write this handler.
issuer-name	Used to get the SAML assertion issuer name. The default value is `www.oracle.com`. It is strongly recommended that you change this to the name of your own assertion issuer.

Table 2–14 describes the subelements of the `<saml-token>` element. All of the subelements are optional.

Table 2–14 Subelements of the <saml-token> Element

Element Name	Description
<confirmation-method>	The supported confirmation methods are `Sender-Vouches` (default), `Sender-Vouches-Unsigned`, and `Holder-Of-Key`. "Configuring Confirmation Methods" on page 3-26 provides more information on confirmation methods and on how to configure this element.
<attribute>	The `<attribute>` element has a mandatory `path` attribute that points to a properties file. The attribute statement is created from the attributes listed in this file. This properties file contains one or more attribute name/value pairs for asserting a user's identity. The attribute name can be prefixed with an optional namespace. For example: [*attribute-name-space/*]*attribute-name=value* The following is an example of a value that can appear in an `<attribute>` subelement. `email=abc@myCompany.com` For more information on using the `<attribute>` element, see "Configuring Authentication and Attributes Statements" on page 3-26.
<saml-authority>	Allows you to retrieve a SAML token from an external SAML authority. For more information, see "Elements for Retrieving SAML Tokens from an External SAML Authority".

Elements for Retrieving SAML Tokens from an External SAML Authority

The `<saml-authority>` element is an optional subelement `<saml-token>`. A configuration of the `<saml-authority>` element and its subelements allow you to retrieve a SAML token from an external SAML authority by issuing a SAMLP request. "Retrieving a SAML Token from an External SAML Authority" on page 3-29 provides more information on configuring this element. Table 2–15 describes the subelements of `<saml-authority>`.

Table 2–15 Subelements of the <saml-authority> Element

Name	Description
<endpoint-address>	(Required) Specifies the SAML Responder URL.
<auth-user-name>	Specifies the username that is used to provide authentication to the SAML authority. This attribute is required for the `Holder-Of-Key` confirmation method, optional otherwise. For the `Holder-Of-Key` subject confirmation method, the SAML assertion token is requested for the user identified by `auth-user-name`. For the `Sender-Vouches` subject confirmation method, the SAML assertion token is requested for the user identified by the `name` attribute of `<saml-token>` element.
<auth-password>	(Optional) Specifies the password that is used to provide authentication to the SAML authority. The `auth-user-name` and `auth-password` elements are required if `password-based-mechanism` is used for authentication.

Table 2–15 (Cont.) Subelements of the <saml-authority> Element

Name	Description
<require-signature>	(Optional) If this boolean attribute is `true`, then the SAMLP request is signed with the client's signature key. In addition, the client-side keystore and signature keys should be configured. The default value for this element is `false`. See "Keystore Elements" on page 2-5 and "Signature and Encryption Key Elements" on page 2-6 for more information on configuring the keystore and the signature key.

Signature Elements for Outbound Messages

Table 2–16 describes the subelements of the `<signature>` element. The subelements describe the options that are available for signing outbound messages.

Table 2–16 Subelements of the <signature> Element

Element Name	Description
<signature-methods>	Collection of `<signature-method>` elements. The `<signature-method>` element specifies the acceptable signature algorithm. Algorithm names are specified using their short names instead of URIs. The default value is `RSA-SHA1`. Table 2–17, " Signature Algorithms and Short Names" on page 2-16 lists the algorithm URIs and corresponding short names that are recognized by OracleAS Web Services Security.
<tbs-elements>	List of elements to be signed. This element has these attributes: ▪ `name-space`—The actual name space of the element in the SOAP message. This attribute can be omitted if there is only one element with this name in the namespace. This attribute can be omitted if all of the elements in the SOAP message share the same name space. ▪ `local-part`—The actual element name.
<add-timestamp>	Adds a timestamp to the outbound SOAP message. (This timestamp is verified by setting the `<verify-timestamp>` element described in Table 2–6 on page 2-9). ▪ `expiry`—Expiration time, in seconds, until the signature expires. Default is 28800 seconds. ▪ `created`—Indicates whether a creation time is inserted into the timestamp. Default is `true`.

Table 2–17 lists the signature algorithms recognized by Web service security and their associated short names.

Table 2–17 Signature Algorithms and Short Names

Signature Algorithm	Short Name
http://www.w3.org/2000/09/xmldsig#rsa-sha1	RSA-SHA1 (default)
http://www.w3.org/2001/04/xmldsig-more#rsa-md5	RSA-MD5
http://www.w3.org/2000/09/xmldsig#dsa-sha1	DSA-SHA1

Encryption Elements for Outbound Messages

The <encrypt> element is an optional subelement of the <outbound> element. It specifies confidentiality requirements of the sender. The <encrypt> element can occur only once within an <outbound> element.

Table 2–18 describes the encryption elements that are available for outbound messages.

Table 2–18 Subelements of the <encrypt> Element

Element Name	Description
<encryption-method>	Specifies the encryption method to be used for encrypting the elements of the outbound SOAP message. Only one encryption method can be listed under the <encrypt> element. The following methods are valid. ■ 3DES ■ AES-128 (default) ■ AES-256 Table 2–8 lists the URIs and corresponding short names for the encryption algorithms recognized by Web services security.
<keytransport-method>	A <keytransport-method> identifies the key transport algorithm. Only one keytransport method can be specified for outbound messages. Algorithm names are specified using their short names instead of URIs. The possible values for this element are: ■ RSA-1_5 (default) ■ RSA-OAEP-MGF1P Table 2–9 lists the URIs and corresponding short names of the algorithms recognized by Web services security.
<tbe-elements>	Elements to be encrypted. This element has these attributes: ■ name-space—The actual name space of the element in the SOAP message. This attribute can be omitted if there is only one element with this name in the namespace. ■ local-part—The actual name of the element. ■ mode—An additional attribute that is used to specify whether the ELEMENT or the CONTENT is encrypted. If the value of mode is ELEMENT then the entire element is encrypted, if the value of mode is CONTENT then the content of the element is encrypted. Default is CONTENT.
<recipient-key>	The key alias of the recipient, which is used to encrypt the data encryption key. The data encryption key is the generated symmetric key that is used to encrypt the actual data. The data encryption key itself is also encrypted using the recipient's public key. The recipient key may or may not have a key usage extension. If the recipient key does have a key usage extension, then it must be of the type KEY_ENCIPHERMENT. If it does not, the encryption request is rejected. This element has these attributes: ■ alias—An alias for the key. ■ key-pass—An optional password attribute to access the key.

Table 2–18 (Cont.) Subelements of the <encrypt> Element

Element Name	Description
<use-request-cert>	The Web service client has sent a signed SOAP message and the Web service application has successfully verified the signature. When the Web service application sends a response back to the same client, it can choose to encrypt the response with the signature certificate that the client sent in the first message exchange. The `use-request-cert` element is configured as part of a Web service application's outbound encryption policy. Note that if the server interceptor is unable to find the signature certificate (that is, the client has not sent a signed SOAP message or the signature verification failed) then the Web service application will reject the encryption request.

3

Administering Web Services Security

This chapter describes how to configure message-level security for SOAP messages. This includes configuring keystores, security tokens, signature, and encryption. The terminology used in this chapter, is described in "Web Service Security Concepts" on page 1-2.

This chapter contains the following sections.

Using Keystores

This section describes how to create and configure a keystore that can be used with Web service security. It also describes how you can enhance the protection of your keystore, signature key, and encryption key passwords by employing password indirection.

Note: This section assumes that you are familiar with public key infrastructure concepts such as trust stores, certificates, and public and private keys. If you are not, the following books and Web sites provide an overview of these topics.

Oracle Application Server Certificate Authority Administrator's Guide

http://download-east.oracle.com/docs/cd/B14099_15/idmanage.htm

Oracle Security Developer Tools (OSDT) Reference

http://download-east.oracle.com/docs/cd/B14099_15/idmanage.htm

Java Security Overview

http://java.sun.com/developer/technicalArticles/Security/whitepaper/index.html

J2SE Security and the Java Platform

http://java.sun.com/security/index.jsp

Creating a Keystore

A keystore is a file that provides information about available public and private keys. Keys are used for a variety of purposes, including authentication and data integrity. For example:

- to sign data, you must have the signer's private key
- to verify a signature, you must have a trusted CA certificate and the public key that matches the private key
- to encrypt data, you must have the recipient's public key
- to decrypt data, you must have the private key which corresponds to the public key

These trusted certificates and public and private keys are stored in the keystore. Oracle Application Server Web Services Security supports a variety of keystores, including Oracle Wallet, PKCS12, and Sun Microsystem's Java Key Store (JKS) format. The following sections describe where you can obtain trusted certificates and how to create and use these keystores.

- How to Obtain a Trusted Certificate
- How to Create and Use a Java Key Store
- How to Create and Use an Oracle Wallet

How to Obtain a Trusted Certificate

You can obtain a certificate from a Certificate Authority (CA), such as Verisign or Entrust, and include them in the keystore. To get the certificate, you must create a Certificate Request and submit it to the CA. The CA will authenticate the certificate requestor and create a digital certificate based on the request. You must then include the certificate in the wallet.

How to Create and Use a Java Key Store

The Java Keystore (JKS) is the proprietary keystore format defined by Sun Microsystems. To create and manage the keys and certificates in the JKS, use the `keytool` utility. You can use the `keytool` utility to perform the following tasks:

- create public and private key pairs, designate public keys belonging to other parties as trusted, and manage your keystore.
- issue certificate requests to the appropriate Certification Authority (CA), and import the certificates which they return.
- administer your own public and private key pairs and associated certificates. This allows you to use your own keys and certificates to authenticate yourself to other users and services. This process is known as "self-authentication". You can also use your own keys and certificates for data integrity and authentication services, using digital signatures.
- cache the public keys of your communicating peers. The keys are cached in the form of certificates.

How to Create Private Keys and Load Trusted Certificates The following section provides an outline of how to create and manage the JKS with the `keytool` utility. It describes how to create a keystore and to load private keys and trusted CA certificates. You can find more detailed information on the commands and arguments for the `keytool` utility at this Web address.

http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html

1. Create a new private key and self-signed certificate.

   Use the `genKey` command to create a private key. It will create a new private key if one does not exist. The following command generates an `RSA` key, with `RSA-SHA1` as the signature algorithm, with the alias `test` in the `test.jks` keystore.

   ```
   keytool -genkey -alias test -keyalg "RSA" -sigalg
   "SHA1withRSA" -dname "CN=test, C=US" -keypass test123
   -keystore test.jks -storepass test123
   ```

   By default, if you omit the algorithm to generate the key pair (`keyalg`), then the `genKey` command generates a `DSA` key. If you omit the signature algorithm (`sigalg`) argument, then the signature algorithm is derived based on the value of `keyalg`.

   Make sure to use the correct values for the `keyalg` and `sigalg` arguments, based on your signature configuration. If the underlying algorithm to generate the key pair (`keyalg`) is `DSA`, then the default signature algorithm (`sigalg`) `SHA1withDSA` is used. If the underlying `keyalg` is `RSA`, then the default `sigalg` `RSAwithMD5` is used.

2. Display the keystore.

   The following command displays the contents of the keystore. It will prompt you for the keystore password.

   ```
   keytool -list -v -keystore test.jks
   ```

3. Import a trusted CA certificate in the keystore.

Use the `-import` command to import the certificate. The following command imports a trusted CA certificate into the `test.jks` keystore. It will create a new keystore if one does not exist.

```
keytool -import -alias aliasfortrustedcacert -trustcacerts
-file trustedcafilename -keystore test.jks -storepass test123
```

4. Generate a certificate request.

   Use the `-certreq` command to generate the request. The following command generates a certificate request for the `test` alias. The CA will return a certificate or a certificate chain.

   ```
   keytool -certreq -alias test -sigalg "RSAwithSHA1" -file
   certreq_file -keypass test123 -storetype jks -keystore
   test.jks -storepass test123
   ```

5. Replace the self-signed certificate with the trusted CA certificate.

   You must replace the existing self-signed certificate with the certificate from the CA. To do this, use the `-import` command. The following command replaces the trusted CA certificate in the `test.jks` keystore.

   ```
   keytool -import -alias test -file trustedcafilename -keystore
   test.jks -storepass test123
   ```

How to Create and Use an Oracle Wallet

The Oracle Wallet acts as a keystore for storing and managing public and private keys, and X.509 certificates. To create a wallet, Oracle provides the `orapki` utility in the *`ORACLE_HOME`*`/bin` directory.

This section provides an outline of how to create and manage the Oracle Wallet with the `orapki` utility. You can find more information on the Oracle Wallet at this Web address.

`http://www.oracle.com/technology/products/oid/oidhtml/sec_idm_training/html_masters/basics06.htm`

You can find more information on using the `orapki` utility for creating and managing the Oracle Wallet in the *Oracle Database Advanced Security Administrator's Guide*.

- "How to Create an Oracle Wallet with Self-Signed Certificates" describes how to create an Oracle Wallet, add a self-signed certificate, and export it.
- "How to Create an Oracle Wallet and User Certificates" describes how to work with certificate requests and replies. Steps 1 to 3 describe how to create and export a certificate request.

How to Create an Oracle Wallet with Self-Signed Certificates The following steps illustrate creating an Oracle Wallet with a self-signed certificate, viewing the wallet, and exporting the certificate.

1. Create a root Oracle Wallet.

   ```
   orapki wallet create -wallet wallet_dir wallet_dir
   ```

   This command creates the root Oracle Wallet in the `wallet_dir` directory. By default, the wallet will be named `ewallet.p12`.

2. Add a root certificate to the Oracle Wallet.

   ```
   orapki wallet add -wallet wallet_dir -dn 'CN=root_test,C=US'
   -keysize 2048 -self_signed -validity 3650
   ```

This command adds a self-signed (root) certificate to the wallet. It creates a self-signed certificate with a validity of 3650 days. The distinguished name of the subject is `CN=root_test,C=US`. The key size for the certificate is 2048 bits.

3. Export the self-signed certificate from the Oracle Wallet.

```
orapki wallet export -wallet wallet_dir -dn 'CN=root_
test,C=US' -cert b64certificate.txt
```

This command exports the self-signed certificate to the `b64certificate.txt` file. Note that the distinguished name used is the same as in the previous step.

4. View the contents of the root Oracle Wallet.

```
orapki wallet display -wallet wallet_dir
```

The `display` command allows you to view the contents of the Oracle Wallet.

How to Create an Oracle Wallet and User Certificates The following steps illustrate creating an Oracle Wallet, adding and exporting a certificate request, and then importing the received certificate.

1. Create an Oracle Wallet with auto login enabled.

```
orapki wallet create -wallet server -auto_login
```

This command creates an Oracle Wallet at `/private/user/server` with auto login enabled.

2. Add a certificate request to the Oracle Wallet.

```
orapki wallet add -wallet server -dn 'CN=server_test,C=US'
-keysize 2048
```

This command adds a certificate request to the wallet that was created. The distinguished name of the subject is `CN=server_test,C=US`. The specified key size is 2048 bits.

3. Export a certificate request from the Oracle Wallet.

```
orapki wallet export -wallet server -dn 'CN=server_test,
C=US' -request creq.txt
```

This command exports the certificate request to the specified file, `creq.txt`. Note that the order of the distinguished name is reversed from above (when the certificate request was created).

4. Create a signed certificate from the certificate request for testing purposes.

```
orapki cert create -wallet wallet_dir -request creq.txt -cert
cert.txt -validity 3650
```

This command creates a certificate, `cert.txt` with a validity of 3650 days. The certificate is created from the certificate request generated in the preceding step.

5. View a certificate.

```
orapki cert display -cert cert.txt -complete
```

This command displays the certificate generated in the preceding step. The `-complete` option allows you to display additional certificate information, including the serial number and public key.

6. Add a trusted certificate to the Oracle wallet.

```
orapki wallet add -wallet server -trusted_cert -cert
b64certificate.txt
```

This command adds a trusted certificate, `b64certificate.txt` to the wallet. You must add all trusted certificates in the certificate chain of a user certificate before adding a user certificate.

7. Add a user certificate to the Oracle wallet.

   ```
   orapki wallet add -wallet server -user_cert -cert cert.txt
   ```

   This command adds the user certificate, `cert.txt` to the wallet.

8. View the contents of the Oracle wallet.

   ```
   orapki wallet display -wallet server
   ```

   The display command allows you to view the contents of the Oracle Wallet.

Configuring a Keystore

Keystore, signature, and encryption keys can be configured at global or port level. Oracle Web Service security implementation can be configured to use JKS or PKCS12 or Oracle Wallet.

- Configuring Instance Keystores and Keys
- Configuring Application Keystores and Keys

Configuring Instance Keystores and Keys

A global keystore and key setting apply to all Web service applications that are deployed in a particular OC4J instance. "Keystore Elements" on page 2-5 provides more information about the global keystore. To configure a instance level keystore and keys use Application Server Control tool. For more information, see the topic *Viewing or Modifying the Instance Keystore and Identity Certificates* in the Application Server Control on-line help.

Configuring Application Keystores and Keys

You can use a keystore and keys for an Oracle Web service application which are separate from those belonging to the instance. An application-specific keystore and key setting takes precedence over instance-specific settings.

To configure an application-specific keystore and keys, add the `<key-store>`, `<signature-key>`, and `<encryption-key>` elements under the `<security>` element in the `oracle-webservices.xml` file for the server and `<generated_name>_Stub.xml` for the client.

Example 3–1 illustrates a keystore configuration for a Web service application. The name of the keystore is `mykeystore.jks` and the password to access it is `abc`.

Example 3–1 Configuration for a Web Service Application Keystore

```
<security>
     <key-store store-pass="abc" path="./mykeystore.jks"/>
     <signature-key alias="signkey" key-pass="signkeypass"/>
     <encryption-key alias="enckey" key-pass="enckeypass"/>
</security>
```

Replacing Cleartext Passwords by Using Password Indirection

Many OC4J components, such as the keystore, signature key, and encryption key, require passwords for authentication. Embedding these passwords into deployment descriptors, such as `oracle-webservice.xml`, poses a security risk, especially if the

permissions on the files allow them to be read by any user. To avoid this problem, you can employ password indirection. Password indirection replaces cleartext passwords with information necessary to look up the password in another location.

Use Application Server Control administration console *Instance/Application Keystore* settings screen to indirect the passwords for keystores and keys. This action automatically adds a user entry with an indirect username and password to the instance-specific *ORACLE_HOME*/j2ee/*instance*/config/system-jazn-data.xml.

See the *Oracle Containers for J2EE Security Guide* for more information on creating and using indirect passwords.

Integrating Security Tokens with Security Providers

With Web Services Security, you can employ a username, X.509, or SAML token as your security element. They can integrate with many of the security providers defined under Oracle Containers for J2EE. See "Web Service Security Integration" on page 1-13 for more information on how Web service security integrates with the OracleAS Web Services Security framework.

Table 3–1 lists the security providers that are supported for the username, X.509, and SAML tokens.

Table 3–1 Security Tokens and Providers Supported by Web Services Security

Token Name	File-Based Security Provider	Oracle Identity Management (versions 10.1.2.0.x and 9.0.4.3)	External LDAP Security Provider	Custom Security Provider	COREid Security Provider (version 7.0.4)
Username token, Plaintext password	YES	YES	YES	YES	YES
Username token, Digest password	YES	NO	NO	NO	NO
X.509 token	YES	YES	YES	NO	YES
SAML token	YES	YES	YES, but must configure manually	NO	YES

Using a Username Token

This section describes how to configure the username token for the server and for the client. It also describes how to integrate the token with OC4J security providers.

- How to Configure the Username Token for the Server Side
- How to Configure the Username Token for the Client Side
- Integrating Username Token with Security Providers (XML, LDAP, Custom, COREid)
- Preventing Replay Attacks with Nonces

How to Configure the Username Token for the Server Side

For the server, you configure the user name token in the `oracle-webservices.xml` deployment descriptor. Configuring the username token for the server consists of the following steps:

1. Configure the <verify-username-token> Element
2. Configure the Nonce Cache with a Digest Password (Optional)

   This step is required for digest password only.

Configure the <verify-username-token> Element

To configure a user name token for the server side, use the `<verify-username-token>` element. This element can occur only once as a subelement of the `<inbound>` element. Attributes of the `<verify-username-token>` element let you configure details about the password, and whether a nonce or a creation time should be included in the token.

Details about the password are specified in the `password-type` attribute. The password can be designated as either `plaintext` or `digest`. `Plaintext` indicates that the token will expect a password that is in unencrypted, human-readable form. `Digest` indicates that a cryptographic checksum value has been calculated for the password that is sent with the token. The purpose of the value is to ensure that the password has not been tampered with. The recipient of the password, in this case the server, recomputes the cryptographic checksum and compares it with the cryptographic checksum passed with the password; if they match, it is probable that the password was not tampered with during transmission.

If you choose to send a digest password, then you must choose whether a nonce or a creation time should be included in the username token. The `require-nonce` attribute specifies whether a nonce must be included in the username token. A nonce is a random value that can be included in the username token to prevent replay attacks. The server caches this value in order to compare it with the nonce included with succeeding requests. Using a nonce ensures that the application generates a different password digest at every invocation; two nonce digests will differ even if they have identical password and time created values. "Preventing Replay Attacks with Nonces" on page 3-14 provides more information on how nonces can be used to help prevent replay attacks.

The `require-created` attribute specifies whether the token's creation time must be included in the username token. This can also be used to prevent replay attacks.

The `<verify-username-token>` and its attributes are listed in Table 2–4 on page 2-7. Example 3–2 illustrates a sample configuration.

Example 3–2 Username Token Configuration for the Server Side

```
...
<inbound>
  <verify-username-token/>
 </inbound>
...
```

Configure the Nonce Cache with a Digest Password

The nonce cache can be statically configured in the `javacache.xml` file (*ORACLE_HOME*`/j2ee/home/config/javacache.xml`). The OracleAS Web Services runtime uses this file to configure the Java Object Cache at runtime. You can configure the

nonce cache by editing the values for the `<max-size>` and `<max-objects>` elements in this file.

The `<max-size>` element specifies the maximum size of the memory, in megabytes, available to the Java Object Cache. The `<max-objects>` element specifies the maximum number of in-memory objects that are allowed in the cache. The count does not include group objects, or objects that have been spooled to disk and are not currently in memory.

"Preventing Replay Attacks with Nonces" on page 3-14 provides more information on how nonces can be used to help prevent replay attacks. For more information on the `javacache.xml` file, see "Java Object Cache" in the *Oracle Containers for J2EE Services Guide*.

Example 3–3 illustrates a sample nonce cache configuration in the `javacache.xml` file. The nonce cache is configured to hold a maximum of 2000 objects and grow up to 512 kilobytes in size.

Example 3–3 Sample Nonce Cache Configuration (javacache.xml)

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <max-objects>
     20000
  </max-objects>
  <max-size>
     512
  </max-size>
</cache-configuration>
```

Tools for Configuring the Username Token for the Server

Application Server Control—You can configure the username token in the *Inbound Policies Authentication Page*. For more information, see the context-sensitive help for this page in the Application Server Control online help.

JDeveloper—Authentication is set on the Authentication page of the Secure Web Services wizard or the Web Services Editor. For more information, see the topic *Setting Authentication for Web Services* in the JDeveloper on-line help.

WebServicesAssembler tool—You can use this tool to assemble a security configuration into your Web service. To do this, include a configuration for the username token in the `oracle-webservices.xml` deployment descriptor. Use `ddFileName` argument to specify this file as input to the appropriate Web service assembly command. For more information on the `ddFileName` argument and assembling Web services, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

How to Configure the Username Token for the Client Side

For the client, you configure the username token in the `<generated_name>_Stub.xml` deployment descriptor. Configuring the username token for the client consists of the following steps:

1. Configure the <username-token> Element
2. Write a Callback Handler (Optional)
3. Pass the User Name and Password with Stub Properties (Optional)

Configure the <username-token> Element

The `<username-token>` element specifies the username token that must be inserted into the security header block. Providing appropriate values for the `name` and `password` attributes allow the message to access the service.

Adding a nonce with the `add-nonce` attribute and a request creation time with the `add-created` attribute to the token can provide additional security. If you choose a digest password, then these attributes must be present and set to `true`. These attributes are optional for plain text password. Example 3–4 illustrates a sample configuration for the server.

Example 3–4 Username Token Configuration for the Client Side

```
...
<outbound>
  <username-token name="SCOTT" password="TIGER"/>
</outbound>
...
```

Write a Callback Handler

A callback handler is a `javax.security.auth.callback.CallbackHandler` instance that allows a login module to interact with a user to obtain login information. Callback handlers are defined in the Java Authentication and Authorization Service (JAAS) package which enables applications to authenticate and enforce access controls upon users.

There are three types of callback handlers, each represented by a class in the `javax.security.auth.callback` package: a name callback handler (`NameCallback`) to handle a user name, a password callback handler (`PasswordCallback`) to handle a password, and a text input callback handler (`TextInputCallback`) to handle any field in a login form other than a user name or password field.

To dynamically pass the username and password to the service, the client can use either a callback handler or a stub property. The callback handler is a user-defined class that handles `javax.security.auth.Namecallback` and `Passwordcallback`. For more information on callback handlers, see the *Oracle Containers for J2EE Security Guide*.

The `<username-token>` element's `cbhandler-name` attribute specifies the name of the handler class.

Example 3–5 illustrates the `cbhandler-name` attribute configured to use the `oracle.ws.wssecurity.usernametoken.UsernameTokenCallbackHandler` callback handler.

Example 3–5 Configuration for the cbhandler-name Attribute

```
<outbound>
    <username-token
cbhandler-name="oracle.ws.wssecurity.usernametoken.UsernameTokenCallbackHandler"/>
  </outbound>
```

Example 3–6 illustrates a fragment of the `UsernameTokenCallbackHandler` callback handler implementation.

Example 3–6 Callback Handler for s Username Token

```
...
```

```
package oracle.ws.wssecurity.usernametoken;

...

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.TextOutputCallback;

...

public class UsernameTokenCallbackHandler implements CallbackHandler {

    Callback c = null;

    public void handle(Callback[] callbacks) {
        try {
            if (callbacks == null) {
                return;
            }

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

                if (c instanceof NameCallback) {
                                        NameCallback nc = (NameCallback)c;
                                        nc.setName("name");


                } else if (callbacks[i] instanceof PasswordCallback) {
                                        PasswordCallback pc =
(PasswordCallback)callbacks[i];
                                        pc.setPassword("password".toCharArray());
                }
                                                        }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Pass the User Name and Password with Stub Properties

If you do not include the `<username-token>` attributes `name` and `password` in your configuration, then the client runtime can dynamically build the username token using the `Stub.USERNAME_PROPERTY` and `Stub.PASSWORD_PROPERTY` properties supported by JAX-RPC. For more information on `Stub` properties, see "Username Token Elements for Outbound Messages" on page 2-12.

Example 3–7 illustrates client code that uses `Stub.USERNAME_PROPERTY` and `Stub.PASSWORD_PROPERTY` to call the Web service with the username `oc4jadmin` and password `welcome`.

Example 3–7 Using Stub Properties to Access a Web Service

```
...
public void UserNameWithSystemProperties() throws Exception {
     // Call the service with Stub property username/password
     ((Stub) stub)._setProperty(Stub.USERNAME_PROPERTY, "oc4jadmin");
```

```
     ((Stub) stub)._setProperty(Stub.PASSWORD_PROPERTY, "welcome");
String response = stub.helloUser(MESSAGE);
System.out.println("Response: ["+response+"]");
...
```

Tools for Configuring the Username Token for the Client

JDeveloper—Authentication is set on the Authentication page of the Secure Web Services wizard or the Web Services Editor. For more information, see the topic *Setting Authentication for Web Services* in the JDeveloper on-line help.

WebServicesAssembler tool—You can use this tool to assemble a security configuration into your web service client proxy. To do this, include a configuration for the username token in the `<generated_name>_Stub.xml` deployment descriptor. Use `ddFileName` argument to specify this file as input to the WebServicesAssembler `genProxy` command. For more information on the `ddFileName` argument and assembling Web service clients, see "Assembling a J2SE Web Service Client" in the *Oracle Application Server Web Services Developer's Guide*.

Integrating Username Token with Security Providers (XML, LDAP, Custom, COREid)

Table 3–1 on page 3-7 provides a summary of the security providers that can be used with the username token. All of the security Providers that are available for the username token can be configured by using Application Server Control.

You should configure OC4J with security providers before you configure Web service security. Table 3–2 indicates where you can find additional information on configuring the security providers that are available for the username token.

Table 3–2 Information on Security Providers that can be Used with the Username Token

For this Security Provider type	See this resource for more information
File-based Security Provider	For information on how to configure a file based provider for the username security token, see the *Oracle Containers for J2EE Security Guide*.
Oracle Identity Management	For information on how to configure Oracle Identity Management, including Oracle Internet Directory (OID), for the username security token, see the *Oracle Containers for J2EE Security Guide*. Single sign-on is not available for the username token.
External LDAP Providers	For information on how to configure external LDAP providers for the username security token, see the *Oracle Containers for J2EE Security Guide*.
Custom Security Providers	For information on how to configure custom providers for the username security token, see the *Oracle Containers for J2EE Security Guide*.
COREid	For information on how to configure COREid as a security provider for the username security token, see "Using COREid as a Security Provider for Username Token Authentication" on page 3-12. For more information on COREid, see the *Oracle Containers for J2EE Security Guide*.

Using COREid as a Security Provider for Username Token Authentication

If you have COREid access system as part of your deployment, then you can use COREid as a security provider to integrate with the COREid access system. The COREid security provider uses the `CoreIDLoginModule` login module, supplied by

Oracle. The `CoreIDLoginModule` is defined in the *`ORACLE_HOME`*`/j2ee/`*`instance`*`/config/system-jazn-data.xml` file.

The login module is configured on a "per application" basis.

On the client, the username token uses the username and password for authentication. Consequently, the Web Service must define validation for the username and password using the COREid login module. Once you define the authentication scheme for username and password verification, you must define these variable values in the login module by using the `coreid.name.attribute` and `coreid.password.attribute` properties.

To authenticate the name, set the value of the `coreid.name.attribute` property to the variable name defined for verifying the user name in the `credential_mapping` plug-in. This property is set as an option under the `<login-module>` element.

To authenticate the password, set the value of the `coreid.password.attribute` property to the variable name defined for verifying the password in the `validate_password` plug-in.

For more information on these properties and the `CoreIDLoginModule`, see the *Oracle Containers for J2EE Security Guide*.

Example 3–8 illustrates setting the `coreid.name.attribute` property to authenticate the user name for the username token. The `username_variable` value is a variable name used to authenticate the user name in the `credential_mapping` plug-in.

Example 3–8 Setting the User Name Variable for COREid in the Username Login Module

```
...
<login-module>
...
   <option>
      <name>coreid.name.attribute</name>
      <value>username_variable</value>
   </option>
...
</login-module>
...
```

Example 3–9 illustrates setting the `coreid.password.attribute` property to authenticate the password for the username token. The `username_password` value is a variable name used to authenticate the password in the `validate_password` plug-in.

Example 3–9 Setting the Password Variable for COREid in the Username Login Module

```
...
<login-module>
...
   <option>
      <name>coreid.password.attribute</name>
      <value>username_password</value>
   </option>

...
</login-module>
...
```

Example 3–10 illustrates a sample `CoreIDLoginModule` configuration for a username token in the `system-jazn-data.xml` file. The value of the `<name>` element, *`application_name`*, represents the name of the Web service application.

Example 3–10 CoreIDLoginModule for Username Token Authentication

```
<application>
   <name>application_name</name>
   <login-modules>
      <login-module>
      <class>
        oracle.security.jazn.login.module.coreid.CoreIDLoginModule
      </class>
      <control-flag>required</control-flag>
      <options>
         <option>
            <name>addAllRoles</name>
            <value>true</value>
         </option>
         <option>
            <name>coreid.resource.type</name>
            <value>res_type</value>
         </option>
         <option>
            <name>coreid.resource.operation</name>
            <value>res_operation</value>
         </option>
         <option>
            <name>coreid.resource.name</name>
            <value>/res_name</value>
         </option>
         <option>
            <name>coreid.name.attribute</name>
            <value>username_variable</value>
         </option>
         <option>
            <name>coreid.password.attribute</name>
            <value>username_password</value>
         </option>
      </options>
      </login-module>
   </login-modules>
</application>
```

Preventing Replay Attacks with Nonces

If you are using the username token for authentication, then you can employ a nonce to help prevent message replay attacks. A message replay attack occurs when an intermediate listener (a "man in the middle") captures an authenticated and validated message. The intermediate party can replay the captured message over and over again unless there is some means for client and service to uniquely identify a message.

A nonce is a unique random number that is included in the message that helps to uniquely identify the message to the client and server. For example, an application can use a nonce when creating a password digest, as follows:

```
password_digest = SHA1 (nonce + created + password)
```

Using a nonce ensures that the application generates a different password digest at every invocation; two nonced digests will differ even if they have identical password and time created values.

A nonce can be added to the username token whether the password type is digest or plain text. To add a nonce, include the `add-nonce="true"` and `add-created="true"` attributes to the `<username-token>` element. For plain text password type, the `add-nonce` and `add-created` are optional. If you specify the password type as `digest`, then the `add-nonce` and `add-created` attributes are required and will be set to `true` by default. For example:

```
<username-token name="jdoe" password="password"
password-type="DIGEST"/>
```

Using an X.509 Token

This section describes how to configure the X.509 token for the server and the client. It also describes how to integrate the token with OC4J security providers.

> **Note:** OracleAS Web Services Security handles X.509 (#X509V3) simple certificate processing only. It does not support certificate path (#X509PK1PathV1) or Set of Certificates and CRLs (#PKCS7).

- How to Configure an X.509 Token for the Server Side
- How to Configure X.509 Token for the Client Side
- Integrating X.509 Token with Security Providers (XML, LDAP, COREid)

How to Configure an X.509 Token for the Server Side

For the server, you configure the X.509 token in the `oracle-webservices.xml` deployment descriptor. Configuring the X.509 token for the server consists of the following steps:

1. Configure the <verify-x509-token> Element
2. Configure the Keystore
3. Map the X.509 Certificates to Valid Users (Optional)

Configure the <verify-x509-token> Element

To configure an X.509 token for the server, use the `<verify-x509-token>` element. The `<verify-x509-token>` element is a subelement of the `<inbound>` element. Example 3–11 illustrates a sample X.509 token configuration for the server.

Example 3–11 X.509 Token Configuration for the Server Side

```
    ...
    <inbound>
      <verify-x509-token/>
    <inbound>
...
```

Configure the Keystore

Create a keystore to store the private keys, public keys, and certificates required by the X.509 token. You can use either a Java Keystore (JKS) or Oracle Wallet. For the steps involved in configuring either of these keystores, see "Using Keystores" on page 3-1.

Map the X.509 Certificates to Valid Users

The mapping attribute (`mapping.attribute`) maps an X.509 certificate to a valid user in an XML or LDAP repository. The LDAP repository can be either an Oracle Internet Directory (OID) or an external repository such as Active Directory or iPlanet. The default mapping attribute uses `DN` or the distinguished name of the X.509 certificate to map the certificate to a user in the LDAP realm.

For an XML provider (`jazn-data.xml`) the `CN` or common-name is used by default. The realm will be based on the security provider configuration.

You can customize the mapping by setting `mapping.attribute` in the bootstrap `jazn.xml` file (*`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/jazn.xml`).

Example 3–12 illustrates using the `mapping.attribute` attribute to map a user in the LDAP repository to an X.509 certificate based on an `email` address. The example assumes that the default `DN` will be used to map the certificate and that the `DN` has the following definition:

```
DN = "CN=jdoe, OU=security, O=Oracle, email="jdoe@oracle.com""
```

The certificate will be mapped to the user with the e-mail address `jdoe@oracle.com`.

Example 3–12 Mapping a User to an X.509 Certificate

```
<jazn provider="LDAP" location=".... ">
<property name="mapping.attribute" value="email"/>
</jazn>
```

If multiple users are found to match the mapping attribute, then the request is rejected. The value of the `mapping.attribute` attribute cannot be changed by using Application Server Control.

Table 3–3 describes the default values for the mapping attribute based on the security provider type.

Table 3–3 Values for mapping.attribute Based on Security Provider Types

Default Value	Security Provider Type
CN	File-based Security provider.
DN	Oracle Identity Management security provider, external LDAP Provider and COREid

`CN` should be the default only for XML providers.

Tools for Configuring the X.509 Token on the Server

Application Server Control—You can configure the X.509 token in the *Inbound Policies Authentication Page*. For more information, see the context-sensitive help for this page in the Application Server Control online help.

JDeveloper—Authentication is set on the Authentication page of the Secure Web Services wizard or the Web Services Editor. For more information, see the topic *Setting Authentication for Web Services* in the JDeveloper on-line help.

WebServicesAssembler tool—You can use this tool to assemble a security configuration into your Web service. To do this, include a configuration for the X.509 token in the `oracle-webservices.xml` deployment descriptor. Use the `ddFileName` argument to specify this file as input to the appropriate Web service assembly command. For more information on the `ddFileName` argument and assembling Web Services, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

How to Configure X.509 Token for the Client Side

For the client, you configure the X.509 token in the `<generated_name>_Stub.xml` deployment descriptor. Configuring the X.509 token for the client consists of the following steps:

1. Configure the <x509-token> Element
2. Configure the Keystore with a Signature Key
3. Authenticate an X.509 Token with a Subject Key Identifier (Optional)
4. Sign the X.509 Token (Optional)

Configure the <x509-token> Element

To configure the client for the X.509 token use the `<x509-token>` element. Example 3–13 illustrates a sample configuration.

Example 3–13 X.509 Token Configuration for the Client Side

```
...
  <outbound>
    <x509-token/>
  <outbound>
...
```

Configure the Keystore with a Signature Key

Create a keystore to store the signature key required by the X.509 token on the client. You can use either a Java Keystore (JKS) or Oracle Wallet to provide this. For the steps involved in obtaining a signature key and for configuring either of these keystores, see "Using Keystores" on page 3-1.

Authenticate an X.509 Token with a Subject Key Identifier

For X.509 token authentication, if the receiver has prior knowledge of the X.509 certificate used for authentication, then the subject key identifier can be sent instead of the entire certificate. The subject key identifier is an extension of the certificate which is used to calculate the public key. If your certificate does not have the subject key identifier, then the request will be rejected.

To authenticate with a subject key identifier, make the following changes to the `<generated_name>_Stub.xml` file.

1. Set the `<property>` subelement of the `<signature>` element to the subject key identifier property `oracle.security.wss.signwithski`.
2. Set the `value` attribute of the `<property>` subelement to `true`. By default, this property is set to `false`.

Example 3–14 illustrates how to use the subject key identifier property `oracle.security.wss.signwithski` to authenticate an X.509 token.

Example 3–14 X.509 Token with a Subject Key Identifier

```
<x509-token/>
<signature>
     <property name="oracle.security.wss.signwithski" value="true"/>
  </signature>
```

> **Note:** Configure the signature key in the security configuration to point to the signer's private key. The signer's certificate must have the `SubjectKeyIdentifierExtension`. Also the receiver's keystore must contain the signer's certificate to resolve the subject key identifier.

Sign the X.509 Token

An unsigned X.509 token is not secure and can be replaced with any other token. You can secure the X.509 token by signing it in outbound messages. To do this, OracleAS Web Services Security defines the Boolean `oracle.security.wss.signX509token` property on the `<x509-token>`.

This property is applied only when the `<x509-token>` is used with signature `<signature>`. If set to `true` (default), the X.509 token will be signed by default. If set to `false`, the X.509 token will not be signed.

For more information on this property, see "X.509 Token Elements for Outbound Messages" on page 2-13.

Example 3–15 illustrates how the `oracle.security.wss.signX509token` property is used in conjunction with the `<x509-token>` and `<signature>` elements.

Example 3–15 Signing the X.509 Token in an Outbound Message

```
...
<x509-token>
   <property name="oracle.security.wss.signX509token" value="false"/>
</x509-token>
<signature>
   ...
</signature>
...
```

Tools for Configuring the X.509 Token on the Client

JDeveloper—Authentication is set on the Authentication page of the Secure Web Services wizard or the Web Services Editor. For more information, see the topic *Setting Authentication for Web Services* in the JDeveloper on-line help.

WebServicesAssembler tool—You can use this tool to assemble a security configuration into your Web service client proxy. To do this, include a configuration for the X.509 token in the `<generated_name>_Stub.xml` deployment descriptor. Use the `ddFileName` argument to specify this file as input to the WebServicesAssembler `genProxy` command. For more information on the `ddFileName` argument and assembling Web service clients, see "Using WebServicesAssembler" in the *Oracle Application Server Web Services Developer's Guide*.

Integrating X.509 Token with Security Providers (XML, LDAP, COREid)

Table 3–1 on page 3-7 provides a summary of the security providers that can be used with the X.509 security token. All of the security Providers that are available for the X.509 token can be configured by using Application Server Control.

You should configure OC4J with security providers before you configure Web service security. Table 3–4 indicates where you can find additional information on configuring the security providers that are available for the X.509 token.

Table 3–4 Information on Security Providers that can be Used with the X.509 Token

For this Security Provider type	See this resource for more information
File-Based Security Provider	For information on how to configure a file-based provider for the X.509 security token, see the *Oracle Containers for J2EE Security Guide*.
Oracle Identity Management	For information on how to configure Oracle Identity Management, including Oracle Internet Directory (OID) and Single Sign-on, for the X.509 security token, see the *Oracle Containers for J2EE Security Guide*.
External LDAP Providers	For information on how to configure external LDAP providers for the X.509 security token, see the *Oracle Containers for J2EE Security Guide*.
COREid	For information on how to configure COREid as a security provider for the X.509 security token, see "Using COREid as a Security Provider for X.509 Token Authentication". For more information on COREid, see the *Oracle Containers for J2EE Security Guide*.

Using COREid as a Security Provider for X.509 Token Authentication

If you have COREid access system as part of your deployment, then you can use COREid as a security provider to integrate with the COREid access system. The COREid security provider uses the `CoreIDLoginModule` login module supplied by Oracle. The `CoreIDLoginModule` is defined in the *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/system-jazn-data.xml` file.

The login module is configured on a "per application" basis.

To use COREid with X.509 authentication, configure the COREid login module to use the mapping attribute for the authentication plug-in. Set the `coreid.name.attribute` property to the value of the `CN` variable defined for verifying the mapping attribute of in the `credential_mapping` plug-in. This property is set as an option under the `<login-module>` element.

For more information on this property and the `CoreIDLoginModule`, see the *Oracle Containers for J2EE Security Guide*.

Example 3–16 illustrates setting the `coreid.name.attribute` property in the `CoreIDLoginModule` to authenticate the mapping for the X.509 token. The `cn_variable` value is a variable name used to authenticate the mapping.

Example 3–16 Setting the CN Variable for the COREid Login Module for X.509 Authentication

```
...
<login-module>
...
   <option>
      <name>coreid.name.attribute</name>
```

```
      <value>cn_variable</value>
   </option>
...
</login-module>
...
```

Example 3–17 illustrates a `CoreIDLoginModule` for X.509 token authentication in the `system-jazn-data.xml` file. The value of the `<name>` element, *`application_name`*, represents the name of the Web service application.

Example 3–17 CoreIDLoginModule for X.509 Token Authentication

```
<application>
   <name>application_name</name>
   <login-modules>
      <login-module>
      <class>
       oracle.security.jazn.login.module.coreid.CoreIDLoginModule
      </class>
      <control-flag>required</control-flag>
      <options>
         <option>
            <name>addAllRoles</name>
            <value>true</value>
         </option>
         <option>
            <name>coreid.resource.type</name>
            <value>res_type</value>
         </option>
         <option>
            <name>coreid.resource.operation</name>
            <value>res_operation</value>
         </option>
         <option>
            <name>coreid.resource.name</name>
            <value>/res_name</value>
         </option>
         <option>
            <name>coreid.name.attribute</name>
            <value>cn_variable</value>
         </option>
      </options>
      </login-module>
   </login-modules>
</application>
```

Using a SAML Token

This section describes how to configure the SAML token for the server and the client. It also describes how to integrate the token with OC4J security providers. For information on the SAML token use case, see "SAML Token Profile" on page 5-7.

- How to Configure a SAML Token for the Server Side
- How to Configure a SAML Token for the Client-Side
- Integrating SAML Token with Security Providers (XML, LDAP, COREid)

How to Configure a SAML Token for the Server Side

There are three possible usecases for configuring the SAML token on the server, based on which confirmation method you want to employ. These confirmation methods are sender vouches (unsigned), sender vouches (signed), and holder of key.

- sender-vouches (unsigned)—the incoming SAML token must supply a sender-vouches confirmation method and the token must not be signed.
- sender-vouches (signed)—the incoming SAML token must supply a sender-vouches confirmation method and the reference to the token that must be signed.
- holder of key—the incoming SAML token must supply a holder of key confirmation method. The assertion must contain the public key of the user.

The following sections list the steps to configure the SAML token for each of these usecases.

Sender-vouches (unsigned)

1. Configure the <verify-saml-token> Element
2. Map the SAML Assertion Subject (Optional)
3. Set Options for the SAMLLoginModule

   The issuer name is set to `www.oracle.com` by default. It is strongly recommended that you change this value to the name of your own assertion issuer.

Sender-vouches (signed)

1. Configure the <verify-saml-token> Element
2. Configure the Keystore
3. Set Options for the SAMLLoginModule

   The issuer name is set to `www.oracle.com` by default. It is strongly recommended that you change this value to the name of your own assertion issuer.
4. Map the SAML Assertion Subject (Optional)

Holder of key

1. Configure the <verify-saml-token> Element
2. Configure the Keystore
3. Set Options for the SAMLLoginModule

   For the holder-of-key confirmation method, the issuer name, trust point alias, and all of the keystore options in the SAMLLoginModule must be set.

Configure the <verify-saml-token> Element

To configure a SAML token for the server, use the `<verify-saml-token>` element. The `<verify-saml-token>` element is a subelement of the `<inbound>` element.

The `<verify-saml-token>` element can have an optional `<subject-confirmation-method>` subelement. When used as part of an inbound `<verify-saml-token>` policy, this subelement refers to the confirmation method used to propagate the identity in the incoming SOAP message.

The actual confirmation method is specified by the `<confirmation-method>` element. This is a subelement of the `<subject-confirmation-method>` element. Possible values for `<confirmation-method>` are `SENDER-VOUCHES`, `SENDER-VOUCHES-UNSIGNED`, and `HOLDER-OF-KEY`.

Example 3–18 provides a sample configuration which uses the `<verify-saml-token>` element to verify a SAML token. This configuration will accept a SAML token that contains the user's public key. The token itself can be either signed, unsigned, or holder of key.

Example 3–18 Verifying a SAML Token

```
<verify-saml-token>
  <confirmation-methods>
    <confirmation-method>SENDER-VOUCHES</confirmation-method>
    <confirmation-method>SENDER-VOUCHES-UNSIGNED</confirmation-method>
    <confirmation-method>HOLDER-OF-KEY</confirmation-method>
</verify-saml-token>
```

Table 2–4 and Table 2–5 provide more information on the `<verify-saml-token>` element and its subelements.

Configure the Keystore

Create a keystore to store the private keys, public keys and certificates required by the SAML token. For a holder of key confirmation method, the keystore must contain the user's trusted certificate for verifying the signature. Additionally, all of the assertion issuer options in the `SAMLLoginModule` must be configured with Application Server Control. See "Set Options for the SAMLLoginModule" on page 3-23 for more information on the `SAMLLoginModule` options.

For the sender-vouches (signed) confirmation method, the keystore must contain the public key and the trusted certificates. The issuer name option in the SAML login module must also be configured with Application Server Control.

The sender-vouches (unsigned) confirmation method does not require a keystore to be configured.

You can use either a Java Keystore (JKS) or Oracle Wallet as the keystore. For the steps involved in configuring either of these keystores, see "Using Keystores" on page 3-1.

Map the SAML Assertion Subject

By default, `SAMLLoginModule` is configured with Oracle's assertion issuer. If the name identifier format is omitted or unspecified, then you must configure the `mapping.attribute` attribute in the `jazn.xml` (*ORACLE_HOME*`/j2ee/`*instance_name*`/config/jazn.xml`) file. The value of this attribute cannot be changed by using Application Server Control; you must edit the file manually.

If the SAML token has a name identifier format, then one of the mapping attributes described in Table 3–5 is used. A mapping attribute property is not required. "Configuring SAML Assertion Subject Name and Format" on page 3-26 provides a description of the subject name formats.

Table 3–5 Subject Name Identifier Format and Corresponding Mapping Values

Subject Name Identifier Format	Mapping Attribute in OID	Mapping Attribute in Active Directory	Mapping Attribute in iplanet
EMAIL_ADDRESS	mail	mail	mail
X509_SUBJECT_NAME	DN	DN	DN
WINDOWS_DOMAIN_NAME	`OrclADSAMAccountName`	`samaccountname`	not applicable
UNSPECIFIED / OMITTED	`mapping.attribute` in `jazn.xml` file	`mapping.attribute` in `jazn.xml` file	`mapping.attribute` in `jazn.xml` file

Table 3–6 describes the default values for the `mapping.attribute` attribute based on the security provider type.

Table 3–6 Values for mapping.attribute Based on Security Provider Types

Default Property Value	Security Provider Type
CN	File-based Security provider
DN	Oracle Identity Management security provider, external LDAP Provider, and COREid.

`CN` should be the default only for XML provider.

Example 3–19 illustrates using the `mapping.attribute` property to map a SAML assertion subject to a valid user in an XML repository.

Example 3–19 Mapping a SAML Assertion Subject

```
<jazn provider="XML" realm="jazn.com">
  <property name="mapping.attribute" value="CN"/>
<jazn>
```

Set Options for the SAMLLoginModule

For SAML authentication to succeed, you must configure a trusted SAML assertion issuer as part of the `SAMLLoginModule` options. The `SAMLLoginModule` is a system login module that authenticates an incoming SAML token.

The `SAMLLogingModule` provides options for setting the trusted SAML issuer name, trust point alias, keystore path, keystore type, and password for the trusted SAML issuer. By default, the SAML login module is not configured with a keystore.

You can set the trusted SAML issuer name, keystore, and trust point alias options in the *Trusted SAML authority* screen in Application Server Control.

> **Note:** For the holder of key subject confirmation method, the issuer signs the SAML assertion. If you are configuring this method, then you must set the issuer name, trust point alias, and all of the keystore options.

Table 3–7 provides descriptions of the `SAMLLogingModule` options. These options must be configured for each assertion issuer. The variable *N* in the option name represents the number of the assertion issuer.

Table 3–7 SAMLLoginModule Options

SAMLLoginModule Options	Descriptions	Required by this Confirmation Method
<issuer.keystorepassword.*N*>	Occurs: 1...*N* Specifies the keystore password.	Holder-of-Key
<issuer.keystorepath.*N*>	Occurs: 1...*N* Specifies the keystore path where the assertion issuer certificate is stored.	Holder-of-Key
<issuer.keystoretype.*N*>	Occurs: 1...*N* Specifies the assertion issuer keystore type. Default keystore type is `JKS`.	Holder-of-Key
<issuer.name.*N*>	Occurs: 1...*N* Specifies the issuer name. For the OC4J Standalone `SAMLLoginModule`, the default issuer name is set to `www.oracle.com`. It is strongly recommended that you change this value to the name of your own assertion issuer.	Holder-of-Key, Sender Vouches
<issuer.trustpointalias.*N*>	Occurs: 1...*N* Specifies the trust point key alias of the assertion issuer. This setting is used to verify the assertion issuer certificate in Holder-Of-Key subject confirmation cases.	Holder-of-Key

How to Configure a SAML Token for the Client-Side

For the Web service client, the SAML token can be configured for the sender-vouches (unsigned), sender-vouches (signed), or holder of key confirmation methods. The configuration can be either static, by hard coding the `<saml-token>` element values in the `<generated_name>_Stub.xml` file, or set dynamically.

Sender Vouches (unsigned)

1. Configure the <saml-token> Element.

   This configuration can be either static or dynamic. A static configuration involves providing values for SAML elements in the `<generated_name>_Stub.xml` file. For a dynamic configuration, you can use Stub properties, callback handlers, identity propagation, or SAMLP. For more information on each of these configurations, see the following sections:

   - Providing a Static SAML Client Configuration
   - Configuring a SAML Assertion Subject by Using a Stub Property
   - Configuring a SAML Assertion Subject by Identity Propagation
   - Writing a SAML Token Callback Handler
   - Retrieving a SAML Token from an External SAML Authority

Sender vouches (signed)

1. Configure the <saml-token> Element

   The configuration can be either static or dynamic. A static configuration involves providing values for SAML elements in the `<generated_name>_Stub.xml` file. For a dynamic configuration, you can use Stub properties, callback handlers,

identity propagation, or SAMLP. For more information on each of these configurations, see the following sections:

- Providing a Static SAML Client Configuration
- Configuring a SAML Assertion Subject by Using a Stub Property
- Configuring a SAML Assertion Subject by Identity Propagation
- Writing a SAML Token Callback Handler
- Retrieving a SAML Token from an External SAML Authority

2. Configure the Keystore with a signature key and certificates.

Holder of key

1. Configure the <saml-token> Element.

   A client-side SAML token configuration for holder of key cannot be static. It must be performed dynamically, either by using a callback handler or SAMLP.

2. Configure the Keystore with keys and certificates.

Configure the <saml-token> Element

A client-side configuration for SAML must provide the following values.

- SAML assertion subject name
- Assertion subject name format
- Name of the assertion issuer
- Authentication and attribute statements

You can provide these values either statically or dynamically. For a static configuration, you provide values for SAML elements. For a dynamic configuration, you write a SAML token callback handler.

- Providing a Static SAML Client Configuration
- Configuring a SAML Assertion Subject by Using a Stub Property
- Configuring a SAML Assertion Subject by Identity Propagation
- Writing a SAML Token Callback Handler
- Retrieving a SAML Token from an External SAML Authority
- Combining Static and Dynamic SAML Configuration
- Retrieving a SAML Token from an External SAML Authority

Providing a Static SAML Client Configuration

This section describes how to provide the required values for a static SAML configuration. The client configuration for security appears in the `<generated_name>_Stub.xml` file.

- Configuring SAML Assertion Subject Name and Format
- Configuring the Assertion Issuer
- Configuring Authentication and Attributes Statements
- Configuring Confirmation Methods

Configuring SAML Assertion Subject Name and Format You can choose a name for the assertion subject by providing a value for the `name` attribute of `<saml-token>` element. The name can optionally be prefixed with the assertion's realm name. If the realm name is already present, then it is set as the name qualifier. The `name` attribute has the following format:

name = [*realm-name/*]*name*

The format of the assertion subject's name is specified by setting the `name-format` attribute. Table 2–13, " Attributes of the <saml-token> Element" on page 2-14 describes the possible values for this attribute.

Configuring the Assertion Issuer You can configure the assertion issuer name by setting the `issuer-name` attribute. If no value is specified for this attribute, then the default assertion issuer, `www.oracle.com`, is used. It is strongly recommended that you change this value to the name of your own issuer. Table 2–13, " Attributes of the <saml-token> Element" on page 2-14 provides more information on this element.

Configuring Authentication and Attributes Statements Authentication statements are generated by default. If you want to generate attribute statements, add an `attributes.properties` file containing the attributes of the subject.

Use the `path` attribute of the `attributes` element to indicate the location of the `attributes.properties` file.

`<attributes path=<`*location-of-attributes.properties file*`>>`

The `attributes` element can be prefixed with an optional name space value.

[*attribute-name-space/*]*name=value*

Configuring Confirmation Methods This section describes the types of confirmation methods supported by OracleAS Web Services Security.

- Configuring a Sender-Vouches (Signed) Confirmation Method
- Configuring a Sender-Vouches-Unsigned Confirmation Method
- Configuring a Holder-of-key Confirmation Method

The default confirmation method is sender-vouches (implicit signature).

Configuring a Sender-Vouches (Signed) Confirmation Method To configure a sender-vouches (signed) confirmation method, configure the `<saml-token>` and `<signature>` elements. Example 3–20 illustrates the configuration for generating a sender-vouches (signed) assertion token. The `<signature>` element follows the `<saml-token>` element. The `<signature-method>` element identifies the signature algorithm. If no algorithm is specified, then the default, `RSA-SHA1`, is used.

See Table 2–16, " Subelements of the <signature> Element" on page 2-16 for a description of the `<signature-method>` element. Table 2–17, " Signature Algorithms and Short Names" on page 2-16 provides a description of the element's possible values.

Example 3–20 Configuration for a Sender-Vouches (Signed) Confirmation Method

```
<saml-token name="jdoe">
    <subject-confirmation-method>
        <confirmation-method>SENDER-VOUCHES</confirmation-method>
    </subject-confirmation-method>
</saml-token>
<signature>
```

```
     <signature-method>DSA-SHA1</signature-method>
</signature>
```

Configuring a Sender-Vouches-Unsigned Confirmation Method Example 3–21 illustrates the configuration for generating a sender-vouches (unsigned) assertion token. The `<confirmation-method>` element identifies the type of confirmation used for the unsigned assertion. See Table 2–14, " Subelements of the <saml-token> Element" on page 2-15 for a description of the possible values for the `<confirmation-method>` element.

Example 3–21 Configuration for a Sender Vouches (Unsigned) Confirmation Method

```
<saml-token name="jdoe" >
    <subject-confirmation-method>
        <confirmation-method>SENDER-VOUCHES-UNSIGNED</confirmation-method>
    </subject-confirmation-method>
</saml-token>
```

Configuring a Holder-of-key Confirmation Method Example 3–22 illustrates the configuration for sending a holder-of-key assertion using a callback handler. The `cbhandler-name` attribute identifies the user-defined class that will handle the `SAMLTokenCallback` callback handler. In this example, `mypackage.SAMLTokenCallbackHandler` is the user-defined class.

Table 2–14 on page 2-15 provides more information on the `cbhandler-name` attribute. "Writing a SAML Token Callback Handler" on page 3-28 provides more information on creating callback handlers for SAML tokens.

Example 3–22 Configuration for a Holder of Key Confirmation

```
<saml-token cbhandler-name="mypackage.SAMLTokenCallbackHandler">
    <subject-confirmation-method>
        <confirmation-method>HOLDER-OF-KEY</confirmation-method>
    </subject-confirmation-method>
</saml-token>
```

Configuring a SAML Assertion Subject by Using a Stub Property

If you do not include a `<saml-token>` element in your configuration, then the client runtime can dynamically build the token using the `Stub.USERNAME_PROPERTY` supported by JAX-RPC. For more information on using this property, see Table 2–14, " Subelements of the <saml-token> Element" on page 2-15.

In the following example, `Stub` is `javax.xml.rpc.Stub` and `_port` is the stub generated by WebServicesAssembler.

```
((Stub) _port)._setProperty(Stub.USERNAME_PROPERTY, name);
```

> **Note:** If you provide a name value in a static configuration file and dynamically set the username with `USERNAME_PROPERTY`, then the `USERNAME_PROPERTY` value will override the value in the static configuration.

Configuring a SAML Assertion Subject by Identity Propagation

Identity propagation allows an application to propagate an ID to the Web service. The ID information resides in the assertion subject. You can obtain the assertion subject

from the current thread of execution by setting the property `oracle.security.wss.propagate.identity` to `true`. This property can be set only at the port level. For example:

```
<property name="oracle.security.wss.propagate.identity" value ="true"/>
```

If the property is set to `true` and an identity already exists, then the identity is propagated to the Web service. If an identity is not associated with the current thread of execution, then the request is rejected. If this property is set to `false` (default), then the identity is not propagated.

Writing a SAML Token Callback Handler

You can write a SAML token callback handler to deliver the assertion to the client interceptor. The callback handler implementation must be able to handle a `SAMLTokenCallback` object. The `SAMLTokenCallback` object is bi-directional; this means that the name of the subject for whom the assertion is requested can be retrieved by using the callback `getName()` method.

Example 3–23 illustrates a sample SAML token callback handler implementation. The `getName()` method obtains the name of the subject. The `getAssertion(subject)` method gets the assertion based on the value of `subject`. These lines are highlighted in bold.

Example 3–23 Sample Implementation of a SAML Token Callback Handler

```
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import oracle.webservices.security.callback.SAMLTokenCallback;
public class SAMLTokenCallbackHandlerHK implements CallbackHandler {
    Callback c = null;
    public void handle(Callback[] callbacks) {
        try {
            if (callbacks == null) {
                return;
            }
            for (int i = 0; i < callbacks.length; i++) {
                c = callbacks[i];
                if (c instanceof SAMLTokenCallback) {
                    SAMLTokenCallback sc = (SAMLTokenCallback) c;
String subject = sc.getName();
Document doc = getAssertion(subject);
sc.setXMLToken(
doc.getDocumentElement());
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
}

    public Document getAssertion(String subject) {

        //User implementation for getting the
           assertion subject
    }
}
```

Retrieving a SAML Token from an External SAML Authority

This section describes how to configure the client to retrieve a SAML token from an external SAML authority. OracleAS Web Services clients can retrieve the token by issuing a SAMLP request. To retrieve the token, you must configure the `<saml-authority>` element, which is an optional subelement of the `<saml-token>` element. Table 2–15, " Subelements of the <saml-authority> Element" on page 2-15 provides more information on this element.

Example 3–24 illustrates a client configuration that accepts a SAML token from an external SAML authority. The username `jdoe`, the password `password` provide authentication to the SAML authority. The `<require-signature="true">` indicates that the SAMLP request is signed with the client's signature key. This implies that the client-side keystore must be configured with the appropriate signature keys.

Example 3–24 Configuration to Retrieve a SAML Token

```
<saml-token>
  <saml-authority>
    <endpoint-address="http://www.example.com/"
    <auth-user-name="jdoe"
    <auth-password="password"
    <require-signature="true">
  <saml-authority/>
</saml-token>
```

Configure the Keystore

Create a keystore (or use an existing keystore) to store the private keys, public keys and certificates required by the SAML token. For a holder of key or sender-vouches (signed) confirmation method, the keystore must contain the private keys. The sender-vouches (unsigned) does not require a keystore to be configured.

You can use either a Java Keystore (JKS) or Oracle Wallet as the keystore. For the steps involved in configuring either of these keystores, see "Using Keystores" on page 3-1.

Combining Static and Dynamic SAML Configuration

You can combine static and dynamic techniques to provide a SAML configuration to the client. For example, you can declare a SAML subject name statically, and then pass it to the callback handler. See "Configuring SAML Assertion Subject Name and Format" on page 3-26 for information on setting the SAML subject name. See "Writing a SAML Token Callback Handler" on page 3-28 for information on writing a SAML callback handler.

Integrating SAML Token with Security Providers (XML, LDAP, COREid)

Table 3–1 on page 3-7 provides a summary of the security providers that can be used with the SAML security token.

You should configure OC4J with security providers before you configure Web service security. Example 3–8 indicates where you can find additional information on configuring the security providers that are available for the SAML token.

Table 3–8 Information on Security Providers that can be Used with the SAML Token

For this Security Provider Type	See this resource for more Information
File-Based Security Provider	For information on how to configure a file based provider for the SAML security token, see the *Oracle Containers for J2EE Security Guide*.

Table 3–8 (Cont.) Information on Security Providers that can be Used with the SAML

For this Security Provider Type	See this resource for more Information
Oracle Identity Management	For information on how to configure Oracle Identity Management, including Oracle Internet Directory (OID) and Single Sign-on, for the SAML security token, see the *Oracle Containers for J2EE Security Guide*. See "Configuring Single Sign-on Using SAML" on page 3-32 for additional information on how to configure single sign-on for this token.
External LDAP Providers	SAML tokens can be used with external LDAP providers, but they must be configured manually. These manual steps are described in the following sections: ■ Map the SAML Assertion Subject ■ Authenticating SAML Tokens with an External LDAP Provider
COREid	For information on how to configure COREid as a security provider for the token, see "Using COREid as a Security Provider for SAML Token Authentication" For more information on COREid, see the *Oracle Containers for J2EE Security Guide*.

Using COREid as a Security Provider for SAML Token Authentication

If you have COREid access system as part of your deployment, then you can use COREid as a security provider to integrate with the COREid access system. The COREid security provider uses the `CoreIDLoginModule` login module supplied by Oracle. The `CoreIDLoginModule` is defined in the *`ORACLE_HOME`*`/j2ee/`*`instance_name`*`/config/system-jazn-data.xml` file.

Edit the `system-jazn-data.xml` file so that the COREid login module appears above the SAML login module:

1. `SAMLLoginModule`
2. `CoreIDLoginModule`

If you enter the modules in reverse order, then the configuration will be invalid. The `<control-flag>` element must be set to `required` for both `SAMLLoginModule` and `CoreIDLoginModule`.

The login modules are configured on a "per application" basis.

To use COREid with SAML authentication, configure the COREid login module to use the mapping attribute for the authentication plug-in. Set the `coreid.name.attribute` property to the value of the SAML subject defined for verifying the mapping attribute of in the `credential_mapping` plug-in. This property is set as an option under the `<login-module>` element.

For more information on this property and the `CoreIDLoginModule`, see the *Oracle Containers for J2EE Security Guide*.

Example 3–25 illustrates setting the `coreid.name.attribute` property to authenticate the SAML subject. The `subject_variable` value is a variable name used to authenticate the SAML subject in the `credential_mapping` plug-in.

Example 3–25 Setting the SAML Subject Variable for the COREid Login Module for SAML Authentication

```
...
```

```
<login-module>
...
   <option>
     <name>coreid.name.attribute</name>
     <value>subject_variable</value>
   </option>
...
</login-module>
...
```

Example 3–26 illustrates the `SAMLLoginModule` and `CoreIDLoginModule` in the `system-jazn-data.xml` file. Note that the `SAMLLoginModule` is stacked above the `CoreIDLoginModule`. The value of the `<name>` element, *`application_name`*, represents the name of the Web service application.

Example 3–26 Using the COREid and SAML Login Modules for SAML Subject Authentication

```
<application>
   <name>application_name</name>
   <login-modules>
      <login-module>
      <class>
      oracle.security.jazn.login.module.saml.SAMLLoginModule
      </class>
      <control-flag>required</control-flag>
      <options>
         <option>
            <name>addAllRoles</name>
            <value>true</value>
         </option>
         <option>
            <name>issuer.name.1</name>
            <value>www.oracle.com</value>
         </option>
      </options>
      </login-module>
      <login-module>
      <class>
        oracle.security.jazn.login.module.coreid.CoreIDLoginModule
      </class>
      <control-flag>required</control-flag>
      <options>
         <option>
            <name>addAllRoles</name>
            <value>true</value>
         </option>
         <option>
            <name>coreid.resource.type</name>
            <value>res_type</value>
         </option>
         <option>
            <name>coreid.resource.operation</name>
            <value>res_operation</value>
         </option>
         <option>
            <name>coreid.resource.name</name>
            <value>/res_name</value>
         </option>
         <option>
```

```
            <name>coreid.name.attribute</name>
            <value>subject_variable</value>
         </option>
      </options>
      </login-module>
   </login-modules>
</application>
```

Authenticating SAML Tokens with an External LDAP Provider

To authenticate an incoming SAML token against an external LDAP Provider, such as Active Directory or iPlanet, `SAMLLoginModule` and `LDAPLoginModule` must be configured with the *`ORACLE_HOME`*/j2ee/*`instance_name`*`/config/system-jazn-data.xml` file. The modules must be entered into the file in the following order.

1. `SAMLLoginModule`
2. `LDAPLoginModule`

If you enter the modules in reverse order, then the configuration will be invalid. The `<control-flag>` element must be set to `required` for both `SAMLLoginModule` and `LDAPLoginModule`.

The login modules are configured on a "per application" basis.

For more information on `LDAPLoginModule` configuration, see the *Oracle Containers for J2EE Security Guide*.

Example 3–27 illustrates a sample configuration. The `SAMLLoginModule` and `LDAPLoginModule`, and the `<contol-flag>` attributes appear in bold.

Example 3–27 Sample Configuration to Authenticate SAML Tokens for an External LDAP Provider

```
<application>
   <name>MyApplication</name>
   <login-modules>
      <login-module>

<class>oracle.security.jazn.login.module.saml.SAMLLoginModule</class>
         <control-flag>required</control-flag>
         <options>
            ……
         </options>
      </login-module>
                  <login-module>
<class>oracle.security.jazn.login.module.LDAPLoginModule </class>
         <control-flag>required</control-flag>
         <options>
            …..

         </options>
      </login-module>
   </login-modules>
</application>
```

Configuring Single Sign-on Using SAML

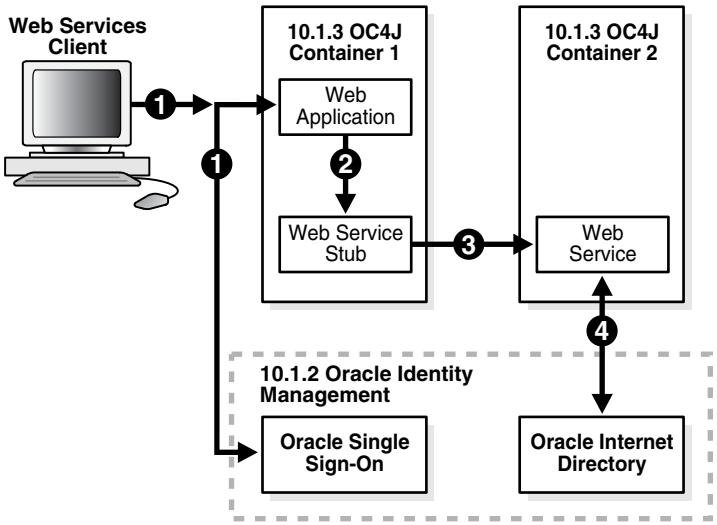
This section describes how to establish Oracle Single Sign-On between a Web application and a Web service deployed on two different containers. It is assumed that both the Web application and the Web service are configured to use SAML.

This scenario assumes that the Web application is deployed on an OC4J container that is associated with Oracle Single Sign-On. When a request arrives at the Web application, it is redirected to Oracle Identity Management where it is authenticated with Oracle Single Sign-On.

The Web application invokes a Web service that uses SAML as its authentication mechanism. The Oracle Single Sign-On identity from the Web application is propagated to the Web service as a SAML assertion. When the Web service application receives the SAML assertion, it verifies the assertion and authenticates the user.

Figure 3–1 illustrates the runtime flow of control between the client, the Web application, and the Web service.

Figure 3–1 Runtime Flow of Single Sign-on Using SAML



The following steps describe the runtime flow between the Web components illustrated in the figure.

1. The client logs in to the Web application; the Web application uses Oracle Single Sign-On for authentication.
2. The Web application calls the Web service stub.
3. The Web service stub invokes the Web service. The stub passes a SAML assertion with the Oracle Single Sign-On identity to the Web service.
4. The Web service application verifies the assertion and uses Oracle Internet Directory to map the Oracle Single Sign-On identity.

The following steps describe how to configure Oracle Single Sign-On between your Web application and a Web service.

Step 1: Install Oracle 10.1.2 Identity Management and Associate OC4J With It

Step 2: Configure the Web application to use Single Sign-on

Step 3a: Assemble and secure the Web service client

Step 3b: Integrate the Web application with the Web service client

Step 4: Assemble and secure the Web service (server side)

Step 1: Install Oracle 10.1.2 Identity Management and Associate OC4J With It

1. Install 10.1.2 Identity Management.

2. Associate 10.1.2 Identity Management with OC4J by using Application Server Control.

   For more information on configuring Oracle Identity Management, see the *Oracle Containers for J2EE Security Guide*.

Step 2: Configure the Web application to use Single Sign-on

Configure your Web application to use Oracle Single Sign-On by using Application Server Control. For more information on using Application Server Control to configure Oracle Single Sign-on, see the Application Server Control on-line help

For more information on configuring Single Sign-On, see the *Oracle Containers for J2EE Security Guide*.

Step 3a: Assemble and secure the Web service client

1. Generate the Web service client by using JDeveloper. In the Security options, choose the SAML sender-vouches confirmation method.
2. Set the `oracle.security.wss.propagate.identity` property at the port level to `true`.

   ```
   <property name="oracle.security.wss.propagate.identity" value ="true"/>
   ```

   If this property is set to `true`, then the Web application authenticated identity is propagated as the SAML subject.
3. Package your Web service proxy/client keystore with your Web application and deploy them to your Web application in the OC4J container (**10.1.3 OC4J Container 1** in Figure 3–1).

Step 3b: Integrate the Web application with the Web service client

1. Get an instance of the Web service stub in the Web application.
2. Set the `Stub.ENDPOINT_ADDRESS_PROPERTY` to the actual Web service endpoint address.

Step 4: Assemble and secure the Web service (server side)

1. Assemble the Web service by using JDeveloper.
2. Deploy the Web service application. Bundle the keystore and signature key and deploy it with the application.
3. Configure your Web service to use SAML sender-vouches confirmation method by using Application Server Control.
4. Configure the OC4J container containing the Web service (**10.1.3 OC4J Container 2** in Figure 3–1) to use Oracle Identity Management by using Application Server Control.

Configuring XML Encryption

Message-level encryption can be configured for a Web service at either the global or port level. The `<encryption-key>` element points to the key required for decrypting the message. If this element is configured at the global level, then it will point to an encryption key in the global-level keystore. If it is configured at the port level, then it will point to an encryption key in the port-level keystore. If the element is not configured at the port level, then global level values are used.

The <encryption-key> element can possess an alias and a password. JDeveloper and the command line keystore tools (Oracle Wallet and Java KeyStore) give you the option of assigning an alias and password when you create the key.

The following sections provide more information on the encryption and decryption of SOAP messages.

- Configuring Encryption for Outbound Messages
- Configuring Encryption for Inbound Elements
- Encrypting the Body of a SOAP Message
- Encrypting Elements of a SOAP Message
- Decrypting Elements of a SOAP Message
- Encrypting a Message with a Signature Key
- Accepting Multiple Keys to Decrypt Messages

Configuring Encryption for Outbound Messages

The <encrypt> subelement of the <outbound> element specifies the confidentiality requirements of the sender. The <encrypt> element contains subelements to specify encryption methods and algorithms. You can specify the encryption algorithm that will be used to encrypt the message with the <encryption-method> element. The <keytransport-method> element identifies an algorithm that can be used to encrypt encryption keys for intended recipients. The <use-request-cert> lets the Web service application encrypt the response message with the signature certificate that the client sent in the original request message

The <recipient-key> element provides the alias and password for the recipient's public key. This key must reside in the keystore. The recipient's public key is used to encrypt the data encryption key.

You can use <tbe-elements> to specify the elements in the SOAP message to be encrypted. This element allows you to encrypt the entire message element or only the contents of an element. "Encrypting the Body of a SOAP Message" on page 3-36 and "Encrypting Elements of a SOAP Message" on page 3-37 provide more information about how to use the <tbe-elements> element for outbound messages.

"Encryption Elements for Outbound Messages" on page 2-17 provides more information on all of the subelements of <encrypt>.

Configuring Encryption for Inbound Elements

For inbound messages, the <decrypt> subelement of the <inbound> element specifies how received messages will be decrypted. You can specify acceptable encryption algorithms for incoming messages with the <encryption-method> element. The <keytransport-method> element identifies acceptable encryption algorithms for encryption keys.

For incoming messages, the <tbe-elements> element indicates which elements are encrypted. "Decrypting Elements of a SOAP Message" on page 3-37 provides more information about how to use the <tbe-elements> element for inbound messages. The recipient's private key used for decryption must be present in the keystore.

The decryption key alias must be configured in the <encryption-key> element of the <keystore> element. "Configuring a Keystore" on page 3-6 provides more information on configuring a keystore.

"Decryption Elements for Inbound Messages" on page 2-10 provides more information on the configuration elements used for decryption.

Encrypting the Body of a SOAP Message

The SOAP message body can be encrypted with Application Sever Control or with JDeveloper. An entry is created for the message body in the `<tbe-element>` element with `Body` as the value of the `local-part` attribute and `CONTENT` as the value of the `mode` attribute. Table 2–18, " Subelements of the <encrypt> Element" on page 2-17 provides more information on these elements and attributes.

> **Note:** If you are encrypting the SOAP `Body` element, the `mode` attribute must be set to `CONTENT`.

Example 3–28 illustrates a sample entry under the `<tbe-element>` element that will encrypt the SOAP message body.

Example 3–28 Encrypting the SOAP Message Body

```
...
<encrypt>
   <recipient-key alias="enckey"/>
   <tbe-elements>
     <tbe-element name-space="http://schemas.xmlsoap.org/soap/envelope"
local-part="Body" mode="CONTENT"/>
   </tbe-elements>
</encrypt>
...
```

Decrypting the Body of a SOAP Message

The SOAP message body can be decrypted with Application Sever Control or with JDeveloper. An entry is created for the message body in the `<tbe-element>` element with `Body` as the value of the `local-part` attribute and `CONTENT` as the value of the `mode` attribute. Table 2–7, " Subelements of the <decrypt> Element" on page 2-10 provides more information on the these elements and attributes.

> **Note:** If you are decrypting the SOAP `Body` element, the `mode` attribute must be set to `CONTENT`.

Example 3–29 illustrates a sample entry under the `<tbe-elements>` element that will decrypt the SOAP message body.

Example 3–29 Decrypting the body of a SOAP Message

```
...
<decrypt>
  <tbe-elements>
    <tbe-element name-space="http://schemas.xmlsoap.org/soap/envelope"
local-part="Body" mode="CONTENT"/>
  </tbe-elements>
</decrypt>
...
```

Encrypting Elements of a SOAP Message

Any part of the SOAP message can be encrypted by adding a `<tbe-element>` element with the `name-space` and `local-part` attributes. If only one element exists in the SOAP message with that specific name, then you can omit the name space.

By default, only the content of the element is encrypted. To encrypt the entire element, set the `mode` attribute to `ELEMENT`.

Example 3–30 illustrates the code for encrypting a user name token.

Example 3–30 Encrypting a User Name Token Element

```
...
<encrypt>
   <recipient-key alias="enckey"/>
   <tbe-elements>
     <tbe-element
name-space=http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-wss-wssecurity-sec
-ext-1.0.xsd local-part="UsernameToken" mode="ELEMENT"/>
   </tbe-elements>
</encrypt>
...
```

Decrypting Elements of a SOAP Message

To decrypt a specific element, specify the `<tbe-element>` with `name-space` and `local-part` attributes. By default, only the content of the element is decrypted. To decrypt the entire element, set the `mode` attribute to `ELEMENT`.

If the element is not encrypted in the incoming request, then the request will be rejected.

Example 3–31 illustrates the code to decrypt a `UsernameToken` element.

Example 3–31 Decrypting a SOAP Message Element

```
...
<decrypt>
   <tbe-elements>
   <tbe-element
name-space=http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-wss-wssecurity-sec
-ext-1.0.xsd local-part="UsernameToken" mode="ELEMENT"/>
   </tbe-elements>
</decrypt>
...
```

Encrypting a Message with a Signature Key

When the Web service application sends a response back to the same client, it can choose to encrypt the response with the signature certificate that the client sent in the first message exchange. Note that this assumes that during the first message exchange the Web service client sent a signed SOAP message and the Web service application successfully verified the signature.

To allow the application to encrypt the response with the client's signature certificate, configure the `<use-request-cert>` element as part of a Web service application's outbound encryption policy. Note that if the server interceptor is unable to find the signature certificate (that is, the client has not sent a signed SOAP message or the signature verification failed) then the Web service application will reject the encryption request.

Accepting Multiple Keys to Decrypt Messages

The default behavior of a service defined under OracleAS Web Services is to accept only one encryption key for the decryption of messages. The boolean property `oracle.security.wss.decryptusingski`, which is set on the `<decrypt>` element, allows a Web service to decrypt a message that could have been encrypted with any one of a number of keys.

If the `decryptusingski` property is set to `true`, then the subject key identifier in the `<encryption-key>` element in the received message is resolved to a private key in the keystore. If this property is set to `false`, then the Web service will accept only one encryption key to decrypt messages. By default, this property is set to `false`.

The following steps describe how to set up the private key in the keystore and the user account to access the key.

1. Add the private key used for decryption to the keystore configured in the `<keystore>` element of the security configuration.
2. Create a user account in *ORACLE_HOME*`/j2ee/`*instance_name*`/config/system-jazn-data.xml` using Application Server Control. The username must be of the type:

   application-name`.`*portname*`.key.<`*decryption-key-alias*`>`

   For example, to create a decryption key for the `SecureService` application with port `SecurePort` and alias `deckey`, the user name would be the following:

   `SecureService.SecurePort.key.deckey`

   The password would be the decryption key password.

Table 2–7, " Subelements of the <decrypt> Element" on page 2-10 provides more information on the `oracle.security.wss.decryptusingski` property.

Example 3–32 illustrates the use of the `oracle.security.wss.decryptusingski` property.

Example 3–32 Decrypting Inbound Messages

```
...
  <inbound>
    ...
    <decrypt>
      ...
      <property name="oracle.security.wss.decryptusingski" value="true"/>
    </decrypt>
    ...
  </inbound>
...
```

Configuring XML Signature

Message-level signature can be configured for a Web service at either the global or port level. The `<signature-key>` element points to the key required for signing the message. If this element is configured at the global level, then it will point to a signature key in the global-level keystore. If it is configured at the port level, then it will point to a signature key in the port-level keystore. If the element is not configured at the port level, then global level values are used. To verify a signature, the trusted root certificate must reside in the keystore. If you are using self-signed certificates, then you can specify an alias in the `<signature-key>` element.

The <signature-key> element can possess an alias and a password. JDeveloper and the command line keystore tools (Oracle Wallet and Java Key Store) give you the option of assigning an alias and password when you create the key.

The following sections provide more information on signing and verifying the signatures on SOAP messages.

- Configuring Signature for Outbound Messages
- Configuring Signature for Inbound Messages
- Signing the Body of a SOAP Message
- Signing Elements of a SOAP Message
- Verifying a Signature on a Specific Element
- Using the Subject Key Identifier for Signing

Configuring Signature for Outbound Messages

The <signature> subelement of the <outbound> element specifies the options that are available for signing outbound messages. You can specify the algorithms that will be used to sign the message with the <signature-method> element.

You can use <tbs-elements> to specify the elements in the SOAP message to be signed. This element allows you to sign the entire message element or only the contents of an element. "Signing the Body of a SOAP Message" on page 3-40 and "Signing Elements of a SOAP Message" on page 3-40 provide more information about how to use the <tbs-elements> element for outbound messages.

The <add-timestamp> element allows you to an additional level of security by adding a timestamp to the signature. This timestamp is verified by the <verify-timestamp> element for incoming messages. "Preventing Replay Attacks with Timestamps" on page 3-41 provides more information on how timestamps can be used to help prevent replay attacks.

Configuring Signature for Inbound Messages

For inbound messages, the <verify-signature> subelement of the <inbound> element specifies how the signatures on received messages will be verified. You can specify acceptable signature algorithms for incoming messages with the <signature-method> element.

The <verify-timestamp> element defines an expiration time on the signature and whether a creation time has been included. This timestamp element is configured with the <add-timestamp> element defined for signing outgoing messages.

For incoming messages, the <tbs-elements> element indicates which elements of the message are signed. "Verifying a Signature on a Specific Element" on page 3-40 provides more information about how to use the <tbs-elements> element for inbound messages.

An additional property, clock-skew, can also be defined for incoming messages. The clock skew value. This property allows you to specify an allowable clock difference between the client and the server.

"Signature Verification Elements for Inbound Messages" on page 2-9 provides more information on these elements.

Signing the Body of a SOAP Message

A digital signature can be applied to the SOAP message body by using Application Sever Control or JDeveloper. For more information, see the topics *Implementing WS-Security for Web Services* in the JDeveloper on-line help or *Configuring Security for a Web Service* in the Application Server Control on-line help.

Signing Elements of a SOAP Message

Any part of a SOAP message can be signed by adding a `<tbs-element>` element with the `name-space` and `local-part` attributes to identify the element. If only one element exists in the SOAP message with the specified name, then you can omit the `name-space` attribute.

Example 3–33 illustrates the code for signing a `UsernameToken` element in a SOAP message.

Example 3–33 Signing a SOAP Message Element

```
<signature>
   <tbs-elements>
   <tbs-element
name-space=http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-wss-wssecurity-sec
-ext-1.0.xsd local-part="UsernameToken"/>
   </tbs-elements>
</signature>
```

You can sign and encrypt the SOAP `Body` element with JDeveloper or Application Server Control. For all other elements, you must edit the `oracle-webservices.xml` and `<generated_name>_Stub.xml` files manually.

Verifying a Signature on a Specific Element

To verify a signature on a specific element, specify the `<tbs-element>` element with `name-space` and `local-part` attributes.

If the element is not signed in the incoming request, then the request will be rejected.

Example 3–34 illustrates the code for verifying a signed user name token.

Example 3–34 Configuration for Verifying a Signature for an Element

```
<verify-signature>
   <tbs-elements>
   <tbs-element
name-space=http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-wss-wssecurity-sec
-ext-1.0.xsd local-part="UsernameToken"/>
   </tbs-elements>
</verify-signature>
```

Using the Subject Key Identifier for Signing

If the receiver of the signed message has prior knowledge of the signer's public key certificate, then instead of sending the entire certificate in the message, you can send the certificate's subject key identifier. The subject key identifier is an extension of the certificate which is used to calculate the public key. If your certificate does not have the subject key identifier, then the request will be rejected.

To sign with subject key identifier, set the `property` element in the signature to `oracle.security.wss.signwithski`, and set the `value` attribute to `true`. By default, this property is set to `false`.

Example 3–35 illustrates how to set the subject key identifier `oracle.security.wss.signwithski` property in the `<property>` subelement.

Example 3–35 Signing a Message Body with a Subject Key Identifier

```
<signature>
 <tbs-elements>
 <tbs-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
</tbs-elements>
     <property name="oracle.security.wss.signwithski" value="true"/>
    </signature>
```

> **Note:** Configure the signature key in the security configuration to point to the signer's private key. The signer's certificate must have the `SubjectKeyIdentifierExtension`. Also, the receiver's keystore must contain the signer's certificate to resolve the subject key identifier.

Preventing Replay Attacks with Timestamps

Applications can use timestamps to prevent replay attacks. A timestamp can be used to protect messages by identifying an expiration time after which the message is no longer valid. Validation is performed by the receiving Web service.

To account for any time differences between the client and the server, you can configure the `clock-skew` property in the `<verify-signature>` element. For more information on this property, see Table 2–6, " Subelements of the <verify-signature> Element" on page 2-9.

Adding Timestamps

The `<add-timestamp>` subelement of the `<signature>` element allows you to add a timestamp to outgoing SOAP messages. To enhance message security, the `expiry` attribute allows you to set an expiration time on the signature; the `created` attribute inserts the signature creation time.

Example 3–36 illustrates a configuration for adding a timestamp to outbound messages. According to the configuration, the outbound message includes a creation time in the timestamp, and the signature should be considered valid for 3000 seconds (50 minutes) after it is sent.

Example 3–36 Configuration to Add a Timestamp to Outbound Messages

```
...
<outbound>
  ...
  <signature>
  ...
    <add-timestamp created="true" expiry="3000"/>
  ...
  </signature>
  ...
</outbound>
```

```
...
```

Verifying TimeStamps

The `<verify-timestamp>` subelement of the `<verify-signature>` element allows you to verify the timestamp placed on inbound SOAP messages. The `<verify-timestamp>` subelement has two attributes, `expiry` and `created`. The `expiry` time sets the expiration time on the freshness of the message. If the message arrives after the expiration time, the message is rejected. If the `created` attribute is set to `true`, then the timestamp on the SOAP message is checked to see if it includes a creation time. If the creation time is not included then the message is rejected.

Example 3–37 illustrates configuration of the verify timestamp. According to the configuration, an inbound message is expected to contain a creation time in its timestamp and the signature should be accepted if message arrives within 28800 seconds (8 hours) of when it is sent.

Example 3–37 Configuring Timestamp Verification on Inbound Messages

```
...
<inbound>
  ...
  <verify-signature>
    ...
    <verify-timestamp expiry="28800" created="true"/>
    ...
  </verify-signature>
...
</inbound>
...
```

Adjusting the Clock Skew Between a Client and a Web Service Application

OracleAS Web Services Security provides a way to synch-up any time differences between the client and a Web service application when they are running on different machines. This is a useful function when SOAP messages with timestamps are being passed.

For example, the client which runs on one machine signs a SOAP message, adds a timestamp, and sends it to the Web Service application. The application, which runs on a separate machine, receives the message and verifies the signature and the timestamp. If the clocks on the two machines are not in sync, there will be a mismatch between the timestamps and the message will be rejected.

Web Services Security provides a `clock-skew` property on the `<verify-signature>` element which will sync-up time differences between the two machines. The default value of `clock-skew` is `0` and the units are measured in milliseconds.

Example 3–38 illustrates setting the clock skew between the client and Web service application to three seconds.

Example 3–38 Setting Clock Skew Between the Client and Web Service Application

```
...
<verify-signature>
    <signature-methods>
       <signature-method>
           RSA-SHA1
       </signature-method>
```

```
    </signature-methods>
    <property name="clock-skew" value="3000"/>
    <verify-timestamp expiry="28800" created="true"/>
    ...
</verify-signature>
...
```

Combining Tokens, Encryption, and Signature in a Configuration

You can combine security tokens, encryption, and signature in a server and client configuration in any combination. The configurations for the client and server must be compatible. As an example, this section illustrates how you can configure the client side and server side deployment descriptors to use a SAML token for authentication with encryption and signature.

Example 3–39 illustrates a configuration for the client side deployment descriptor, *<generated_name>*`.Stub.xml`. The configuration uses the SAML token for authentication, and the message body and SAML assertion are signed for integrity protection. The message body is encrypted to provide message confidentiality. This can be configured using JDeveloper.

By default, the SAML subject confirmation method is sender-vouches (signed). A SAML assertion is generated for the user identity `jdoe` with `sender-vouches` confirmation. The assertion is signed using the `signkey`. The message body is signed using the `signkey` and a timestamp is added for message freshness with the expiration time set to 3000 seconds. The default signature algorithm `RSA-SHA1` is used for signing. The message body is also encrypted with the recipient's key that has the alias `enckey`. The default encryption algorithm is `AES-128` and the default key transport algorithm is `RSA-1_5`. The signature and the recipient's key are stored in `mykeystore.jks`.

Example 3–39 Client-Side Deployment Descriptor Configured for SAML Token, Encryption, and Signature

```
<oracle-webservice-clients>
  <webservice-client>
    <port-info>
      <runtime enabled="security">
        <security>
          <key-store path="./mykeystore.jks" store-pass="password"/>
          <signature-key alias="signkey" key-pass="password"/>
            <outbound>
              <saml-token name="jdoe"/>
                <signature>
                  <tbs-elements>
                    <tbs-element
name-space="http://schemas.xmlsoap.org/soap/envelope/" local-part="Body"/>
                  </tbs-elements>
                  <add-timestamp created="true" expiry="3000"/>
                </signature>
                <encrypt>
                  <recipient-key alias="enckey"/>
                  <tbe-elements>
                    <tbe-element
name-space="http://schemas.xmlsoap.org/soap/envelope/" local-part="Body"/>
                  </tbe-elements>
                </encrypt>
              </outbound>
            </security>
```

```
        </runtime>
       <operations>
        <operation name="sayHello">
        </operation>
       </operations>
     </port-info>
   </webservice-client>
 </oracle-webservice-clients>
```

The server decrypts the message, verifies the signature on the message body and the assertion, and then authenticates the user using the SAML token.

Example 3–40 illustrates the complimentary configuration for the server side deployment descriptor, `oracle-webservices.xml`. This configuration can also be created by using JDeveloper.

The message body is decrypted using the private key with alias `deckey`. The signature is verified (both body and assertion signature) using the public key attached in the SOAP message. The trusted CA certificates are verified using the keystore `mykeystore.jks`. The user is authenticated and a subject is created for the user `jdoe`. The sender's public key certificate and trusted certificates are stored in `mykeystore.jks`. The recipient's public key certificate for decryption is also stored in `mykeystore.jks`.

Note: The subject confirmation method in the `<verify-saml-token>` element can be omitted. If it is present, it must match either the subject confirmation in the `<saml-token>` element or the default value.

Example 3–40 Server-Side Deployment Descriptor Configured for SAML Token, Encryption, and Signature

```
<oracle-webservices>
<webservice-description name="SecureService">
<port-component name="SecurePort">
  <runtime enabled="security">
    <security>
      <key-store path="./mykeystore.jks" store-pass="password"/>
      <signature-key alias="signkey" key-pass="password"/>
      <encryption-key alias="deckey" key-pass="password"/>
    </security>
  </runtime>
  <operations>
    <operation name="sayHello">
      <runtime>
        <security>
          <inbound>
            <verify-saml-token>
              <subject-confirmation-methods>
                <confirmation-method>SENDER-VOUCHES</confirmation-method>
              </subject-confirmation-methods>
            </verify-saml-token>
            <verify-signature>
            <tbs-elements>
              <tbs-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
            </tbs-elements>
            </verify-signature>
            <decrypt>
```

```
            <tbe-elements>
              <tbe-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
              </tbe-elements>
            </decrypt>
          </inbound>
        </security>
      </runtime>
    </operation>
  </operations>
</port-component>
</webservice-description>
</oracle-webservices>
```

4

Building Secure Web Services

This chapter provides the generalized steps for assembling a secure Web service. Oracle Application Server Web Services provides the WebServicesAssembler tool which allows you to assemble the service top down (from a WSDL) or bottom up (from Java classes, EJBs, or database resources). The tool can assemble your service either from Ant tasks in a build script or by entering commands on the command line. For a detailed description of using the WebServicesAssembler tool to build Web services, see the *Oracle Application Server Web Services Developer's Guide*.

> **Note:** The Oracle JDeveloper IDE and the Application Server Control tools can aid your development of secure Web services. You can use the Oracle JDeveloper IDE to build, configure, deploy, and test a secure Web service. You can also use Application Server Control tool to deploy, manage, and monitor it. For more information on using these tools to build and deploy Web services, see the topics *Developing with Web Services* in the JDeveloper on-line help and *Web Services Page* in the Application Server Control on-line help.

Assembling a Secure Web Service

In OracleAS Web Services Security, the security policies for a Web service are specified in XML configuration files. Some examples of the security policies they can describe include username token authentication and XML signature and encryption of the SOAP message body.

The WebServicesAssembler tool adds the policies declared in the configuration files to the Web service at assembly time. The policies can be assembled into the service by either the bottom up or top down approach. You must provide one configuration file for the server and a corresponding file for the client.

The following sections describe how to assemble security into a Web service.

- Assembling Security into a Web Service Top Down
- Assembling Security into a Web Service Bottom Up

These sections describe how to create the files for the server and the client. They also describe how to specify their content to implement username authentication and the XML signature and encryption security policies.

- Creating a Server-Side Security Configuration File
- Creating a Client-Side Security Configuration File

Assembling Security into a Web Service Top Down

The following steps provide a general outline of how to add a security configuration to a Web service application that is being assembled top down (from an existing WSDL). To perform this assembly, the WebServicesAssembler tool provides the `topDownAssemble` command. Only the steps that concern adding security are described in detail. If you need more information on an individual step, see "Assembling a Web Service from a WSDL" in the *Oracle Application Server Web Services Developer's Guide*.

1. Generate the Service Endpoint Interface using WebServicesAssembler.

   Provide a WSDL from which the Web service will be generated as input to the WebServicesAssembler `genInterface` command.

2. Compile the generated interfaces and type classes from Step 1 using the Java compiler.

3. Implement the Java Service Endpoint for the Web service you want to provide.

   The Java Service Endpoint must have a method signature that matches every method in the generated Java interface in Step 1.

4. Compile the Java Service Endpoint.

5. Create a server-side security configuration file.

   The WebServicesAssembler tool uses this file to generate security information into the `oracle-webservices.xml` server-side deployment descriptor. "Creating a Server-Side Security Configuration File" on page 4-8 provides instructions on how to create this file.

6. (Optional) Create a keystore.

   If you are signing or encrypting data, or verifying signatures or decrypting data, you must have a keystore to store trusted certificates and public and private keys. See "Creating a Keystore" on page 3-2 for more information on creating a Oracle Wallet or JKS keystore.

7. (Optional) Bundle the keystore.

   If your server-side security configuration contains a `<decrypt>`, `<verify-signature>`, `<verify-x509-token>`, or `<verify-saml-token>` in the inbound policy, or if it contains an `<encrypt>`, `<x509-token>`, `<saml-token>`, or `<signature>` element in the outbound policy, then you must either bundle a keystore with your application or specify a global-level keystore.

   To bundle a keystore with your application, follow these steps:

   a. Create an `ear/META-INF` directory in the directory where the service is generated (that is, the directory specified as the target of the WebServicesAssembler topDownAssemble command's `output` argument).

   b. Copy the keystore into the `ear/META-INF` directory.

8. Assemble the service.

   Assemble the Web service using WebServicesAssembler `topDownAssemble` command. This is where you generate the security configuration into the Web service. Note that the path to the `serverConfig.xml` file, which contains the security configuration, is specified with the `ddFileName` argument.

```
java -jar wsa.jar -topDownAssemble
                  -wsdl SecureService.wsdl
                  -unwrapParameters false
```

```
                    -className oracle.demo.security.SecureServiceImpl
                    -input build/classes/service
                    -output build
                    -ear dist/secure_service.ear
                    -packageName oracle.demo.security
                    -fetchWsdlImports true
                    -ddFileName serverConfig.xml
                    -classpath ./build/classes/client/ :
                        $OC4J_HOME/jlib/jaxen.jar :
                        $OC4J_HOME/jlib/osdt_wss.jar :
                        $OC4J_HOME/jlib/osdt_cert.jar :
                        $OC4J_HOME/jlib/osdt_xmlsec.jar :
                        $OC4J_HOME/jlib/osdt_core.jar :
                        $OC4J_HOME/jlib/osdt_saml.jar :
                        $OC4J_HOME/jlib/oraclepki.jar :
                        $OC4J_HOME/j2ee/home/jazn.jar :
                        $OC4J_HOME/j2ee/home/jazncore.jar :
                        $OC4J_HOME/j2ee/home/jaznplugin.jar
```

Ant task:

```
<oracle:topDownAssemble appName="secure_service"
         wsdl="./wsdl/SecureService.wsdl"
         unwrapParameters="false"
         input="build/classes/service "
         output="build"
         ear="dist/secure_service.ear"
         packageName="oracle.demo.security"
         fetchWsdlImports="true"
         ddFileName=serverConfig.xml>
         <oracle:portType
            className="oracle.demo.topdowndoclit.service.DocLitLoggerImpl"
         </oracle:portType>
         <oracle:classpath>
             <pathelement location="build/classes/client/ />
             <pathelement location="$OC4J_HOME/jlib/jaxen.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_wss.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_cert.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_xmlsec.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_core.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_saml.jar />
             <pathelement location="$OC4J_HOME/jlib/oraclepki.jar />
             <pathelement location="$OC4J_HOME/j2ee/home/jazn.jar />
             <pathelement location="$OC4J_HOME/j2ee/home/jazncore.jar />
             <pathelement location="$OC4J_HOME/j2ee/home/jaznplugin.jar />
       </oracle:classpath>
 </oracle:topDownAssemble>
```

At a minimum, specify the name of the WSDL, the class name that implements the Service (generated in Step 3), and the name of the output directory. The WebServicesAssembler tool outputs an EAR file, and a WAR file within the EAR. The WAR file contains the service artifacts, the implementation classes, the Web deployment descriptor (`web.xml`) and the JAX-RPC deployment descriptor (`webservices.xml`) and `oracle-webservices.xml` with the security configuration.

> **Note:** If you want to authenticate the user based on the username token, then a user entry must exist with the username and password sent by client. For example, a username and password can be `SCOTT` and `TIGER`. For information on creating a file-based repository of user entries, see the *Oracle Containers for J2EE Security Guide*.

9. Deploy and bind the service into a running instance of OC4J.

   You can deploy the service by using Application Server Control or by using `admin_client.jar` on the command line. For more information on deployment, see *Deploying an Application* in the Application Server Control on-line help and the *Oracle Containers for J2EE Deployment Guide*.

10. Configure a security provider on the server-side to authenticate user data sent by the client.

    For more information on configuring security providers, see the *Oracle Containers for J2EE Security Guide*.

11. (Optional) Check that deployment succeeded. OracleAS Web Services provides a Web Service Home Page for each deployed Web service.

    For more information on the Web Service Home Page, see the *Oracle Application Server Web Services Developer's Guide*.

12. Create the client security configuration file.

    Create a client configuration file that specifies the security that will be applied to your Web service client. In this example, the file is called `clientConfig.xml`. The WebServicesAssembler tool uses this file to generate the client-side deployment descriptor `<generated_name>_Stub.xml`. "Creating a Client-Side Security Configuration File" on page 4-12 provides instructions on how to create the client configuration file.

13. Generate the secure client code.

    For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command.

    For information on assembling a J2EE client, see "Assembling a J2EE Client" in the *Oracle Application Server Web Services Developer's Guide*.

    For example, the following command generates client proxies (stubs) that can be used for a J2SE client. Note that the `ddFileName` argument specifies the client configuration file.

    ```
    java -jar wsa.jar -genProxy
    -wsdl http://localhost:8888/webservice/webservice?WSDL
    -unwrapParameters false
    -output build/src/client
    -packageName oracle.demo.security.stubs
    -ddFileName clientConfig.xml
    ```

    Ant task:

    ```
    <oracle:genProxy
    wsdl="http://localhost:8888/webservice/webservice?WSDL"
    unwrapParameters="false"
    output="build/src/client"
    packageName="oracle.demo.security.stubs"
    ddFileName="clientConfig.xml"
    ```

```
/>
```

At a minimum, specify the name of the WSDL, the name of the output directory and the name of the client configuration file. The WebServicesAssembler tool generates a `<generated_name>_Stub.xml`. A client application uses the stub to invoke operations on a remote service.

14. Compile and run the client.

Assembling Security into a Web Service Bottom Up

The following steps provide a general outline of how to add a security configuration to a Web Service application that is being assembled bottom up. The WebServicesAssembler tool provides several specialized commands based on whether you are assembling the services from Java classes, EJBs, JMS destinations, or database resources. These commands are summarized in "Ant Tasks and WebServicesAssembler" on page 4-21.

In the following example, a Web service is assembled from Java classes. To perform this assembly, the WebServicesAssembler tool provides the `assemble` command. Only the steps that concern adding security are described in detail. If you need more information on an individual step, see "Assembling a Web Service with Java Classes" in the *Oracle Application Server Web Services Developer's Guide*.

1. Provide the compiled Java class that you want to expose as a Web service and its compiled interface.
2. Create the Web service security configuration file.

   In this example, the file is called `serverConfig.xml`. The WebServicesAssembler tool uses this file to generate security information into the `oracle-webservices.xml` server-side deployment descriptor. "Creating a Server-Side Security Configuration File" on page 4-8 provides instructions on how to create this file.
3. (Optional) Create a keystore.

   If you are signing or encrypting data, or verifying signatures or decrypting data, you must have a keystore to store trusted certificates and public and private keys. See "Creating a Keystore" on page 3-2 for more information on creating a Oracle Wallet or JKS keystore.
4. (Optional) Bundle the keystore.

   If your server-side security configuration contains a `<decrypt>`, `<verify-signature>`, `<verify-x509-token>`, or `<verify-saml-token>` in the inbound policy, or if it contains an `<encrypt>`, `<x509-token>`, `<saml-token>`, or `<signature>` element in the outbound policy, then you must either bundle a keystore with your application or specify a global-level keystore.

   To bundle a keystore with your application, follow these steps:

   a. Create an `ear/META-INF` directory in the directory where the service is generated (that is, the directory specified as the target of the WebServicesAssembler `output` argument).

   b. Copy the keystore into the `ear/META-INF` directory.
5. Assemble the secure Web service.

   Assemble the Web service by running the WebServicesAssembler tool with the appropriate `*Assemble` command for the Web service you are creating. Note that

the path to the `serverConfig.xml` server configuration file is specified with the `ddFileName` argument.

Note that this example assumes that the command is run on the UNIX operating system. Directory paths use the forward slash (/) and the classpath argument uses a colon (:) to separate individual classpath elements. If you are using the Windows operating system, use a back slash (\) in directory paths and separate individual classpath elements with a semicolon (;).

```
java -jar wsa.jar -assemble
          -appName securehello
          -serviceName SecureHelloService
          -interfaceName oracle.demo.hello.HelloInterface
          -className oracle.demo.hello.HelloImpl
          -input ./build/classes/client
          -output build
          -ear dist/securehello.ear
          -uri SecureHelloService
          -ddFileName serverConfig.xml
          -classpath ./build/classes/client/ :
                     $OC4J_HOME/jlib/jaxen.jar :
                     $OC4J_HOME/jlib/osdt_wss.jar :
                     $OC4J_HOME/jlib/osdt_cert.jar :
                     $OC4J_HOME/jlib/osdt_xmlsec.jar :
                     $OC4J_HOME/jlib/osdt_core.jar :
                     $OC4J_HOME/jlib/osdt_saml.jar :
                     $OC4J_HOME/jlib/oraclepki.jar :
                     $OC4J_HOME/j2ee/home/jazn.jar :
                     $OC4J_HOME/j2ee/home/jazncore.jar :
                     $OC4J_HOME/j2ee/home/jaznplugin.jar
```

Ant task:

```
<oracle:assemble appName="securehello"
        serviceName'"SecureHelloService"
        input="./build/classes/client"
        output="build"
        ear="dist/securehello.ear"
        >
         <oracle:porttype
           interfaceName="oracle.demo.hello.HelloInterface"
           className="oracle.demo.hello.HelloImpl"
           <oracle:port uri="SecureHelloService" />
         </oracle:porttype>
         <oracle:classpath>
             <pathelement location="build/classes/client/ />
             <pathelement location="$OC4J_HOME/jlib/jaxen.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_wss.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_cert.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_xmlsec.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_core.jar />
             <pathelement location="$OC4J_HOME/jlib/osdt_saml.jar />
             <pathelement location="$OC4J_HOME/jlib/oraclepki.jar />
             <pathelement location="$OC4J_HOME/j2ee/home/jazn.jar />
             <pathelement location="$OC4J_HOME/j2ee/home/jazncore.jar />
             <pathelement location="$OC4J_HOME/j2ee/home/jaznplugin.jar />
         </oracle:classpath>
</oracle:assemble>
```

The output of this command is an EAR file that contains the contents of a WAR file that can be deployed to an OC4J instance. The `build` directory specified by the

`output` argument contains separate directories for the EAR file and the Java classes. The `dist` directory contains the J2EE Web services-compliant application EAR file, `securehello.ear`.

Note: If you want to authenticate the user based on the username token, then a user entry must exist with the username and password sent by client. For example, a username and password can be `SCOTT` and `TIGER`. See "Configuring File Based Providers" in the *Oracle Containers for J2EE Security Guide* for information on adding user entries to the server.

---

6. Deploy and bind the service.

   You can deploy the service by using Application Server Control or by using `admin_client.jar` on the command line. For more information on deployment, see *Deploying an Application* in the Application Server Control on-line help and the *Oracle Containers for J2EE Deployment Guide*.

7. Configure a security provider on the server-side to authenticate user data sent by the client.

   See the *Oracle Containers for J2EE Security Guide* for more information on security provider configurations.

8. (Optional) Check that deployment succeeded. OracleAS Web Services provides a Web Service Home Page for each deployed Web service.

   For more information on the Web Services Home Page, see "Testing Web Service Deployment" in the *Oracle Application Server Web Services Developer's Guide*.

9. Create a client security configuration.

   Create a client configuration file that specifies the security that will be applied to your web service client. In this example, the file is called `clientConfig.xml`. The WebServicesAssembler tool uses this file to generate the client-side deployment descriptor `<generated_name>_Stub.xml`. "Creating a Client-Side Security Configuration File" on page 4-12 provides instructions on how to create the client configuration file.

10. Assemble secure client code.

    For the J2SE environment, generate stubs (client proxies) for a J2SE Web service client by running the WebServicesAssembler tool with the `genProxy` command.

    For information on assembling a J2EE client, see "Assembling a J2EE Client" in the *Oracle Application Server Web Services Developer's Guide*.

    For example, the following command generates client proxies (stubs) that can be used for a J2SE client. Note that the `ddFileName` argument specifies the client configuration file.

    ```
    java -jar wsa.jar -genProxy
    -wsdl http://localhost:8888/webservice/webservice?WSDL
    -unwrapParameters false
    -output build/src/client
    -packageName oracle.demo.security.stubs
    -ddFileName clientConfig.xml
    ```

    Ant task

    ```
    <oracle:genProxy
    ```

```
wsdl="http://localhost:8888/webservice/webservice?WSDL"
unwrapParameters="false"
output="build/src/client"
packageName="oracle.demo.security.stubs"
ddFileName="clientConfig.xml"
/>
```

At a minimum, specify the name of the WSDL, the name of the output directory, and the client configuration file. The WebServicesAssembler tool generates a *<generated_name>*_Stub.xml. A client application uses the stub to invoke operations on a remote service.

11. Compile and run the client.

Creating a Server-Side Security Configuration File

This section describes how to create a server-side Web service security configuration file to verify a username token. In this example, the file is called `serverConfig.xml`. The WebServicesAssembler tool appends the data found in this file into the `oracle-webservices.xml` server-side deployment descriptor.

> **Note:** If you are using JDeveloper to secure your Web service, then the security configuration is automatically generated into the `oracle-webservices.xml` file.

The following general steps describe how to create the server-side Web service security configuration file.

1. Create a file using any text editor.

   Example 4–1 provides a sample server-side configuration file named `serverConfig.xml`.

2. Enter the XML elements to set the security configuration in the file.

   The following sections provide sample server-side Web service security configurations for username token and for XML signature and encryption.

   - "Defining a Server-Side, Port Level Security Configuration for Username Token" provides a sample configuration file that enforces username token verification at the port-level.
   - "Defining a Server-Side, Operation-Level Security Configuration for Username Token" provides a sample configuration file that enforces username token verification at the operation-level.
   - "Defining a Server-Side, Port-Level Security Configuration to Verify XML Signature and Decryption" provides a sample configuration file that implements XML signature and XML encryption at the port level.
   - "Defining a Server-Side, Operation-Level Security Configuration for XML Signature and Decryption" provides a sample configuration file that implements XML signature and XML encryption at the operation level.

3. Save the file.

Defining a Server-Side, Port Level Security Configuration for Username Token

Example 4–1 illustrates a sample server-side Web services configuration file which enforces security on the port level. This means that the user name token will have to be verified before access is granted to any of the operations in the port. Note that line numbers have been added for reference purposes only. They should not appear in your file. "Security Elements for Inbound Messages" on page 2-7 provides more information on the `<verify-username-token>` element for inbound security.

- Line 2: This line identifies the Web service to which security will be applied. The `name` attribute value should match the service name in the WSDL.
- Line 3: This line identifies the port name. It must match a valid port name in the WSDL.
- Lines 4-10: In this example security is applied at port level. The `<runtime enabled="security">` element indicates that at run time, Web services security will be enforced based on the policy defined in the `<security>` element. Since security is configured at port level in this example, the same security will be applied to all operations exposed by this Web service.
- Lines 6-8: These lines configure the inbound security for the incoming request. In this example, the `<verify-username-token/>` element indicates that the Web service application is expecting a username token in the request header. You can also enforce the password type and whether a nonce is required. See "Security Elements for Inbound Messages" on page 2-7 for more information on this element.
- Lines 11-13: These lines define the operations that the Web service port exposes.

Example 4–1 Sample Server-Side Configuration File for Port-Level Username Token Verification

```
1. <oracle-webservices>
2.   <webservice-description name="SecureService">
3.       <port-component name="SecureServiceSoap">
4.       <runtime enabled="security">
5.       <security>
6.          <inbound>
7.           <verify-username-token/>
8.          </inbound>
9.       </security>
10.      </runtime>
11.      <operations>
12.        <operation name="getCatalog"/>
13.      </operations>
14.      </port-component>
15.   </webservice-description>
16. </oracle-webservices>
```

Defining a Server-Side, Operation-Level Security Configuration for Username Token

Example 4–2 illustrates a sample server-side Web services configuration file which enforces security on the operation level. This means that the username token will have to be verified before access is granted to the specified operation. Note that line numbers have been added for reference purposes only. They should not appear in your file. See "Security Elements for Inbound Messages" on page 2-7. for more information on the `<verify-username-token>`element.

This configuration is similar to Example 4–1 with one major difference:

- Lines 6-14: The username verification is applied only for the `getCatalog` operation.

Example 4–2 Sample Server-Side Configuration File for Operation Level Username Token Verification

```
1. <oracle-webservices>
2.   <webservice-description name="SecureService">
3.   <port-component name="SecureServiceSoap">
4.       <runtime enabled="security"/>
5.   <operations>
6.     <operation name="getCatalog">
7.     <runtime>
8.     <security>
9.     <inbound>
10.       <verify-username-token/>
11.    </inbound>
12.    </security>
13.    </runtime>
14.  </operation>
15. </operations>
16. </port-component>
17. </webservice-description>
18. </oracle-webservices>
```

Defining a Server-Side, Port-Level Security Configuration to Verify XML Signature and Decryption

Since the settings are made on the port level, verification must take place before access is granted to any of the operations in the port. See "Keystore Elements" on page 2-5 and "Signature and Encryption Key Elements" on page 2-6 for more information on configuring keystore, encryption and signature elements. Note that line numbers have been added for reference purposes only. They should not appear in your file.

- Lines 4 -25: These lines set the keystore, keys, and inbound policy at the port level. Chapter 3, "Administering Web Services Security" provides more information on generating keystore and keys.
- Lines 6-8: These lines set the `key-store path` attribute to point to the keystore that is deployed with the application. The `store-pass` attribute contains the keystore password. The `signature-key alias` attribute is used to verify the trusted certificate. The `encryption-key alias` attribute is used to decrypt the message. Note that `signalias/encalias` must be in the `testks.jks` file.
- Lines 11-18: These lines specify the message integrity policy. The server expects the body and the timestamp to be signed. The signature must have a timestamp with the created value set to the creation time. The `expiry` attribute specifies when the signature will expire (that is, the current time must be less than the value of the creation time plus the value of `expiry`). The value of the `clock-skew` attribute is used to adjust the system clocks, when the Web service client and the Web service are on different machines.
- Lines 19-23: These lines specify the message confidentiality policy. The server expects the content of the message body to be encrypted.

Example 4–3 Sample Server-Side Configuration File with Keystore and Inbound Policy for Decryption and Signature Verification

```
1. <oracle-webservices>
2. <webservice-description name="SecureHelloService">
```

```
3. <port-component name="SecureHelloPort">
4. <runtime enabled="security">
5. <security>
6. <key-store path="META-INF/testks.jks" store-pass="keystorepwd"/>
7. <signature-key alias="signalias" key-pass="signkeypwd"/>
8. <encryption-key alias="encalias" key-pass="enckeypwd"/>
9.     </security>
10.    <inbound>
11.      <verify-signature>
12.        <tbs-elements>
13.          <tbs-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
14.          <tbs-element
name-space="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-uti
lity-1.0.xsd" local-part="Timestamp"/>
15.        </tbs-elements>
16.          <verify-timestamp created="true" expiry="28800"/>
17.          <property name="clock-skew" value="3000" />
18.      </verify-signature>
19.      <decrypt>
20.        <tbe-elements>
21.          <tbe-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
22.        </tbe-elements>
23.      </decrypt>
24.    </inbound>
25.   </runtime>
26.   <operations>
27.     <operation name="sayHello">
28.     </operation>
29.   </operations>
30.   </port-component>
31.   </webservice-description>
32. </oracle-webservices>
```

Defining a Server-Side, Operation-Level Security Configuration for XML Signature and Decryption

Example 4–4 illustrates a sample server-side Web services configuration file where the global level keystore settings are used for signature verification and decryption. Note that line numbers have been added for reference purposes only. They should not appear in your file.

- Lines 7-26: These lines indicate that security is enforced for the `sayHello()` operation. The server expects to receive signed and encrypted messages.
- Lines 10-17: These lines specify the message integrity policy. The server expects the message body and the timestamp to be signed. The signature must have a timestamp with the created value set to creation time. The `expiry` attribute specifies when the timestamp will expire (that is, the current time must be less than the value of the creation time plus the value of `expiry`). The `clock-skew` attribute is used to adjust the system clocks, when the Web service client and the Web service are on different machines.
- Lines 18 -22: These lines specify the message confidentiality policy. The server expects the content of the body to be encrypted.

Example 4–4 Sample Server-Side Configuration File with Operation-Level Decryption and Signature Verification

```
1. <oracle-webservices>
2.   <webservice-description name="SecureHelloService">
3.   <port-component name="SecureHelloPort">
4.   <runtime enabled="security">
5.   <operations>
6.     <operation name="sayHello">
7.     <runtime>
8.     <security>
9.     <inbound>
10.     <verify-signature>
11.      <tbs-elements>
12.        <tbs-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
13.        <tbs-element
name-space="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-uti
lity-1.0.xsd" local-part="Timestamp"/>
14.      </tbs-elements>
15.        <verify-timestamp created="true" expiry="28800"/>
16.        <property name="clock-skew" value="3000" />
17.    </verify-signature>
18.    <decrypt>
19.      <tbe-elements>
20.        <tbe-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
21.      </tbe-elements>
22.    </decrypt>
23.    </inbound>
24.    </security>
25.    </runtime>
26.    </operation>
27.   </operations>
28.  </port-component>
29.  </webservice-description>
30. </oracle-webservices>
```

Creating a Client-Side Security Configuration File

This section describes how to create a client-side Web service security configuration file to generate a username token. In this example, the file is called `clientConfig.xml`. The WebServicesAssembler tool appends the data found in this file into the *<generated_name>*`_Stub.xml` client-side deployment descriptor.

> **Note:** If you are using JDeveloper to secure your Web service, then the security configuration is automatically generated into the *<generated_name>*`_Stub.xml` file.

The following general steps describe how to create the client-side Web service security configuration file.

1. Create a file using any text editor.

   Example 4–5 provides a sample client-side configuration file named `clientConfig.xml`.

2. Enter the XML elements to generate a username token security configuration in the file.

The following sections provide examples of client-side security configuration files for username token and for signature and encryption.

- "Defining a Client-Side, Port Level Security Configuration for Username Token" provides a sample configuration file that allows a user name and password to be sent to access any operations exposed at the port level.
- "Defining a Client-Side, Port-Level Security Configuration for XML Signature and Encryption" provides a sample configuration file that signs and encrypts the message body.

3. Save the file.

Defining a Client-Side, Port Level Security Configuration for Username Token

Example 4–5 illustrates a sample client-side Web services configuration file which enforces security on the port level. In this example, the client must send a user name and password to access any operations that the service exposes at the port level. Note that line numbers have been added for reference purposes only. They should not appear in your file. "Username Token Elements for Outbound Messages" on page 2-12 provides more information on the `<username-token>` element.

- Lines 4-10: In this example, security is applied at port level. The `<runtime enabled="security">` element indicates that at run time, the client will be secured based on the policy defined in the `<security>` element. Since the security is configured at the port level, the same security will be applied to all operations of the Web service.
- Lines 6-8: The `<outbound>` clause is where you define the outbound message policy. The `<username-token>` element indicates that a username token will be added to the security header with username `SCOTT` and password `TIGER`. You can also specify the password type as either `PLAINTEXT` (default) or `DIGEST`.

Example 4–5 Sample Client-Side Web Services Configuration File with Port-Level Username Token Security

```
1. <oracle-webservice-clients>
2.  <webservice-client>
3.   <port-info>
4.   <runtime enabled="security">
5.   <security>
6.   <outbound>
7.     <username-token name="SCOTT " password="TIGER"/>
8.   </outbound>
9.   </security>
10.   </runtime>
11.     <operations>
12.     <operation name="sayHello"/>
13.    </operations>
14. </port-info>
15. </webservice-client>
16. </oracle-webservice-clients>
```

Defining a Client-Side, Port-Level Security Configuration for XML Signature and Encryption

Example 4–6 illustrates a sample client-side Web services configuration file which enforces security on the port level. In this example, the configuration sets the security policy for the client side keystore, the key configuration policy, and the outbound

message policy. Note that line numbers have been added for reference purposes only. They should not appear in your file.

- Lines 4-24: These lines specify the security policy for the client side keystore, the key configuration policy, and the outbound message policy. You must generate the keystore and the signature and encryption keys. For more information on creating a keystore and adding keys, see "Using Keystores" on page 3-1.
- Lines 6-7: These lines specify the client side keystore path and signature key alias.
- Lines 8-22: These lines specify the outbound message policy.
- Lines 9-15: These lines specify the signature policy. In this example, these lines indicate that the body and timestamp must be signed.
- Lines 16-21: Specify the encryption policy and the recipients key that will be used to encrypt the request.

Example 4–6 Sample Client-Side Web Services Configuration File with Port Level Signature and Encryption Security

```
1. <oracle-webservice-clients>
2.  <webservice-client>
3. <port-info>
4.   <runtime enabled="security">
5.   <security>
6.     <key-store path="etc/testks.jks" store-pass="keystorepwd"/>
7.     <signature-key alias="signalias" key-pass="signkeypwd"/>
8.       <outbound>
9.        <signature>
10.          <tbs-elements>
11.            <tbs-element
name-space="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-uti
lity-1.0.xsd" local-part="Timestamp"/>
12.           <tbs-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
13.         </tbs-elements>
14.           <add-timestamp created="true" expiry="28800"/>
15.       </signature>
16.       <encrypt>
17.         <tbe-elements>
18.           <tbe-element name-space="http://schemas.xmlsoap.org/soap/envelope/"
local-part="Body"/>
19.           </tbe-elements>
20.           <recipient-key alias="encalias"/>
21.       </encrypt>
22.      </outbound>
23.  </security>
24.  </runtime>
25.  <operations>
26.    <operation name="sayHello">
27.    </operation>
28.  </operations>
29. </port-info>
30. </port-component>
31. </webservice-description>
32. </oracle-webservices>
33. </webservice-client>
34. </oracle-webservice-clients>
```

Creating Users For Authentication

The server requires username and password data to authenticate users trying to access the Web service secured with the username policy. To provide this data, you can enter information for each user in the instance level `system-jazn-data.xml` file. You can find this file at *`OC4J_HOME`*`/j2ee/<`*`instance_name`*`>/config/system-jazn-data.xml`. For more information on the contents of the `system-jazn-data.xml` file, see the *Oracle Containers for J2EE Security Guide*.

Adding User Entries via Application Server Control

The Application Server Control console provides screens where you can create users and passwords and add them to the server. For more information, see the topic *Managing Security Provider Roles and Users* in the Application Server Control on-line help.

Client JAR Files

Example 4–1 lists the client JAR files required if you are assembling security into your Web service client. "Web Service APIs and JARs" in the *Oracle Application Server Web Services Developer's Guide* lists the other JARs that must appear on the classpath to compile a Web service client.

Table 4–1 Client JAR Files for Security

JAR Name and Path	Description
OC4J_HOME/jlib/jaxen.jar	Contains the classes that define Jaxen—a Java XPath Engine capable of evaluating XPath expressions across multiple modes (such as dom4j, JDOM, and so on).
OC4J_HOME/jlib/osdt_wss.jar	Contains Oracle Security Developer's Toolkit Web services security (WS-Security) APIs.
OC4J_HOME/jlib/osdt_cert.jar	Contains the Oracle Security Developer's Toolkit cryptography APIs.
OC4J_HOME/jlib/osdt_xmlsec.jar	Contains Oracle Security Developer's Toolkit XML signing and encryption APIs.
OC4J_HOME/jlib/osdt_core.jar	Contains the Oracle Security Developer's Toolkit (OSDT) APIs
OC4J_HOME/jlib/osdt_saml.jar	Contains Oracle Security Developer's Toolkit Security Assertion Markup Language (SAML) APIs.
OC4J_HOME/jlib/oraclepki.jar	Contains the Oracle `orapki` keytool utility.
OC4J_HOME/j2ee/home/jazn.jar	Contains the JAZN (Oracle JAAS provider) administration tool.
OC4J_HOME/j2ee/home/jazncore.jar	Contains the JAZN (Oracle JAAS provider) implementation.
OC4J_HOME/j2ee/home/jaznplugin.jar	Contains the JAZN (Oracle JAAS provider) custom plug-in module.

Adding Transport-Level Security to a Web Service

You can secure a Web service on the transport level by using basic, digest, or client certification (client-cert) authentication. If your Web service was assembled from a version 2.1 or 3.0 EJB, you can secure it on the transport level by making additions to

the `oracle-webservices.xml` deployment descriptor. This section also provides information on how to write J2SE and J2EE clients to access Web services secured on the transport level.

Adding Basic Authentication

With basic authentication, the user is prompted directly for a user name and password, without going through OracleAS Single Sign-On. A login module (such as `RealmLoginModule`, for example) is used to generate a login dialog.

To specify basic authentication at the transport level, provide a value of `BASIC` for the `<auth-method>` subelement of `<login-config>` in `web.xml`. For example:

```
<web-app ...>
   ...
   <login-config>
      <auth-method>BASIC</auth-method>
      ...
   <login-config>
   ...
</web-app>
```

For more information on providing basic authentication, see the *Oracle Containers for J2EE Security Guide*.

Adding Digest Authentication

With the digest authentication mechanism, the password that a client presents to authenticate itself is encrypted through the use of an MD5 digest. This is transmitted in the request message. From a user perspective, digest authentication behaves in the same way as basic authentication.

To specify digest authentication at the transport level, provide a value of `DIGEST` for the `<auth-method>` subelement of `<login-config>` in `web.xml`. For example:

```
<web-app ...>
   ...
   <login-config>
      <auth-method>DIGEST</auth-method>
      ...
   <login-config>
   ...
</web-app>
```

For more information on providing digest authentication see the *Oracle Containers for J2EE Security Guide*.

Adding Client Certification Authentication

The client certification (client-cert) method authenticates the client through HTTPS. The user must possess a public key certificate.

To specify client-cert authentication at the transport level, provide a value of `CLIENT-CERT` for the `<auth-method>` subelement of `<login-config>` in `web.xml`. For example:

```
<web-app ...>
   ...
   <login-config>
      <auth-method>CLIENT-CERT</auth-method>
      ...
   <login-config>
   ...
</web-app>
```

For more information on providing client-cert authentication, see the *Oracle Containers for J2EE Security Guide*.

Adding Transport-Level Security for Web Services Based on EJBs

Version 2.1 and 3.0 Enterprise Java Beans (EJBs) can be exposed as a Web service. You can define transport-level security constraints Web services based on EJBs by configuring the `<ejb-transport-security-constraint>` and `<ejb-transport-login-config>` elements in the `oracle-werbservices.xml` deployment descriptor.

The `<ejb-transport-security-constraint>` element lets you specify whether the security constraints should apply to a SOAP port, a WSDL URL, or both. You can also specify a security role and a transport guarantee.

The `<ejb-transport-login-config>` element lets you specify whether the EJB application uses basic authentication, digest authentication, or client certificate as its authentication mechanism.

The client of a secured EJB Web service can be configured to pass a username and password to the secured service either statically, by entering configuration parameters into the proprietary deployment descriptors, such as `orion-ejb-jar.xml` or `orion-web.xml` file, or programmatically. "Accessing Web Services Secured on the Transport Level" provides more information on writing a client that can access a secured Web service.

For more information on how to use these elements and their subelements, see the `oracle-webservices-10_0.xsd` schema and "Packaging and Deploying Web Services" in the *Oracle Application Server Web Services Developer's Guide*.

Example 4–7 illustrates `<ejb-transport-security-constraint>` and its subelements in the `oracle-webservices.xml` deployment descriptor. This element associates transport-level security constraints for a version 2.1 or 3.0 EJB exposed as a Web service. The URL of the EJB exposed as a Web service is indicated by the `<endpoint-address-uri>` element in the port component. The sub-elements `<wsdl-url>` and `<soap-port>` are identifiers that let you choose whether the security constraints will apply to a WSDL URL or to a SOAP port. If `<wsdl-url>` and `<soap-port>` are both present or both absent in the `<ejb-transport-security-constraint>` element, then the security constraints will apply to both the WSDL and the SOAP port.

Example 4–7 <ejb-transport-security-constraint> Element in oracle-webservices.xml

```
...
   <port-component name="String">
       <endpoint-address-uri>String</endpoint-address-uri>
       <ejb-transport-security-constraint>
```

```
            <wsdl-url/>
            <soap-port/>
            <role-name>Manager</role-name>
            <role-name>Administrator</role-name>
            <transport-guarantee>NONE</transport-guarantee>
        </ejb-transport-security-constraint>
        ...
    </port-component>
...
```

Example 4–8 illustrates `<ejb-transport-login-config>` and its subelements in the `oracle-webservices.xml` deployment descriptor. This element configures the transport-level authentication method (such as basic, digest, or client certificate) and the realm name that should be used for this EJB application. The URL of the EJB application exposed as a Web service is indicated by the `<endpoint-address-uri>` element in the port component.

Example 4–8 <ejb-transport-login-config> Element in oracle-webservices.xml

```
...
    <ejb-transport-login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>sec-ejb</realm-name>
    </ejb-transport-login-config>
...
```

Accessing Web Services Secured on the Transport Level

A Web service that is secured on the transport level, either by basic or digest authentication, can be accessed by a J2SE or J2EE Web service client. To access the service, the J2EE platform provides properties that can be used by a client side proxy (stub) or a dynamic invocation of the service endpoint. These properties are used to pass the user name and password to the service.

The following properties can be used by the `javax.xml.rpc.Stub` and `javax.xml.rpc.Call` interfaces.

```
javax.xml.rpc.security.auth.username
javax.xml.rpc.security.auth.password
```

For convenience, these interfaces also define constant field values for these properties.

```
javax.xml.rpc.Stub.USERNAME_PROPERTY
javax.xml.rpc.Stub.PASSWORD_PROPERTY
javax.xml.rpc.Call.USERNAME_PROPERTY
javax.xml.rpc.Call.PASSWORD_PROPERTY
```

The following sections describe how to use these field values and properties to access the secured Web service from a J2SE or J2EE Web service client.

- Accessing a Secured Service from a J2SE Client
- Accessing a Secured Service from a J2EE Client

Accessing a Secured Service from a J2SE Client

To access a Web service secured by transport level basic or digest authentication, a J2SE client can pass a username and password by using either the stub field values, `Stub.USERNAME_PROPERTY` and `Stub.PASSWORD_PROPERTY`, or helper class methods.

Example 4–9 illustrates Web service client code which uses the `Stub.USERNAME_PROPERTY` and `Stub.PASSWORD_PROPERTY` properties.

Example 4–9 Accessing a Secured Service from a J2SE Client with Stub Properties

```
....
javax.xml.rpc.Stub port = (Stub)time.getPort();
port.setProperty(Stub.USERNAME_PROPERTY, "helen");
port.setProperty(Stub.PASSWORD_PROPERTY, "password");
...
```

To provide programmatic access to the service, these stub properties can be replaced by the following helper class methods:

```
setPassword(String userName)
setUsername(String password)
```

For more information on helper classes in the client utility class file, see "Writing Web Service Client Applications" in the *Oracle Application Server Web Services Developer's Guide*.

Accessing a Secured Service from a J2EE Client

To access a Web service secured by transport level basic or digest authentication, a J2EE client can pass a username and password to the service. This information can be passed statically, by entering it in the deployment descriptor for your Web service, or programmatically.

- Passing Authentication Information in a Deployment Descriptor
- Passing Authentication Information Programmatically

Passing Authentication Information in a Deployment Descriptor To pass the username and password in a deployment descriptor, you must manually edit the `<service-ref-mapping>` element in the `orion-*.xml` file for your service.

If you are accessing the service endpoint with a static client proxy add, a `<stub-property>` subelement to the `<service-ref-mapping>` element. If you are dynamically invoking the service endpoint using Dynamic Invocation Interface (DII), add a `<call-property>` element instead. For more information on the `<call-property>` and `<stub-property>` elements, see the *Oracle Application Server Web Services Developer's Guide*.

The following steps describe how to edit the `<service-ref-mapping>` clause and the values you must provide for the `<call-property>` and `<stub-property>` elements.

1. Edit the `<service-ref-mapping>` element of the appropriate `orion-*.xml` deployment descriptor for your Web service.
   - If you are working with a client proxy, enter a `<stub-property>` subelement for the username and password, or
   - If you are dynamically invoking the service endpoint, enter a `<call-property>` subelement.
2. Specify the username property `javax.xml.rpc.security.auth.username` and its value in the `<name>` and `<value>` subelements of the `stub-` or `call-property` element entered in Step 1.

3. Specify the username property `javax.xml.rpc.security.auth.password` and its value in the `<name>` and `<value>` subelements of the `stub-` or `call-property` element entered in Step 1.
4. When you generate your Web service client, specify the `orion-*.xml` deployment descriptor as input to the WebServicesAssembler `genProxy` command.

Example 4–10 illustrates a sample `<service-ref-mapping>` clause which uses `<stub-property>` elements to specify a username and password to access the secured Web service.

Example 4–10 Accessing a Secured Service from a Static J2EE Client Configuration

```
<service-ref-mapping name="service/MyHelloServiceRef">
    ...
    <stub-property>
             <name>javax.xml.rpc.security.auth.password</name>
             <value>welcome</value>
    </stub-property>
    <stub-property>
               <name>javax.xml.rpc.security.auth.username</name>
               <value>helen</value>
    </stub-property>
 </service-ref-mapping
```

Passing Authentication Information Programmatically Instead of editing the `orion-*.xml` file, you can use the `Stub.USERNAME_PROPERTY` and `Stub.PASSWORD_PROPERTY` properties to pass these values programmatically to the service via the `setProperty` method.

Example 4–11 Accessing a Secured Service from a J2EE Client Programmatically

```
Context ic = new InitialContext();
Service service = (Service) ic.lookup("java:comp/env/service/MyHelloServiceRef");
//Service.getPort(portQName,SEI class)
HelloInterface helloPort = (HelloInterface)
service.getPort(portQName,hello.HelloInterface.class);
Stub port = (Stub) helloPort;
     port.setProperty(Stub.USERNAME_PROPERTY, "helen");
     port.setProperty(Stub.PASSWORD_PROPERTY, "password");
```

Propagating Identities from a Web Service to an EJB

The Oracle Web Service Security implementation provides seamless integration for propagating the user's identity when a Web service invokes an EJB.

The Web service is authenticated by using message level security. Re-authentication is not required when accessing the EJB from the Web service. The same Web service user identity is propagated to the EJB. The EJB application can access the user identity by using standard EJB methods such as `javax.ejb.EjbContext.getCallerPrincipal`.

Further access checks can be performed by using `javax.ejb.EjbContext.isCallerInRole`.

Note: The EJB application must be configured with J2EE security. Refer to "EJB Security Configuration" in the *Oracle Containers for J2EE Security Guide* for more information.

Ant Tasks and WebServicesAssembler

The WebServicesAssembler tool assists in assembling OracleAS Web Services. It allows you to generate the artifacts required to develop and deploy Web services, regardless of whether you are creating the service top down (from a WSDL) or bottom up from Java classes, EJBs, JMS destinations, or database resources. The WebServicesAssembler tool can also be invoked to create Web service client objects based on a WSDL.

The WebServicesAssembler tool can be invoked either on the command line or by Ant tasks. The WebServicesAssembler tool allows you flexibility in how you assemble a Web service. You can break the assembly process into a number of steps that let you more closely control how the Web service is created.

The following list provides a summary of the tasks that you can perform with the WebServicesAssembler tool. The "Using WebServicesAssembler" chapter in the *Oracle Application Server Web Services Developer's Guide* provides detailed information on how to use WebServicesAssembler commands to perform each task.

- Web service assembly—assemble Web services. These commands create all of the files necessary to create a deployable archive such as a WAR, an EAR, or an EJB JAR.
- WSDL management—perform actions on a WSDL, such as generate a WSDL for bottom-up development, manage its contents and location, and determine whether it can be processed by WebServicesAssembler.
- Java generation—generate code to create a Java interface from a WSDL, a proxy/stub, or JAX-RPC value type classes.
- Deployment descriptor generation—generate deployment descriptors for EARs, WARs, or EJB JARs.
- Information—return a short description of WebServicesAssembler commands and the version number of the tool.

Getting an Authenticated User Identity in a Web Service Application

On the server, you can obtain the name of an authorized user from the username (plaintext or digest), SAML, or X.509 security tokens. This information can be used for further validation checks before the user is allowed to access additional server resources. This section describes how you can obtain a user identity with the `AccessControlContext` and `ServiceLifeCycle` APIs.

- Getting an Authenticated Subject with the AccessControlContext API
- Getting an Authenticated Principal with the ServiceLifeCycle API

Getting an Authenticated Subject with the AccessControlContext API

You can use methods from the `java.security.AccessControlContext`, `java.security.AccessController`, and `javax.security.auth.Subject` classes to get the authenticated subject.

Note: One of the purposes of getting the authenticated subject is to perform JAAS Provider authorization. "Performing JAAS Provider Authorization on a Web Service" on page 4-23 provides a summary of how to perform this type of authorization on a Web service.

The following general steps describe how to get the authenticated subject from an `AccessController` object.

1. Create an `AccessControlContext` object by calling the `AccessController.getContext` method. The `getContext` method takes a "snapshot" of the current calling context and returns it in an `AccessControlContext` object.
2. Get the subject of the context by using the `Subject.getSubject` method.

For more information on the classes and methods described in this section, see the API for the `java.security` and `javax.security.auth` packages at the following Web address.

`http://java.sun.com/j2se/1.4.2/docs/api/index.html`

Example 4–12 illustrates using the `AccessControlContext`, `AccessController`, and `Subject` APIs to get the name of an authorized user.

Example 4–12 Obtaining the Name of an Authenticated User with the AccessControlContext API

```
java.security.AccessControlContext context =
          java.security.AccessController.getContext();
javax.security.auth.Subject sub = javax.security.Subject.getSubject(context);
```

Getting an Authenticated Principal with the ServiceLifeCycle API

You can use the `javax.xml.rpc.server.ServiceLifecycle` and `ServletEndpointContext` interfaces to obtain user information from the SAML, username, or X.509 security tokens. For example, the `ServletEndpointContext.getUserPrincipal` method returns the name of the authenticated user; the `ServletEndpointContext.isUserInRole` method returns whether the user belongs to a specific role.

For more information on the `ServiceLifecycle` and `ServletEndpointContext` interfaces, see the API for the `javax.xml.rpc.server` package at the following Web address.

`http://java.sun.com/j2ee/1.4/docs/api/index.html`

To obtain user information from the Web service, follow these general steps:

1. Implement the `ServiceLifecycle` interface for the Web service.
2. In the implementation of the `ServiceLifecycle.init` method, cast the `context` parameter to `ServletEndpointContext`.
3. Call methods, such as `getUserPrincipal` and `isUserInRole`, on the `ServletEndpointContext context` parameter to obtain user information.

Example 4–13 illustrates how to obtain the name of an authenticated user from a secure Web service. The `X509Service` class in the example implements the `ServiceLifecycle` interface. In the `init` method implementation, the `context` parameter is cast to the `ServletEndpointContext` interface. The

`getUserPrincipal` method returns access to the authenticated user name, which is obtained by a call to `getName`.

Example 4–13 Obtaining the Name of an Authenticated User with the ServiceLifeCycle API

```
package test;

public class SecureService implements securePort,
javax.xml.rpc.server.ServiceLifecycle {

 private javax.xml.rpc.server.ServletEndpointContext context;

    public void init(Object obj) throws javax.xml.rpc.ServiceException {
        context = (javax.xml.rpc.server.ServletEndpointContext)obj;
    }

    public String helloUser(String message) throws java.rmi.RemoteException {
        java.security.Principal principal = context.getUserPrincipal();
        if(principal == null) {
            throw new RuntimeException("Principal not found");
        }
        String userName = principal.getName();
        return "Hi "+userName+"! "+message;
    }
    public void destroy() {
        context = null;
    }
}
```

Performing JAAS Provider Authorization on a Web Service

OracleAS Web Services includes OracleAS JAAS Provider, a highly scalable Java Authentication and Authorization Service (JAAS) provider. OracleAS Web Services can protect resources using JAAS authorization for enforcing fine-grained access control over protected resources.

The following general steps describe how to perform JAAS authorization on a Web service.

1. Get the authenticated subject from the security token.

   See "Getting an Authenticated Subject with the AccessControlContext API" on page 4-21 for more information on obtaining the authenticated subject.

2. Use the authenticated subject to issue JAAS authentication calls, such as `checkPermission`.

For more information on JAAS Provider authorization, see the *Oracle Containers for J2EE Security Guide*.

WS-Security and XML APIs

Oracle Security Developer Tools (OSDT) provide you with the cryptographic building blocks necessary for developing robust security applications ranging from basic tasks like secure messaging to more complex projects such as securely implementing a service-oriented architecture. The tools build upon the core foundations of cryptography, public key infrastructure, Web services security and federated identity management.

You can find the Reference Guide and Javadocs for OSDT at the following Web address.

http://download-east.oracle.com/docs/cd/B14099_15/idmanage.htm

Development Decisions

Chapter 5, "Secure Web Service Usage Scenarios", presents a number of use cases that describe the different ways in which you can integrate security into a Web service.

5

Secure Web Service Usage Scenarios

This chapter describes common scenarios for using Web service Security. It begins with the simplest use case, then proceeds through increasingly more complex use cases. The first section of the chapter discusses use cases with no security implications; these are then modified to add security features. This chapter is divided into the following sections:

Non-Secured Web Services

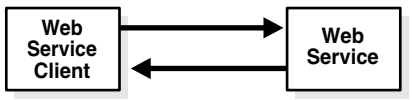
This section contains simple use cases that do not use WS-Security. These use cases serve as a basis to demonstrate adding security features.

For information on how to implement non-secured Web services see the *Oracle Application Server Web Services Developer's Guide*.

Basic Web Service

Figure 5–1 illustrates a basic Web service use case, where a Web service client invokes a Web service:

Figure 5–1 Basic Web Service Use Case

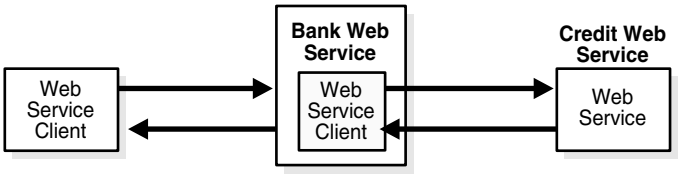


This Web service offers real-time stock quotes to its clients. The Web service client supplies a stock ticker symbol, for example "ORCL", and expects the current value of the stock as the response.

Complex Business Process

Figure 5–2 illustrates a complex workflow, a bank's Web service that allows users to apply for an auto loan.

Figure 5–2 Complex Business Process

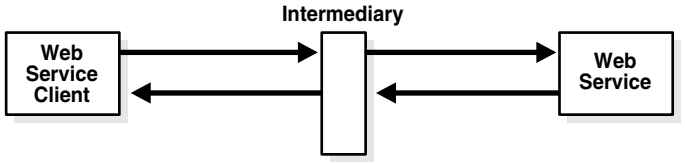


The Bank Web service receives an applicant's application from a Web service client, then invokes a Web service to make an inquiry to a credit bureau. The Bank Web service contains a Web service client that invokes the credit bureau Web service. The bureau sends a response back to the Bank Web service; this response might be an approval or no-approval string and the credit score of the applicant.

Intermediary

Figure 5–3 illustrates an intermediary that separates a Web service client and a Web service. This intermediary could be an external notary service, an XML firewall, a BPEL process manager, or a quality of service (QOS) agent.

Figure 5–3 Intermediary Use Case

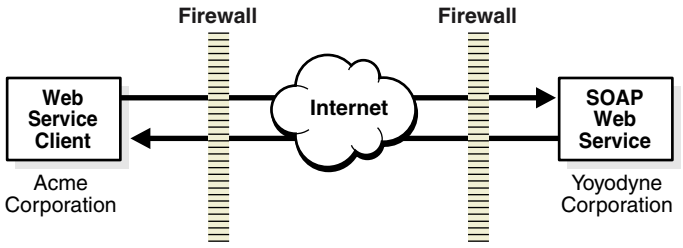


A typical intermediary does not modify the contents of the SOAP body, but adds or modifies SOAP headers. For example, an XML firewall might validate content, authenticate and authorize users, but would not process the SOAP body, because the intermediary is not the Web service's final destination.

Federated

The federated use case is similar to the basic use case, except that it spans security boundaries. Figure 5–4 shows a company that allows its customers to submit purchase orders through a Web service; note that the company exposing the Web service does not own or control the Web service client.

Figure 5–4 Federated Use Case



HTTP-Based Security

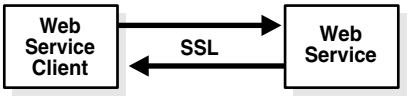
The traditional method of securing data, such as SOAP messages, transmitted over HTTP is to use the Secure Sockets Layer and HTTP authentication.

- Secure Sockets Layer
- HTTP Basic Authentication and Digest Authentication

Secure Sockets Layer

The Secure Sockets Layer (SSL) prevents intermediate applications from reading a message transmitted between a client and a server. SSL uses public-key encryption to exchange a session key between the client and server; this session key is used to encrypt the HTTP request and response traffic. SSL is the most commonly used HTTP-based security solution. Figure 5–5 illustrates the flow of data when using SSL encryption.

Figure 5–5 SSL Encryption In a Web Service



For the steps to configure SSL, see the *Oracle Containers for J2EE Security Guide*.

> **Note:** Oracle Application Server Web Services also supports TLS encryption; for details, see the *Oracle Database Advanced Security Administrator's Guide*.

HTTP Basic Authentication and Digest Authentication

Business requirements often dictate that access to Web services be restricted to authenticated users. The following sections describe the two most common approaches.

- Basic Authentication
- Digest Authentication

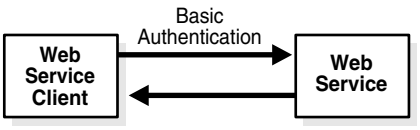
For information on how to implement basic and digest authentication, see the *Oracle Containers for J2EE Security Guide*.

Basic Authentication

When using basic authentication, the Web service client authenticates itself to the service by sending a base-64 encoded user name and password in an HTTP

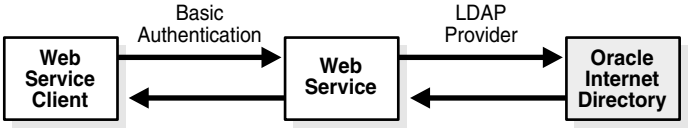
authorization header. Basic authentication has a security weakness: it sends credentials using base-64 encoding, which is trivial to decode. Sending credentials in a base-64 encoding is nearly as insecure as sending them in the clear. Figure 5–6 illustrates the flow of data when using basic authentication.

Figure 5–6 HTTP Basic Authentication in a Web Service



When a Web service validates an authentication request, it must do so against a credential store. In OracleAS Web Services applications, authentication requests are validated through the LDAP security provider. For example, your Web service could be configured to use an LDAP provider.

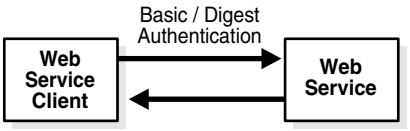
Figure 5–7 Basic Authentication With LDAP



Digest Authentication

Digest authentication uses the same HTTP headers as basic authentication, but it transmits a digest for the password. Although digest authentication is much more secure than basic authentication, it is not interoperable in all security environments, so is rarely used. Also, digest authentication uses weak algorithms such as MD5. Encrypting transmissions using SSL is the preferred method to prevent credential snooping.

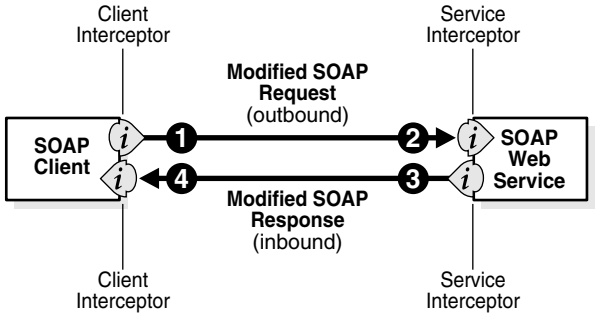
Figure 5–8 Digest Authentication



WS-Security

Oracle provides pre-built JAX-RPC handlers called *interceptors* to provide WS-Security message-level security. For outbound messages, these interceptors add the necessary WS-Security headers to support WS-Security authentication, XML encryption, and XML digital signature operations; for inbound messages, the interceptors process the corresponding header information.

Figure 5–9 Interceptor Framework



Note: An application can use SSL alone, WS-Security alone, or both together.

Web Services Security Authentication

WS-Security authentication is more flexible than either basic or digest authentication. WS-Security provides three different authentication profiles to authenticate against a Web service:

- Username Token Profile
- X.509 Token Profile
- SAML Token Profile

Each of these profiles defines how to use its token type within the WS-Security specification. Oracle provides a pre-built interceptor for all the WS-Security Token profiles.

Username Token Profile

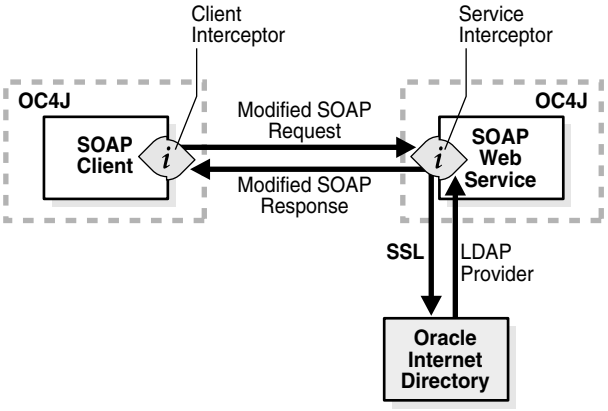
The Username Token Profile is similar to HTTP basic and digest authentication. Applications can construct a username token with three different credential types:

- username without password
- username with password
- username with encrypted password

The username token interceptor adds the token to the SOAP envelope.

If hosted on OracleAS Web Services, the receiving Web service also has a username token interceptor. The interceptor processes the username token and validates the credentials contained within it against the Web service's configured security provider, such as the Oracle Internet Directory LDAP-based provider.

Figure 5–10 Username Token Use Case



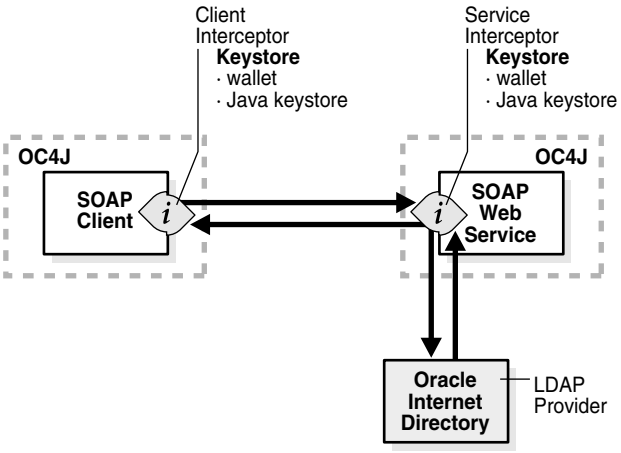
The username token use case is a common use case for OracleAS Portal and other Web-based applications; its advantage is that all credential checks are made against a central user repository.

For information on how to configure the username token, see "Using a Username Token" on page 3-7.

X.509 Token Profile

The X.509 Profile uses an X.509 certificate to authenticate against the receiving Web service. Oracle provides a pre-built interceptor that adds WS-Security X.509 authentication to outbound messages. A certificate token is added to the WS-Security header in the outgoing SOAP envelope. The certificate is read from a credential store such as an Oracle Wallet or a Java keystore.

Figure 5–11 X.509 Token Use Case



Under OracleAS Web Services, the interceptor on the receiving Web service validates the X.509 certificate's signature and then checks to see if the user exists in the directory. The distinguished name in the certificate must have a match in the directory. If the receiving Web service is an external J2EE container or .NET Web service, it provides its own means of validating the WS-Security X.509 certificate.

For information on how to configure the X.509 token, see "Using an X.509 Token" on page 3-15.

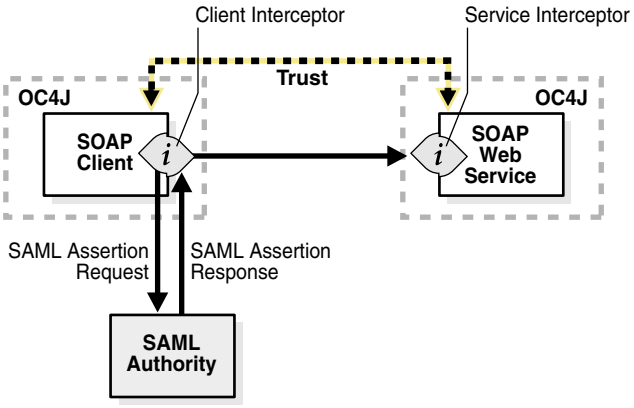
SAML Token Profile

SAML tokens, unlike Username and X.509, assert that the SAML user or subject has already been authenticated. SAML security tokens are composed of *assertions*: one or more statements about a user. SAML assertions are attached to SOAP messages using WS-Security by placing assertion elements inside the header. Oracle supports two SAML use cases: sender vouches and holder of key.

For information on how to configure the SAML token for these use cases, see "Using a SAML Token" on page 3-20.

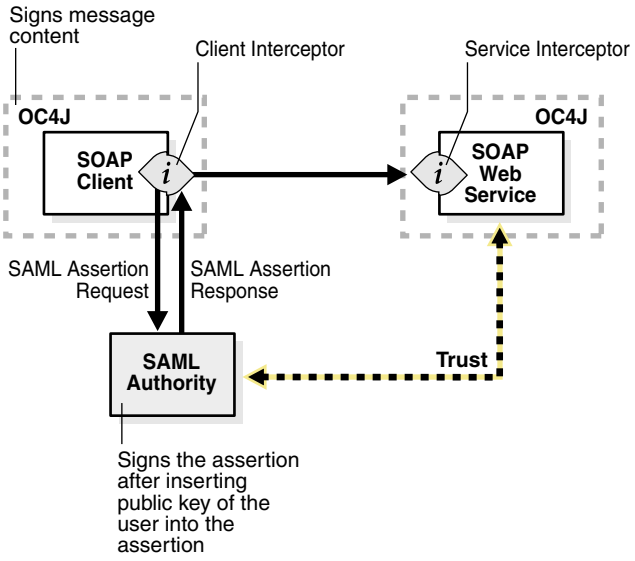
Sender Vouches In the sender vouches use case, the sender vouches for the verification of the assertion's subject. The sender's private key is used to sign both the assertion and the message body. If the sender gets the assertion from a SAML authority, then the SAML authority could also sign the message. In this scenario double signing of the assertion may occur.

Figure 5–12 SAML Assertion Use Case



Holder of Key In the holder of key use case, the sender submits a request for an assertion to the SAML authority. The issuer then returns a signed assertion with user's public key embedded in it. The sender then signs the message body with their private key.

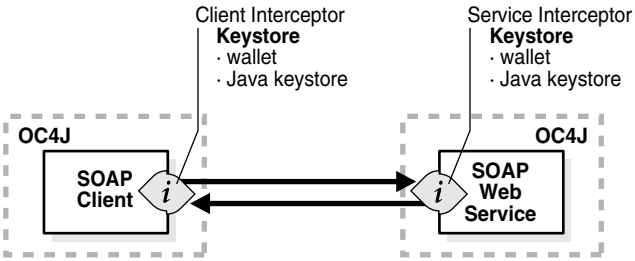
Figure 5–13 SAML Authority Assertion Use Case



XML Signature

XML signatures ensure message integrity for SOAP messages. When you use XML signatures, outbound message bodies are digitally signed with the sender's private key. The receiver uses the sender's public key to validate the signature.

Figure 5–14 Digital Signature Use Case



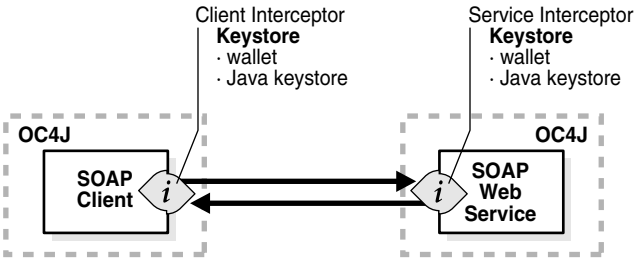
Oracle also supports protecting the integrity of the message response. There are many scenarios for using XML signatures to sign message bodies, message headers, or parts of either. For example, an application could sign the whole message body to officially identify the sender. Another application could sign several of the elements of the body, each element from a different origin and identified by a signature from a different signer. The signature can also be used to validate content integrity, detecting tampering during transit or storage.

For information on how to configure XML signature in a Web service, see "Configuring XML Signature" on page 3-38.

XML Encryption

XML encryption allows the encryption of an XML element, which could be the SOAP body, the SOAP header, or any XML element within these structures. The interceptor substitutes an encrypted element for the original XML data; the receiver is responsible for decrypting the element.

Figure 5–15 XML Encryption Use Case



XML encryption is used for targeted encryption. An application could encrypt an entire message body or encrypt only sub-elements within a message body. XML encryption does not replace SSL; the two can be used together or separately. For example, a SOAP payload may contain a credit check that uses XML encryption only on the elements that contain the credit card account numbers. It may be that this information isn't encrypted until reaching a final destination in a multi-step process. Therefore the entire message is sent over SSL to protect it while in transit.

To support encryption, the interceptor must be able to access a Java keystore or an Oracle Wallet. For XML encryption, the Web service client uses the Web service's public key. The Web service decrypts any XML encrypted elements sent by the client by using its private key.

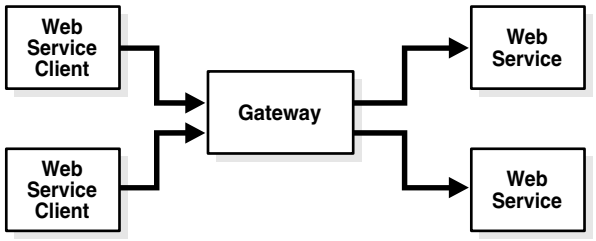
For information on how to configure XML encryption in a Web service, see "Configuring XML Encryption" on page 3-34.

Gateways

Some applications use gateways (centralized policy enforcement points) to implement message security, instead of applying security within a J2EE container. Gateways can be useful as checkpoints that all Web services must traverse before accessing external Web services. For example, there could be a gateway that separates a company's Web service clients and external Web services. This gateway would enforce WS-Security policies on all outbound Web services.

The gateway shown in Figure 5–6 ensures that all outbound messages are signed.

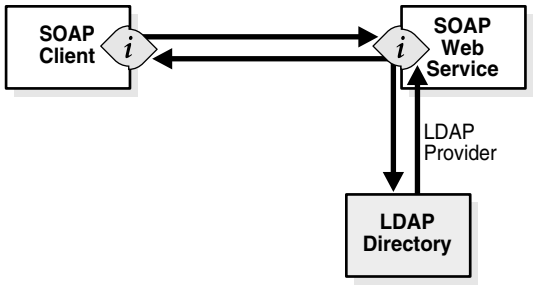
Figure 5–16 Gateway Use Case



Identity Management

Web services often authenticate and authorize against an Identity Management infrastructure. In OracleAS Web Services, the interceptors use the Identity Management security provider configuration to determine where to authenticate. For example, if the provider is the LDAP provider, then it will authenticate against an LDAP directory.

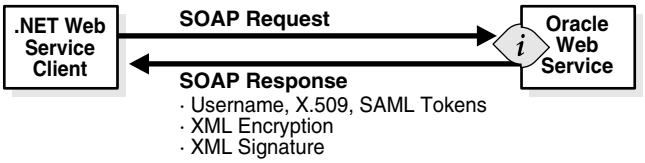
Figure 5–17 Identity Management Use Case



Interoperability

Many business situations require a mixed-vendor environment, in which, for instance, Oracle web service clients interact with .NET Web services.

Figure 5–18 Interoperability Use Case



There are a number of use cases for interoperability between vendors to consider with Web services. Interoperability between vendors is covered by the Basic Security Profile (BSP) specification which is available from the Web Services Interoperability (WS-I) organization (http://www.ws-i.org/).

6

Troubleshooting

This chapter describes solutions to some of the errors you might encounter when working with Oracle Application Server Web Services Security. The errors are divided into these categories.

- General Errors
- Keystore-Related Errors
- Message Integrity Errors
- Message Confidentiality Errors
- Authentication Errors

When you are troubleshooting errors that occur on the client side, the important file to look at is the `<generated_name>_Stub.xml` deployment descriptor file. For errors that occur on the server side, the important files to check are the `oracle-webservices.xml` deployment descriptor and the `wsmgtm.xml` management configuration file.

Logging Errors

Errors are logged in the `log.xml` file (*ORACLE_HOME*`/j2ee/home/log/oc4j/log.xml`). To log the errors described in this appendix, turn logging level to `TRACE:32` in the `j2ee-logging.xml` logging configuration file.

General Errors

Missing <wsse:Security> in SOAP header

This error message is thrown when the incoming message is missing the security header. Check that the client outbound policy is present in `<generated_name>_Stub.xml` client-side deployment descriptor and the server outbound policy is present in the `oracle-webservices.xml` server-side deployment descriptor or the `wsmgmt.xml` file.

If the outbound policy (security configuration) does not exist, then check the Web service description and the port name. If you configured security only on the operation level, then make sure that the `<runtime enabled="security"/>` element is set at the port level.

SOAP must understand error

The `enabled="security"` attribute may be missing. Check whether the `enabled="security"` attribute is present in the `oracle-webservices.xml` file and in the client side management file. If it is present in `oracle-webservices.xml` then check it in the `wsmgmt.xml` file.

Unable to fetch realm

The realm name may be incorrect or missing. Check the default realm name in *ORACLE_HOME*`/j2ee/home/config/jazn.xml` file. If you have included your own `orion-application.xml` file, then check the realm name in the `<jazn>` element. The realm name must be a valid realm and must exist in the XML/LDAP repository base on the JAZN provider.

Keystore-Related Errors

Could not resolve subject key identifier

Make sure that the keystore contains the signer's public key certificate. For example, if the client side keystore has a signature key with the alias `xyz`, then the server side keystore must have the public key certificate corresponding to the alias `xyz`.

See "Using Keystores" on page 3-1 for information on importing certificates into the keytool or Oracle Wallet.

Exception when validating the signer certificate path to the trusted root

Import the signer's certificate which contains the matching public key (which corresponds to signer's private key) and trusted CA certificates into the keystore. If the certificates are chained, you must import all of the certificates in the chain.

See "Using Keystores" on page 3-1 for information on importing certificates into the keytool or Oracle Wallet.

Certificate for encryption not found

The `<recipient-key>` element may be missing or may contain an incorrect value. Check the `<recipient-key>` element is present in the outbound `<encrypt>` element. It must point to a valid key; the key must be specified in the `<key-store>` element's `path` attribute. If you are using the `<use-request-cert>` element for encrypting the response back to the client using the client's public key, make sure that the outbound policy for the client is configured with `<x509-token/>`.

For information on the `<recipient-key>` element, see "Encryption Elements for Outbound Messages" on page 2-17. For information on the `<use-request-cert>` element, see "Encrypting a Message with a Signature Key" on page 3-37.

No key/certificate exists for alias <*some_alias*>

Check that a private key or certificate with the specified alias is present in the keystore.

See "Using Keystores" on page 3-1 for information on adding keys and certificates to the keystore or Oracle Wallet.

Invalid recipient-key alias

Check that you have included a public key with the correct recipient key alias in the keystore.

See "Using Keystores" on page 3-1 for information on adding keys and certificates to the keystore or Oracle Wallet.

Invalid keystore path

Check the value of the `<key-store>` element's `path` attribute is correct.

For the client side, the keystore path must be absolute. If the client application is deployed in the OC4J container, then it can be relative to *ORACLE_HOME*`/j2ee/home`.

For the server side port-level keystore, the path must be relative to the application's root directory. For example, if you have deployed an application `test-application`, the path to the keystore must be relative to the ORACLE_HOME`/j2ee/instance/applications/test-application/` directory.

For the server side global-level keystore, the path must be relative to the *ORACLE_HOME*`/j2ee/home/config` directory.

If you are using the `oracle.security.jazn.config` system property to point to the `config` directory, then the server side global keystore path must be relative to the directory specified in this property. Note that this property is valid only for the server side global-level keystore path.

See "Keystore Elements" on page 2-5 for more information on the `<key-store>` element.

Error reading keystore data

Check the value of the `<key-store>` element's `password` attribute. Either the password is incorrect or some one has tampered with the keystore.

See "Keystore Elements" on page 2-5 for more information on the `password` attribute.

Error getting certificate chain with alias <some_alias>

Check that the certificate chain is present in the keystore. Use the `keytool -list` command to view your keystore.

See "Using Keystores" on page 3-1 for more information on keytool commands.

Error getting trusted certificates

Check that you have added all the trusted CA certificates to the keystore. Use the `keytool -list` command to view your keystore.

See "Using Keystores" on page 3-1 for more information on keytool commands.

Message Integrity Errors

Element with specific namespace and local part must be signed

In the outbound `<signature>` element, make sure that the values for the `name-space` and `local-part` attributes of `<tbs-elements>` are correct. These values must match the `name-space` and the `local-part` values for the element expected to be signed in the inbound `<verify-signature>` element.

For more information on these elements, see "Signature Verification Elements for Inbound Messages" on page 2-9 and "Signature Elements for Outbound Messages" on page 2-16.

Element with specific namespace and local part not found

In the outbound <signature> element, make sure that the values for the name-space and local-part attributes of <tbs-elements> are present and correct.

For more information on these elements and attributes, see"Signature Elements for Outbound Messages" on page 2-16.

Missing created or Missing timestamp

The outbound <signature> element must contain an <add-timestamp> element with its created attribute set to true. For example:

```
<signature>
   <add-timestamp created="true" expiry="28800"/>
</signature>
```

See "Signature Elements for Outbound Messages" on page 2-16 for more information about the <add-timestamp> element.

Timestamp expired

This indicates that there may be a mis-match between the clock times on the machines on which the client and Web service application run. Set the clock-skew property in the inbound <verify-signature> element to adjust the time difference between the machines. The value is in milliseconds. For example:

```
<verify-signature>
   <property name="clock-skew" value="5000"/>
</ verify-signature>
```

For more information on using clock skew, see "Adjusting the Clock Skew Between a Client and a Web Service Application" on page 3-42.

Invalid timestamp

This indicates that there may be a mis-match between the clock times on the machines on which the client and Web service application run. Check the value of the timestamp set in the request, and adjust the setting of the clock-skew property in the inbound <verify-signature> element.

The value of the timestamp in the request message must be earlier than the value (message arrival time – clock skew).

For more information on using clock skew, see "Adjusting the Clock Skew Between a Client and a Web Service Application" on page 3-42.

Policy requires integrity

Check that the outbound <signature> element contains a valid configuration. It must also match the inbound <verify-signature> configuration (if it is present).

For more information on these elements, see "Signature Verification Elements for Inbound Messages" on page 2-9 and "Signature Elements for Outbound Messages" on page 2-16.

Element not found for signing

In the outbound policy <signature> element, check that the values of the name-space and local-part attributes of <tbs-elements> are valid.

For more information on these elements, see "Signature Elements for Outbound Messages" on page 2-16.

Signature certificate missing Subject Key Identifier

Check the signature key that you have configured for signing. The public key certificate must have a Subject Key Identifier extension.

Cannot use certificate for KEY_ENCIPHERMENT

Check that the recipient-key (or client signature key in the case of `use-request-cert`) has a key usage extension with value key encipherment.

Signature key/certificate not found

Check that the `<signature-key>` element has been configured with a valid alias and password. The private key or certificate must be present in the keystore configured in the `<key-store>` tag.

For more information on these elements, see "Keystore Elements" on page 2-5 and "Signature and Encryption Key Elements" on page 2-6.

Invalid <signature-methods/> configured must have at least one signature algorithm

The `<signature-methods>` element in the inbound `<verify-signature>` element is optional. It should be omitted if no signature algorithms are being specified. If the `<signature-methods>` element is present, then it must have at least one valid signature algorithm. Acceptable signature algorithms are `RSA-SHA1` (default), `RSA-MD5`, and `DSA-SHA1`.

For more information on the `<signature-methods>` element, see "Signature Verification Elements for Inbound Messages" on page 2-9.

Message Confidentiality Errors

Encryption key not found at Port/Global Level

You will see this error message if the inbound policy is configured to `<decrypt>` and the encryption key is missing at port/global level. Check the `<encryption-key>` element and the `<key-store>` element. Make sure that the values for the `alias` and `password` attributes are correct.

For more information on these elements, see "Keystore Elements" on page 2-5 and "Signature and Encryption Key Elements" on page 2-6.

URI content must be encrypted

In the outbound `<encrypt>` element, check that the `<tbe-element>` subelement in the `<tbe-elements>` element has the correct `name-space` and `local-part` attribute values.

For more information on the `<tbe-element>` subelement, see "Encryption Elements for Outbound Messages" on page 2-17.

Policy requires confidentiality

Check that the outbound `<encrypt>` element has a valid configuration and matches the configuration of the inbound `<decrypt>` element.

For more information about these elements, see "Decryption Elements for Inbound Messages" on page 2-10 and "Encryption Elements for Outbound Messages" on page 2-17.

Invalid <keytransport-methods/> configured. Must have at least one algorithm

The `<keytransport-methods>` element in the inbound `<decrypt>` element is optional and should be omitted if no key transport methods are being specified. If the `<keytransport-methods>` element is present, then it must have at least one valid key transport algorithm. Acceptable key transport algorithms are `RSA-1_5` (default) or `RSA-OAEP-MGF1P`.

For more information on the `<keytransport-methods>` element, see "Decryption Elements for Inbound Messages" on page 2-10.

Invalid <encryption-methods/> configured. Must have at least one algorithm.

The `<encryption-methods>` element in the inbound `<decrypt>` element is optional and should be omitted if no encryption methods are being specified. If the `<encryption-methods>` element is present, then it must have at least one valid encryption algorithm. Acceptable encryption algorithms are `3DES`, `AES-128` (default), and `AES-192`.

For more information on the `<encryption-methods>` element, see "Decryption Elements for Inbound Messages" on page 2-10.

Authentication Errors

Invalid security token specified

An "Invalid security token specified" or similar error message is returned when attempting to invoke the Web service.

There could be several reasons for this error:

- For a user name token with plain text password authentication:
  - An incorrect user name was entered.
  - An incorrect password was entered.
  - An incorrect realm <*realm name*>/<*user name*> was entered. Check whether the realm name and user name are correct. Check whether the format is correct. For example, the format requires a forward slash (/), not a double slash (//) or back slash (\).
  - An incorrect value for the `password-type` attribute was entered. The value should be `PLAINTEXT` (one word). Check that it was not entered as two words or misspelled.
- For a user name token with digest password authentication:
  - Check all of the reasons listed under user name token with plain text password authentication (above).
  - An incorrect value for the password-type attribute was entered. The value should be `DIGEST`.
  - Check the values for the `add-nonce` and `add-created` attributes. For digest password authentication, these attributes should always be set to `true`.
- For X.509 token authentication:

  By default, the service expects `RSA-SHA1` as the certificate's key encryption algorithm. If DSA was used as the certificate's key encryption algorithm instead,

then check that it was correctly configured. The server and client side configuration should explicitly specify DSA as the signature method.

- For SAML token authentication:

  An incorrect user name was entered.

Policy requires authentication token

The outbound policy must have one of the authentication tokens expected by the server. For example, if your inbound policy contains `<verify-x509-token>` and `<verify-username-token>` then your outbound policy must have either `<username-token>` or `<x509-token>`.

> **Note:** Only one authentication token can be included in the outbound policy. The inbound policy can have more than one authentication token.

Encryption element not found

In the outbound `<encrypt>` element, check that the `<tbe-element>` subelement in the `<tbe-elements>` element has valid `name-space` and `local-part` attribute values.

For more information on these attributes, see "Encryption Elements for Outbound Messages" on page 2-17.

Invalid subject-confirmation method

Check that the subject `<confirmation-method>` subelement in the `<saml-token>` element contains a valid value. The acceptable confirmation methods are `SENDER-VOUCHES` (default), `SENDER-VOUCHES-UNSIGNED`, and `HOLDER-OF-KEY`.

For more information on the `<confirmation-method>` subelement, see "SAML Token Elements for Outbound Messages" on page 2-14.

Cannot find X509 token

Check that the configuration for the `<signature-key>` element contains valid values for the `alias` and `password` attributes.

The private key with the `alias` value must exist in the keystore.

For more information on checking the keystore contents, see"Using Keystores" on page 3-1. For more information on the `<signature-key>` element, see "Signature and Encryption Key Elements" on page 2-6.

No username found

Check that the `<username-token>` element in the outbound policy has a valid `name` attribute. If you are setting the user name with `Stub.USERNAME_PROPERTY`, then check that the value that is set by this property is correct. If you are using a callback handler, then check the value set in the `Namecallback`.

For more information about the `<username-token>` element, see "Username Token Elements for Outbound Messages" on page 2-12. For more information about using Stub properties and callback handlers, see "How to Configure the Username Token for the Client Side" on page 3-9.

Invalid assertion, missing public key

Check the definition of the holder-of-key assertion. The assertion must contain the user's public key in the subject.

Cannot authenticate user or invalid token

The primary reason for this error is that the user cannot be found in the repository. Depending on your provider, different attributes are used to map the user. Check that the mapping attribute and the realm name is correct in the `jazn.xml` file. Also, if the provider is XML, then check the location of the `jazn-data.xml` file specified in `orion-application.xml`. By default, this will be `system-jazn-data.xml` and the user must be present in this repository. If there is a local `jazn-data.xml` file, then the user must be present in this file. By default, for an XML provider, the mapping attribute is `CN`.

A

Schemas

This appendix lists the Oracle Web Services Security schema.

Oracle Web Services Security Schema Listing

Example A–1 illustrates the contents of the `oracle-webservices-security-10_0.xsd` schema file.

Example A–1 Contents of the oracle-webservices-security-10_0.xsd Security Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="password-type-enum">
    <xsd:restriction base="xsd:string">
      <xsd:whiteSpace value="collapse"/>
      <xsd:enumeration value="PLAINTEXT"/>
      <xsd:enumeration value="DIGEST"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="signature-method-enum">
    <xsd:restriction base="xsd:string">
      <xsd:whiteSpace value="collapse"/>
      <xsd:enumeration value="RSA-SHA1"/>
      <xsd:enumeration value="RSA-MD5"/>
      <xsd:enumeration value="DSA-SHA1"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="encryption-method-enum">
    <xsd:restriction base="xsd:string">
      <xsd:whiteSpace value="collapse"/>
      <xsd:enumeration value="3DES"/>
      <xsd:enumeration value="AES-128"/>
      <xsd:enumeration value="AES-256"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="keytransport-method-enum">
    <xsd:restriction base="xsd:string">
      <xsd:whiteSpace value="collapse"/>
      <xsd:enumeration value="RSA-OAEP-MGF1P"/>
      <xsd:enumeration value="RSA-1_5"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="encrypt-mode-enum">
    <xsd:restriction base="xsd:string">
      <xsd:whiteSpace value="collapse"/>
```

```
      <xsd:enumeration value="CONTENT"/>
      <xsd:enumeration value="ELEMENT"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="nonce-config-type">
    <xsd:attribute name="clock-skew" type="xsd:integer" default="0"/>
    <xsd:attribute name="cache-ttl" type="xsd:integer" default="300"/>
  </xsd:complexType>
  <xsd:complexType name="security-config-type">
    <xsd:sequence>
      <xsd:element name="key-store" type="key-store-config-type" minOccurs="0"/>
      <xsd:element name="signature-key" type="key-config-type" minOccurs="0"/>
      <xsd:element name="encryption-key" type="key-config-type" minOccurs="0"/>
      <xsd:element name="nonce-config" type="nonce-config-type" minOccurs="0"/>
      <xsd:element name="inbound" type="inbound-config-type" minOccurs="0"/>
      <xsd:element name="outbound" type="outbound-config-type" minOccurs="0"/>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="security-operation-config-type">
    <xsd:sequence>
      <xsd:element name="inbound" type="inbound-config-type" minOccurs="0"/>
      <xsd:element name="outbound" type="outbound-config-type" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="security-global" type="security-config-type"/>
  <xsd:element name="security-port" type="security-config-type"/>
  <xsd:element name="security-operation" type="security-operation-config-type"/>
  <xsd:complexType name="inbound-config-type">
    <xsd:sequence>
      <xsd:element name="verify-username-token"
type="verify-username-token-config-type" minOccurs="0"/>
      <xsd:element name="verify-x509-token" type="verify-x509-token-config-type"
minOccurs="0"/>
      <xsd:element name="verify-saml-token" type="verify-saml-token-config-type"
minOccurs="0"/>
      <xsd:element name="verify-signature" type="verify-signature-config-type"
minOccurs="0"/>
      <xsd:element name="decrypt" type="decrypt-config-type" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="outbound-config-type">
    <xsd:sequence>
      <xsd:choice minOccurs="0">
        <xsd:element name="username-token" type="username-token-config-type"/>
        <xsd:element name="x509-token" type="x509-token-config-type"/>
        <xsd:element name="saml-token" type="saml-token-config-type"/>
      </xsd:choice>
      <xsd:element name="signature" type="signature-config-type" minOccurs="0"/>
      <xsd:element name="encrypt" type="encrypt-config-type" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="signature-config-type">
    <xsd:sequence>
      <xsd:element name="signature-method" default="RSA-SHA1" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="RSA-SHA1"/>
```

```
            <xsd:enumeration value="RSA-MD5"/>
            <xsd:enumeration value="DSA-SHA1"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="tbs-elements" type="sign-elements-config-type"
minOccurs="0"/>
      <xsd:element name="add-timestamp" type="timestamp-config-type"
minOccurs="0"/>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="encrypt-config-type">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="recipient-key" type="key-config-type"/>
        <xsd:element name="use-request-cert" type="xsd:boolean"/>
      </xsd:choice>
      <xsd:element name="encryption-method" default="AES-128" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="3DES"/>
            <xsd:enumeration value="AES-128"/>
            <xsd:enumeration value="AES-256"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="keytransport-method" default="RSA-1_5" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="RSA-OAEP-MGF1P"/>
            <xsd:enumeration value="RSA-1_5"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="tbe-elements" type="encrypt-elements-config-type"/>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="verify-signature-config-type">
    <xsd:sequence>
      <xsd:element name="signature-methods" type="signature-methods-config-type"
minOccurs="0"/>
      <xsd:element name="tbs-elements" type="sign-elements-config-type"
minOccurs="0"/>
      <xsd:element name="verify-timestamp" type="timestamp-config-type"
minOccurs="0"/>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="decrypt-config-type">
    <xsd:sequence>
      <xsd:element name="encryption-methods" type="encryption-methods-config-type"
minOccurs="0"/>
      <xsd:element name="keytransport-methods"
```

```
type="keytransport-methods-config-type" minOccurs="0"/>
      <xsd:element name="tbe-elements" type="encrypt-elements-config-type"
minOccurs="0"/>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="property-config-type">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
  </xsd:complexType>
  <xsd:complexType name="key-store-config-type">
    <xsd:attribute name="store-pass" type="xsd:string" use="optional"/>
    <xsd:attribute name="path" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="optional"/>
    <xsd:attribute name="name" type="xsd:string" use="optional"/>
  </xsd:complexType>
  <xsd:complexType name="key-config-type">
    <xsd:attribute name="alias" type="xsd:string" use="required"/>
    <xsd:attribute name="key-pass" type="xsd:string" use="optional"/>
  </xsd:complexType>
  <xsd:complexType name="signature-methods-config-type">
    <xsd:sequence>
      <xsd:element name="signature-method" default="RSA-SHA1" maxOccurs="3">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="RSA-SHA1"/>
            <xsd:enumeration value="RSA-MD5"/>
            <xsd:enumeration value="DSA-SHA1"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="encryption-methods-config-type">
    <xsd:sequence>
      <xsd:element name="encryption-method" default="AES-128" maxOccurs="3">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="3DES"/>
            <xsd:enumeration value="AES-128"/>
            <xsd:enumeration value="AES-256"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="keytransport-methods-config-type">
    <xsd:sequence>
      <xsd:element name="keytransport-method" default="RSA-1_5" maxOccurs="2">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="RSA-OAEP-MGF1P"/>
            <xsd:enumeration value="RSA-1_5"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
```

```
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="username-token-config-type">
    <xsd:sequence>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="optional"/>
    <xsd:attribute name="password" type="xsd:string" use="optional"/>
    <xsd:attribute name="password-type" type="password-type-enum" use="optional"
default="PLAINTEXT"/>
    <xsd:attribute name="cbhandler-name" type="xsd:string" use="optional"/>
    <xsd:attribute name="add-nonce" type="xsd:boolean" use="optional"
default="false"/>
    <xsd:attribute name="add-created" type="xsd:boolean" use="optional"
default="false"/>
  </xsd:complexType>
  <xsd:complexType name="verify-username-token-config-type">
    <xsd:sequence>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="password-type" type="password-type-enum" use="optional"/>
    <xsd:attribute name="require-nonce" type="xsd:boolean" use="optional"
default="false"/>
    <xsd:attribute name="require-created" type="xsd:boolean" use="optional"
default="false"/>
  </xsd:complexType>
  <xsd:complexType name="verify-x509-token-config-type">
    <xsd:sequence>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="x509-token-config-type">
    <xsd:sequence>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="cbhandler-name" type="xsd:string" use="optional"/>
  </xsd:complexType>
  <xsd:complexType name="timestamp-config-type">
    <xsd:attribute name="expiry" type="xsd:long" default="28800"/>
    <xsd:attribute name="created" type="xsd:boolean" default="true"/>
  </xsd:complexType>
  <xsd:complexType name="encrypt-elements-config-type">
    <xsd:sequence>
      <xsd:element name="tbe-element" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:attribute name="name-space" type="xsd:anyURI" use="required"/>
          <xsd:attribute name="local-part" type="xsd:string" use="required"/>
          <xsd:attribute name="mode" type="encrypt-mode-enum" use="optional"
default="CONTENT"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="sign-elements-config-type">
    <xsd:sequence>
      <xsd:element name="tbs-element" maxOccurs="unbounded">
```

```
        <xsd:complexType>
          <xsd:attribute name="name-space" type="xsd:anyURI" use="required"/>
          <xsd:attribute name="local-part" type="xsd:string" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="saml-token-config-type">
    <xsd:sequence>
      <xsd:element name="subject-confirmation-method"
type="subject-confirmation-method-config-type" minOccurs="0"/>
      <xsd:element name="attribute" type="attribute-config-type" minOccurs="0"/>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="saml-authority" type="saml-authority-config-type"
minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name-format" type="name-identifier-format-enum"
default="UNSPECIFIED"/>
    <xsd:attribute name="name" type="xsd:string" use="optional"/>
    <xsd:attribute name="cbhandler-name" type="xsd:string" use="optional"/>
    <xsd:attribute name="issuer-name" type="xsd:string" use="optional"/>
  </xsd:complexType>
  <xsd:complexType name="verify-saml-token-config-type">
    <xsd:sequence>
      <xsd:element name="subject-confirmation-methods"
type="subject-confirmation-methods-config-type" minOccurs="0"/>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="subject-confirmation-methods-config-type">
    <xsd:sequence>
      <xsd:element name="confirmation-method" default="SENDER-VOUCHES"
maxOccurs="3">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="SENDER-VOUCHES"/>
            <xsd:enumeration value="SENDER-VOUCHES-UNSIGNED"/>
            <xsd:enumeration value="HOLDER-OF-KEY"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="subject-confirmation-method-config-type">
    <xsd:sequence>
      <xsd:element name="confirmation-method" default="SENDER-VOUCHES"
minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"/>
            <xsd:enumeration value="SENDER-VOUCHES"/>
            <xsd:enumeration value="SENDER-VOUCHES-UNSIGNED"/>
            <xsd:enumeration value="HOLDER-OF-KEY"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
```

```
  </xsd:complexType>
  <xsd:complexType name="attribute-config-type">
    <xsd:attribute name="path" type="xsd:string" use="required"/>
  </xsd:complexType>
  <xsd:simpleType name="name-identifier-format-enum">
    <xsd:restriction base="xsd:string">
      <xsd:whiteSpace value="collapse"/>
      <xsd:enumeration value="UNSPECIFIED"/>
      <xsd:enumeration value="EMAIL"/>
      <xsd:enumeration value="X509-SUBJECT-NAME"/>
      <xsd:enumeration value="WINDOWS-DOMAIN-NAME"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="saml-authority-config-type">
    <xsd:sequence>
      <xsd:element name="property" type="property-config-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="endpoint-address" type="xsd:string" use="required"/>
    <xsd:attribute name="auth-user-name" type="xsd:string" use="optional"/>
    <xsd:attribute name="auth-password" type="xsd:string" use="optional"/>
    <xsd:attribute name="require-signature" type="xsd:boolean" use="optional"/>
  </xsd:complexType>
</xsd:schema>
```

B

Security Threats and Solutions

This appendix describes the security threats that are present in today's Web services environment, and how Oracle Application Server Web Services Security responds to these threats. The descriptions of the security threats are provided by the Web Services Interoperability (WS-I) Organization's document *Security Challenges, Threats and Countermeasures Version 1.0*. This document identifies the following information:

- the security challenges: these are the goals or features that help you decide specific security scenarios.
- the threats that prevent the fulfillment of each goal.
- the countermeasures you can employ to guard against each threat.
- the possible usage scenarios and the security challenges and threats that might apply to each.

For more information on these security mechanisms and threats, see *Security Challenges, Threats and Countermeasures Version 1.0* at the following Web address.

`http://www.ws-i.org/Profiles/BasicSecurity/SecurityChallenges-1.0.pdf`.

This appendix identifies how the functionality in OracleAS Web Services can be used to address the threats described in the *Security Challenges* document. For example, Table B–1 describes message-level security threats and Table B–2 describes transport-level security threats. These tables also identify possible solutions to the security threats and whether you can implement the solutions with Application Server Control or JDeveloper. The tables also provide a roadmap to where you can find more information on the solutions in the documentation.

These tables use tags, such as SC1, SA1, and BISP1, to indicate message and transport layer security mechanisms. These tags are briefly described in Table B–3. These tables also use threat IDs, such as T-01 and T-02 to indicate types of security threats. These threats are briefly described in Table B–4.

Table B–1 Message Layer Security Solutions

Solution	Threat Number and Name	Supported Solutions	Application Server Control Support	JDeveloper Support	Where Documented
Sender Authentication Username with clear text password or digest password with encrypted password/digest (SA1)	T-05: Principal Spoofing	SA1	Inbound configuration: (verifying username token) is supported. Outbound configuration: (username token), Application Server Control does not support encrypting or decrypting the username token.	Inbound configuration: (verifying username token) is supported. Outbound configuration: (username token with clear text/digest password) is supported. Encrypting and decrypting the username token are manual steps	"Encrypting Elements of a SOAP Message" on page 3-37 and "Decrypting Elements of a SOAP Message" on page 3-37 provides information on encrypting and decrypting the username token. "Assembling a Secure Web Service" on page 4-1 provides bottom up and top down examples which use username token.
Sender Authentication Username with clear text password or digest password (SA2)	T-05: Principal Spoofing	SA2	Inbound configuration is supported via the `verify-username-token` element.	Both inbound and outbound configuration are supported.	"Using a Username Token" on page 3-7.
Message Integrity, Sender Authentication XML Digital Signature (SI1) with: ▪ Username with clear text password or digest password (SA2), or ▪ X509 Certificate (SA3), or ▪ SAML Token (SA5), or ▪ REL Token (SA6)	T-01: Message Alteration T-05: Principal Spoofing	SI1, SA2, SA3 and SA5 are supported SA6 is not supported.	Inbound policy for SI1 (verify signature) is supported through Application Server Control. You must configure an instance/port level keystore with a signature key. Inbound policies for SA2, SA3 and SA5 are supported through Application Server Control. Outbound policies for SA2, SA3 and SA5 are not supported through Application Server Control.	Both Inbound and Outbound policies for SI1, SA2, SA3, SA5 are supported through JDeveloper. You must configure a key store with a signature key.	Chapter 2, "Configuring Web Service Security" "Assembling Security into a Web Service Bottom Up" on page 4-5 describes the bottom up XML Signature and Username token cases.

Table B–1 (Cont.) Message Layer Security Solutions

Solution	Threat Number and Name	Supported Solutions	Application Server Control Support	JDeveloper Support	Where Documented
Message Confidentiality, Sender Authentication XML Encryption (SC1) with: ▪ Username token with password or digest with encrypted password (SA1), or ▪ Username token with password or digest (SA2), or ▪ X509 Token (SA3), or ▪ SAML Token (SA5), or ▪ REL Token (SA6)	T-02: Message Confidentiality T-05: Principal Spoofing	SC1, SA1, SA2, SA3, and SA5 are supported SA6 is not supported.	Inbound policy for SC1 is supported through Application Server Control. You must configure an instance/port-level keystore with an encryption key. Inbound policies for SA2, SA3, and SA5 are supported through Application Server Control. Outbound policies for SA2, SA3, SA5, and SC1 are not supported through Application Server Control.	Both Inbound and Outbound policies for SC1, SA2, SA3, and SA5 are supported through JDeveloper. You must configure an instance/port-level keystore with an encryption key.	Configuring security tokens and encryption are covered in Chapter 3, "Administering Web Services Security" "Assembling Security into a Web Service Bottom Up" on page 4-5 describes the bottom up XML Encryption case.
One-Way AnyNode – AnyNode Message Confidentiality, Integrity, Sender Authentication XML Digital Signature (SI1) with: ▪ XML Encryption (SC1), or ▪ Username token with password or digest with encrypted password (SA1), or ▪ Username token with password or digest (SA2), or ▪ X509 Token (SA3), or ▪ SAML Token (SA5), or ▪ REL Token (SA6)	T-01: Message Alteration T-02: Confidentiality T-05: Principal Spoofing T-06: Forged claims	SI1, SC1, SA1, SA2, SA3, and SA5 are supported SA6 is not supported	Inbound policies for SI1 and SC1 are supported through Application Server Control. Inbound policies for SA2, SA3, and SA5 are supported through Application Server Control. Outbound policies for SC1, SC2, SA2, SA3, and SA5 are not supported through Application Server Control	Both Inbound and Outbound policies for SI1, SC1, SA2, SA3, and SA5 are supported through JDeveloper. You must configure a keystore with signature and encryption keys.	Configuring security tokens and XML signature are covered in Chapter 3, "Administering Web Services Security"

Table B–1 (Cont.) Message Layer Security Solutions

Solution	Threat Number and Name	Supported Solutions	Application Server Control Support	JDeveloper Support	Where Documented
Two-Way AnyNode – AnyNode Message Confidentiality, Integrity, Sender Authentication XML Digital Signature (SI1) with: ▪ XML Encryption (SC1), or ▪ Username token with password/digest with encrypted password (SA1), or ▪ Username token with password/digest (SA2), or ▪ X509 Token (SA3), or ▪ SAML Token (SA5), or ▪ REL Token (SA6)	T-01: Message Alteration T-02: Message Confidentiality T-05: Principal Spoofing T-06: Forged claims	SI1, SC1, SA1, SA2, SA3, and SA5 are supported SA6 is not supported	Inbound policies for SI1 and SC1 are supported through Application Server Control. Inbound policies for SA2, SA3, and SA5 are supported through Application Server Control. Outbound policies for SI1, SC1, SA2, SA3, and SA5 are not supported through Application Server Control	Both Inbound and Outbound policies for SI1, SC1, SA2, SA3, SA5 are supported through JDeveloper.	Configuring security tokens and XML signature are covered in Chapter 3, "Administering Web Services Security"
Hybrid: Transport Integrity and Confidentiality, AnyNode-AnyNode Message Confidentiality, Integrity, Mutual Authentication SSL/TLS (BISP1) with XML Signature (SI1) with: ▪ XML Encryption (SC1), or ▪ Username token with password/digest with encrypted password (SA1), or ▪ Username token with password/digest (SA2), or ▪ X509 Token (SA3), or ▪ SAML Token (SA5), or ▪ REL Token (SA6)	T-01: Message Alteration T-02: Message Confidentiality T-03: Falsified Messages T-04: Man in the Middle T-05: Principal Spoofing T-06: Forged claims T-07: Replay of Message Parts T-08: Replay	BISP, BC1, SI1, SC1, SA1, SA2, SA3, and SA5 are supported SA6 is not supported	Inbound policies for SI1 and SC1 are supported through Application Server Control. Inbound policies for SA2, SA3, and SA5 are supported through Application Server Control. Outbound policies for SC1, SI1, SA2, SA3, SA5, BISP, and BC1 are not supported through Application Server Control	Both Inbound and Outbound policies for SI1, SC1, SA2, SA3, and SA5 are supported through JDeveloper. BISP and BC1 are not supported through JDeveloper.	Configuring security tokens and XML signature are covered in Chapter 3, "Administering Web Services Security" For the manual steps to configure SSL, see the *Oracle Containers for J2EE Security Guide*

Table B–1 (Cont.) Message Layer Security Solutions

Solution	Threat Number and Name	Supported Solutions	Application Server Control Support	JDeveloper Support	Where Documented
Hybrid: Transport Integrity and Confidentiality, Mutual Authentication AnyNode-AnyNode Message Confidentiality, Integrity, Mutual Authentication SSL/TLS (BISP) with SSL/TLS and client authentication (BC1) with: ▪ XML Signature (SI1), or ▪ XML Encryption (SC1), or ▪ Username token with password/digest with encrypted password (SA1), or ▪ Username token with password/digest (SA2), or ▪ X509 Token (SA3), or ▪ SAML Token (SA5), or ▪ REL Token (SA6)	T-01: Message Alteration T-02: Message Confidentiality T-03: Falsified Messages T-04: Man in the Middle T-05: Principal Spoofing T-06: Forged claims T-07: Replay of Message Parts T-08: Replay	BISP, SI1, SC1, SA1, SA2, SA3, and SA5 are supported SA6 is not supported	Inbound policies for SI1 and SC1 are supported through Application Server Control. Inbound policies for SA2, SA3, and SA5 are supported through Application Server Control. Outbound policies for SI1, SC1, SA2, SA3, SA5, and BISP are not supported through Application Server Control	Both Inbound and Outbound policy for SI1, SC1, SA2, SA3, and SA5 are supported through JDeveloper. BISP is not supported through JDeveloper	Manual steps for configuring SSL are described in the *Oracle Containers for J2EE Security Guide*.

Table B–2 describes the security threats that can impact the transport layer and the possible solutions that can be implemented under OracleAS Web Services Security.

Table B–2 Transport Layer Security Solutions

Solution	Threat Number and Name	Solutions Supported	Application Server Control Support	JDeveloper Support	Where Documented
Consumer Authentication ▪ HTTP Basic Authentication (BC2), or ▪ HTTP Digest Authentication (BC3), or ▪ HTTP Attributes (BC4)	T-05: Principal Spoofing	Yes	No	No	"Adding Transport-Level Security to a Web Service" on page 4-15. See also, the *Oracle Containers for J2EE Security Guide*.
Transport Integrity, Confidentiality, Provider Authentication SSL/TLS (BISP1)	T-01: Message Alteration T-02: Message Confidentiality	Yes	No	No	"Adding Transport-Level Security to a Web Service" on page 4-15. See also the *Oracle Containers for J2EE Security Guide*.
Transport Integrity, Confidentiality, Mutual Authentication SSL/TLS (BISP1) with SSL/TLS with client authentication (BC1)	T-01: Message Alteration T-02: Message Confidentiality T-03: Falsified Messages T-04: Man in the Middle T-05: Principal Spoofing T-06: Forged claims T-07: Replay of Message Parts T-08: Replay	Yes	No	No	"Adding Transport-Level Security to a Web Service" on page 4-15. See also the *Oracle Containers for J2EE Security Guide*.
Transport Integrity, Confidentiality, Mutual Authentication with Enhanced Consumer Authentication SSL/TLS (BISP1) with HTTP Basic/ HTTP Digest Authentication (BC5)	T-01: Message Alteration T-02: Message Confidentiality T-03: Falsified Messages T-05: Principal Spoofing T-06: Forged claims T-07: Replay of Message Parts T-08: Replay	Yes	No	No	"Adding Transport-Level Security to a Web Service" on page 4-15. See also, the *Oracle Containers for J2EE Security Guide*.

Table B–3 provides a brief description of the tags that represent message- and transport-layer security mechanisms described in Table B–1 and Table B–2.

Table B–3 Unique IDs for Message and Transport Layer Security Mechanisms

Tag	Description
BC1	SSL/TLS with client authentication
BC2	HTTP basic
BC3	HTTP digest
BC4	HTTP attributes
BC5	HTTP basic or HTTP digest
BISP1	SSL/TSL

Table B–3 (Cont.) Unique IDs for Message and Transport Layer Security Mechanisms

Tag	Description
SA1	XML encryption, username token with either password or digest
SA2	username and either password or digest
SA3	X.509 certificate
SA5	SAML token
SA6	REL token
SC1	XML encryption
SI1	XML digital signature

Table B–4 provides a brief description of the security threat IDs and names used in Table B–1 and Table B–2.

Table B–4 Security Threats Addressed by OracleAS Web Services Security

Threat ID	Threat Name	Description
T-01	Message Alteration	The message information is altered by inserting, removing or otherwise modifying information created by the originator of the information and mistaken by the receiver as being the originators intention.
T-02	Confidentiality	Information within the message is viewable by unintended and unauthorized participants.
T-03	Falsified Messages	Fake messages are constructed and sent to a receiver who believes them to have come from a party other than the sender.
T-04	Man in the Middle	A party poses as the other participant to the real sender and receiver in order to fool both participants (for example, the attacker is able to downgrade the level of cryptography used to secure the message).
T-05	Principal Spoofing	A message is sent which appears to be from another principal.
T-06	Forged claims	A message is sent in which the security claims are forged in an effort to gain access to otherwise unauthorized information (for example, a security token is used which wasn't really issued by the specified authority).
T-07	Replay of Message Parts	A message is sent which includes portions of another message in an effort to gain access to otherwise unauthorized information or to cause the receiver to take some action.
T-08	Replay	A whole message is resent by an attacker.
T-09	Denial of Service	Amplifier Attack: attacker does a small amount of work and forces system under attack to do a large amount of work.

C

Third Party Licenses

This appendix includes the Third Party License for all the third party products included with Oracle Application Server Web Services Security.

Apache

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights.

The Apache license agreements apply to the following included Apache components:

- Apache HTTP Server
- Apache JServ
- mod_jserv
- Regular Expression package version 1.3
- Apache Expression Language packaged in commons-el.jar
- mod_mm 1.1.3
- Apache XML Signature and Apache XML Encryption v. 1.4 for Java and 1.0 for C++
- log4j 1.1.1
- BCEL v. 5
- XML-RPC v. 1.1
- Batik v. 1.5.1
- ANT 1.6.2 and 1.6.5
- Crimson v. 1.1.3
- ant.jar
- wsif.jar
- bcel.jar
- soap.jar
- Jakarta CLI 1.0
- jakarta-regexp-1.3.jar

- JSP Standard Tag Library 1.0.6 and 1.1
- Struts 1.1
- Velocity 1.3
- svnClientAdapter
- commons-logging.jar
- wsif.jar
- commons-el.jar
- standard.jar
- jstl.jar

The Apache Software License

License for Apache Web Server 1.3.29

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000-2002 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
 *    permission of the Apache Software Foundation.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
```

```
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 * ====================================================================
 *
 * This software consists of voluntary contributions made by many
 * individuals on behalf of the Apache Software Foundation.  For more
 * information on the Apache Software Foundation, please see
 * <http://www.apache.org/>.
 *
 * Portions of this software are based upon public domain software
 * originally written at the National Center for Supercomputing
Applications,
 * University of Illinois, Urbana-Champaign.
```

License for Apache Web Server 2.0

```
Copyright (c) 1999-2004, The Apache Software Foundation
Licensed under the Apache License, Version 2.0 (the "License"); you may not use
this file except in compliance with the License.  You may obtain a copy of the
License at http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed
under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied.  See the License for the
specific language governing permissions and limitations under the License.
Copyright (c) 1999-2004, The Apache Software Foundation
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
```

and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or
Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices
stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

Apache SOAP

This program contains third-party code from the Apache Software Foundation ("Apache"). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

Apache SOAP License

Apache SOAP license 2.3.1

Copyright (c) 1999 The Apache Software Foundation. All rights reserved.
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses

granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

JSR 110

This program contains third-party code from IBM Corporation ("IBM"). Under the terms of the IBM license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the IBM software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the IBM software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or IBM.

Copyright IBM Corporation 2003 - All rights reserved

Java APIs for the WSDL specification are available at:
http://www-124.ibm.com/developerworks/projects/wsdl4j/

Jaxen

This program contains third-party code from the Apache Software Foundation ("Apache") and from the Jaxen Project ("Jaxen"). Under the terms of the Apache and Jaxen licenses, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache and Jaxen software, and the terms contained in the following notices do not change those rights.

The Jaxen License

```
Copyright (C) 2000-2002 bob mcwhirter & James Strachan. All rights reserved.
Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list
of conditions, and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this
list of conditions, and the disclaimer that follows these conditions in the
documentation and/or other materials provided with the distribution.

The name "Jaxen" must not be used to endorse or promote products derived from this
software without prior written permission. For written permission, please contact
license@jaxen.org.

Products derived from this software may not be called "Jaxen", nor may "Jaxen"
appear in their name, without prior written permission from the Jaxen Project
Management (pm@jaxen.org).

In addition, we request (but do not require) that you include in the end-user
documentation provided with the redistribution and/or in the software itself an
acknowledgment equivalent to the following: "This product includes software
developed by the Jaxen Project (http://www.jaxen.org/)." Alternatively, the
acknowledgment may be graphical using the logos available at
http://www.jaxen.org/.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE Jaxen
AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on
behalf of the Jaxen Project and was originally created by bob mcwhirter and James
Strachan . For more information on the Jaxen Project, please see
http://www.jaxen.org/.
```

SAXPath

This program contains third-party code from SAXPath. Under the terms of the SAXPath license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the SAXPath software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the SAXPath software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or SAXPath.

The SAXPath License

```
Copyright (C) 2000-2002 werken digital. All rights reserved. Redistribution and
use in source and binary forms, with or without modification, are permitted
provided that the following conditions are met:
```

Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

The name "SAXPath" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@saxpath.org.

Products derived from this software may not be called "SAXPath", nor may "SAXPath" appear in their name, without prior written permission from the SAXPath Project Management (pm@saxpath.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgment equivalent to the following: "This product includes software developed by the SAXPath Project (http://www.saxpath.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.saxpath.org/.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE SAXPath AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the SAXPath Project and was originally created by bob mcwhirter and James Strachan . For more information on the SAXPath Project, please see http://www.saxpath.org/.

W3C DOM

This program contains third-party code from the World Wide Web Consortium ("W3C"). Under the terms of the W3C license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the W3C software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the W3C software is provided by Oracle AS IS and without warranty or support of any kind from Oracle or W3C.

The W3C License

W3C® SOFTWARE NOTICE AND LICENSE
http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231
This work (and included software, documentation such as READMEs, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X