# Oracle® Fusion Middleware

Securing WebLogic Web Services for Oracle WebLogic Server

11*g* Release 1 (10.3.3)

**E13713-03**

April 2010

This document explains how to secure WebLogic Web services for Oracle WebLogic Server, including configuring transport- and message-level security.

ORACLE

Oracle Fusion Middleware Securing WebLogic Web Services for Oracle WebLogic Server, 11*g* Release 1 (10.3.3)

E13713-03

# Contents

**Preface** ..... vii

Documentation Accessibility ..... vii
Conventions ..... vii

## 1 Overview of Web Services Security

1.1 Overview of Web Services Security ..... 1-1
1.2 What Type of Security Should You Configure? ..... 1-1

## 2 Configuring Message-Level Security

2.1 Overview of Message-Level Security ..... 2-2
2.1.1 Web Services Security Supported Standards ..... 2-2
2.1.1.1 Web Services Trust and Secure Conversation ..... 2-2
2.1.1.2 Web Services SecurityPolicy 1.2 ..... 2-3
2.2 Main Use Cases of Message-Level Security ..... 2-3
2.3 Using Policy Files for Message-Level Security Configuration ..... 2-4
2.3.1 Using Policy Files With JAX-WS ..... 2-4
2.3.2 WS-Policy Namespace ..... 2-4
2.3.3 WS-SecurityPolicy Namespace ..... 2-5
2.3.4 Version-Independent Policy Supported ..... 2-5
2.4 Configuring Simple Message-Level Security: Main Steps ..... 2-6
2.4.1 Ensuring That WebLogic Server Can Validate the Client's Certificate ..... 2-8
2.4.2 Updating the JWS File with @Policy and @Policies Annotations ..... 2-8
2.4.2.1 Loading a Policy From the CLASSPATH ..... 2-11
2.4.3 Using Key Pairs Other Than the Out-Of-The-Box SSL Pair ..... 2-12
2.5 Updating a Client Application to Invoke a Message-Secured Web Service ..... 2-13
2.5.1 Invoking a Web Service From a Client Running in a WebLogic Server Instance ..... 2-17
2.6 Example of Adding Security to a JAX-WS Web Service ..... 2-18
2.7 Creating and Using a Custom Policy File ..... 2-25
2.8 Configuring the WS-Trust Client ..... 2-26
2.8.1 Supported Token Types ..... 2-27
2.8.2 Configuring WS-Trust Client Properties ..... 2-27
2.8.2.1 Obtaining the URI of the Secure Token Service ..... 2-28
2.8.2.2 Configuring STS URI for WS-SecureConversation: Standalone Client ..... 2-28
2.8.2.3 Configuring STS URI for SAML: Standalone Client ..... 2-29
2.8.2.4 Configuring STS URI Using WLST: Client On Server Side ..... 2-29

2.8.2.5 Configuring STS URI Using Console: Client On Server Side ... 2-30
2.8.2.6 Configuring STS Security Policy: Standalone Client ... 2-31
2.8.2.7 Configuring STS Security Policy Using WLST: Client On Server Side ... 2-31
2.8.2.8 Configuring STS Security Policy: Using the Console ... 2-32
2.8.2.9 Configuring the STS SOAP and WS-Trust Version: Standalone Client ... 2-32
2.8.2.10 Configuring the SAML STS Server Certificate: Standalone Client ... 2-33
2.8.3 Sample WS-Trust Client for SAML 2.0 Bearer Token over HTTPS ... 2-33
2.8.4 Sample WS-Trust Client for SAML 2.0 Bearer Token with WSS 1.1 Message Protections ... 2-37
2.9 Configuring and Using Security Contexts and Derived Keys ... 2-43
2.9.1 Specification Backward Compatibility ... 2-44
2.9.2 WS-SecureConversation and Clusters ... 2-44
2.9.3 Updating a Client Application to Negotiate Security Contexts ... 2-44
2.10 Associating Policy Files at Runtime Using the Administration Console ... 2-46
2.11 Using Security Assertion Markup Language (SAML) Tokens For Identity ... 2-47
2.11.1 Using SAML Tokens for Identity: Main Steps ... 2-48
2.11.2 Specifying the SAML Confirmation Method ... 2-49
2.11.2.1 Specifying the SAML Confirmation Method (Proprietary Policy Only) ... 2-50
2.12 Associating a Web Service with a Security Configuration Other Than the Default ... 2-52
2.13 Valid Class Names and Token Types for Credential Provider ... 2-53
2.14 Using System Properties to Debug Message-Level Security ... 2-53
2.15 Using a Client-Side Security Policy File ... 2-54
2.15.1 Associating a Policy File with a Client Application: Main Steps ... 2-54
2.15.2 Updating clientgen to Generate Methods That Load Policy Files ... 2-55
2.15.3 Updating a Client Application To Load Policy Files (JAX-RPC Only) ... 2-55
2.16 Using WS-SecurityPolicy 1.2 Policy Files ... 2-57
2.16.1 Transport Level Policies ... 2-58
2.16.2 Protection Assertion Policies ... 2-59
2.16.3 WS-Security 1.0 Username and X509 Token Policies ... 2-60
2.16.4 WS-Security 1.1 Username and X509 Token Policies ... 2-61
2.16.5 WS-SecureConversation Policies ... 2-62
2.16.6 SAML Token Profile Policies ... 2-64
2.17 Choosing a Policy ... 2-65
2.18 Unsupported WS-SecurityPolicy 1.2 Assertions ... 2-66
2.19 Using the Optional Policy Assertion ... 2-67
2.20 Configuring Element-Level Security ... 2-68
2.20.1 Define and Use a Custom Element-Level Policy File ... 2-69
2.20.1.1 Adding the Policy Annotation to JWS File ... 2-70
2.20.2 Implementation Notes ... 2-71
2.21 Smart Policy Selection ... 2-71
2.21.1 Example of Security Policy With Policy Alternatives ... 2-71
2.21.2 Configuring Smart Policy Selection ... 2-74
2.21.2.1 How the Policy Preference is Determined ... 2-74
2.21.2.2 Configuring Smart Policy Selection in the Console ... 2-74
2.21.2.3 Understanding Body Encryption in Smart Policy ... 2-75
2.21.2.4 Smart Policy Selection for a Standalone Client ... 2-76
2.21.3 Multiple Transport Assertions ... 2-76
2.22 Example of Adding Security to MTOM Web Service ... 2-76

2.22.1 Files Used by This Example ... 2-77
2.22.2 SecurityMtomService.java ... 2-77
2.22.3 MtomClient.java ... 2-79
2.22.4 configWss.py Script File ... 2-82
2.22.5 Build.xml File ... 2-84
2.22.6 Building and Running the Example ... 2-87
2.22.7 Deployed WSDL for SecurityMtomService ... 2-88
2.23 Example of Adding Security to Reliable Messaging Web Service ... 2-92
2.23.1 Overview of Secure and Reliable SOAP Messaging ... 2-92
2.23.2 Overview of the Example ... 2-92
2.23.2.1 How the Example Sets Up WebLogic Security ... 2-93
2.23.3 Files Used by This Example ... 2-94
2.23.4 Revised ReliableEchoServiceImpl.java ... 2-95
2.23.5 Revised configWss.py ... 2-95
2.23.6 Revised configWss_Service.py ... 2-96
2.23.7 Building and Running the Example ... 2-97
2.24 Securing Web Services Atomic Transactions ... 2-97
2.25 Proprietary Web Services Security Policy Files (JAX-RPC Only) ... 2-98
2.25.1 Abstract and Concrete Policy Files ... 2-100
2.25.2 Auth.xml ... 2-100
2.25.3 Sign.xml ... 2-101
2.25.4 Encrypt.xml ... 2-102
2.25.5 Wssc-dk.xml ... 2-103
2.25.6 Wssc-sct.xml ... 2-104

## 3 Configuring Transport-Level Security

3.1 Configuring Transport-Level Security Through Policy ... 3-1
3.1.1 Configuring Transport-Level Security Through Policy: Main Steps ... 3-2
3.2 Example of Using JWS Annotations in Your JWS File ... 3-3
3.3 Example of Configuring Transport Security for JAX-WS ... 3-3
3.3.1 One-Way SSL (HTTPS and HTTP Basic Authentication Example) ... 3-4
3.4 New Two-Way Persistent SSL Client API for JAX-WS ... 3-8
3.4.1 Example of Getting SSLSocketFactory From System Properties ... 3-9
3.5 Configuring Transport-Level Security Via UserDataConstraint: Main Steps (JAX-RPC Only) ... 3-10
3.6 Configuring Two-Way SSL for a Client Application ... 3-11
3.7 Using a Custom SSL Adapter with Reliable Messaging ... 3-12

## 4 Configuring Access Control Security (JAX-RPC Only)

4.1 Configuring Access Control Security: Main Steps ... 4-1
4.2 Updating the JWS File With the Security-Related Annotations ... 4-3
4.3 Updating the JWS File With the @RunAs Annotation ... 4-5
4.4 Setting the Username and Password When Creating the Service Object ... 4-5

## A Using Oracle Web Services Manager Security Policies

A.1 Overview ... A-1

A.1.1 When Should You Use Oracle WS-Security Policies? ..... A-1
A.2 What Oracle WSM Security Policies Are Available? ..... A-4
A.2.1 Is There Compatibility Between WebLogic Policies and Oracle WSM Policies? ..... A-9
A.3 What Oracle WSM WS-Security Policies Are Not Available? ..... A-10
A.4 Where are the Oracle WSM Policies Documented? ..... A-10
A.5 Adding Oracle WSM WS-Security Policies to a Web Service ..... A-11
A.5.1 SecurityPolicy and SecurityPolicies Annotations ..... A-11
A.5.2 Configuring Oracle WSM Security Policies in Administration Console ..... A-12
A.6 Adding Oracle WSM WS-Security Policies to Clients ..... A-14
A.6.1 Associating a Policy File with a Client Application: Main Steps ..... A-15
A.7 Configuring Permission-Based Authorization Policies ..... A-15
A.8 Configuring the Credential Store Using WLST ..... A-16
A.8.1 How to Create and Use a Java Keystore ..... A-17
A.8.1.1 How to Create Private Keys and Load Trusted Certificates ..... A-17
A.8.2 Manage the Credential Store Framework ..... A-18
A.8.2.1 How to Update Your Credential Store Using WLST ..... A-19
A.9 Policy Configuration Overrides for the Web Service Client ..... A-19
A.10 Creating Custom Assertions ..... A-20
A.10.1 Overview of Custom Assertion Creation ..... A-20
A.10.2 Step 1: Create the Custom Policy File ..... A-21
A.10.3 Step 2: Add the Custom Policy to the Policy Store ..... A-21
A.10.4 Step 3: Create the Custom Assertion Class ..... A-22
A.10.5 Step 4: Create the Custom Assertion Class JAR File ..... A-23
A.10.6 Step 5: Update Your CLASSPATH ..... A-23
A.10.7 Step 6: Develop and Deploy a JAX-WS Web Service ..... A-24
A.10.8 Step 7: Attach the Custom Policy to the JAX-WS Web Service ..... A-24
A.11 Monitoring and Testing the Web Service ..... A-24

# Preface

This preface describes the document accessibility features and conventions used in this guide—Securing WebLogic Web Services for Oracle WebLogic Server.

## Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. 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.

### Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

## 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. |

| Convention | Meaning |
|---|---|
| *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 Overview of Web Services Security

The following sections describe how to configure security for your Web service:

- Section 1.1, "Overview of Web Services Security"
- Section 1.2, "What Type of Security Should You Configure?"

## 1.1 Overview of Web Services Security

To secure your WebLogic Web service, you configure one or more of three different types of security.

***Table 1–1 Web Services Security***

| Security Type | Description |
|---|---|
| Message-level security | Data in a SOAP message is digitally signed or encrypted. May also include identity tokens for authentication. See Chapter 2, "Configuring Message-Level Security". |
| Transport-level security | SSL is used to secure the connection between a client application and the Web service. See Chapter 3, "Configuring Transport-Level Security". |
| Access control security | Specifies which roles are allowed to access Web services. See Chapter 4, "Configuring Access Control Security (JAX-RPC Only)". |

## 1.2 What Type of Security Should You Configure?

**Message-level security** includes all the security benefits of SSL, but with additional flexibility and features. Message-level security is end-to-end, which means that a SOAP message is secure even when the transmission involves one or more intermediaries. The SOAP message itself is digitally signed and encrypted, rather than just the connection. And finally, you can specify that only individual parts or elements of the message be signed, encrypted, or required.Transport-level security, however, secures only the connection itself. This means that if there is an intermediary between the client and WebLogic Server, such as a router or message queue, the intermediary gets the SOAP message in plain text. When the intermediary sends the message to a second receiver, the second receiver does not know who the original sender was. Additionally, the encryption used by SSL is "all or nothing": either the entire SOAP message is encrypted or it is not encrypted at all. There is no way to specify that only selected parts of the SOAP message be encrypted. Message-level security can also include identity tokens for authentication.

**Transport-level security** secures the connection between the client application and WebLogic Server with Secure Sockets Layer (SSL). SSL provides secure connections by allowing two applications connecting over a network to authenticate the other's

identity and by encrypting the data exchanged between the applications. Authentication allows a server, and optionally a client, to verify the identity of the application on the other end of a network connection. A client certificate (two-way SSL) can be used to authenticate the user.

Encryption makes data transmitted over the network intelligible only to the intended recipient.

Transport-level security includes HTTP BASIC authentication as well as SSL.

**Access control security** answers the question "who can do what?" First you specify the security roles that are allowed to access a Web service; a *security role* is a privilege granted to users or groups based on specific conditions. Then, when a client application attempts to invoke a Web service operation, the client authenticates itself to WebLogic Server, and if the client has the authorization, it is allowed to continue with the invocation. Access control security secures only WebLogic Server resources. That is, if you configure *only* access control security, the connection between the client application and WebLogic Server is not secure and the SOAP message is in plain text.

# 2

# Configuring Message-Level Security

In this release of WebLogic Server, message-level security features are supported in both the JAX-RPC and JAX-WS stacks.

The following sections describe how to configure security for your Web service:


- Section 2.1, "Overview of Message-Level Security"
- Section 2.2, "Main Use Cases of Message-Level Security"
- Section 2.3, "Using Policy Files for Message-Level Security Configuration"
- Section 2.4, "Configuring Simple Message-Level Security: Main Steps"
- Section 2.5, "Updating a Client Application to Invoke a Message-Secured Web Service"
- Section 2.6, "Example of Adding Security to a JAX-WS Web Service"
- Section 2.7, "Creating and Using a Custom Policy File"
- Section 2.8, "Configuring the WS-Trust Client"
- Section 2.9, "Configuring and Using Security Contexts and Derived Keys"
- Section 2.10, "Associating Policy Files at Runtime Using the Administration Console"
- Section 2.11, "Using Security Assertion Markup Language (SAML) Tokens For Identity"
- Section 2.12, "Associating a Web Service with a Security Configuration Other Than the Default"
- Section 2.13, "Valid Class Names and Token Types for Credential Provider"
- Section 2.14, "Using System Properties to Debug Message-Level Security"
- Section 2.15, "Using a Client-Side Security Policy File"
- Section 2.16, "Using WS-SecurityPolicy 1.2 Policy Files"
- Section 2.17, "Choosing a Policy"
- Section 2.18, "Unsupported WS-SecurityPolicy 1.2 Assertions"
- Section 2.19, "Using the Optional Policy Assertion"
- Section 2.20, "Configuring Element-Level Security"
- Section 2.21, "Smart Policy Selection"
- Section 2.21.3, "Multiple Transport Assertions"
- Section 2.23, "Example of Adding Security to Reliable Messaging Web Service"

- Section 2.24, "Securing Web Services Atomic Transactions"
- Section 2.25, "Proprietary Web Services Security Policy Files (JAX-RPC Only)"

## 2.1 Overview of Message-Level Security

Message-level security specifies whether the SOAP messages between a client application and the Web service invoked by the client should be digitally signed or encrypted, or both. It also can specify a shared security context between the Web service and client in the event that they exchange multiple SOAP messages. You can use message-level security to assure:

- Confidentiality, by encrypting message parts
- Integrity, by digital signatures
- Authentication, by requiring username, X.509, or SAML tokens

See Section 2.4, "Configuring Simple Message-Level Security: Main Steps" for the basic steps you must perform to configure simple message-level security. This section discusses configuration of the Web services runtime environment, as well as configuration of message-level security for a particular Web service and how to code a client application to invoke the service.

You can also configure message-level security for a Web service at runtime, after a Web service has been deployed. See Section 2.10, "Associating Policy Files at Runtime Using the Administration Console" for details.

---

**Note:** You cannot digitally sign or encrypt a SOAP attachment.

---

### 2.1.1 Web Services Security Supported Standards

---

**Note:** *Standards Supported by WebLogic Web Services* is the definitive source of Web service standards supported in this release.

---

WebLogic Web services implement the following OASIS Standard 1.1 Web Services Security (WS-Security 1.1 (`http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`) specifications, dated February 1, 2006:

- WS-Security 1.0 and 1.1
- Username Token Profile 1.0 and 1.1
- X.509 Token Profile 1.0 and 1.1
- SAML Token Profile 1.0 and 1.1

These specifications provide security token propagation, message integrity, and message confidentiality. These mechanisms can be used independently (such as passing a username token for user authentication) or together (such as digitally signing and encrypting a SOAP message and specifying that a user must use X.509 certificates for authentication).

#### 2.1.1.1 Web Services Trust and Secure Conversation

WebLogic Web services implement the Web Services Trust (WS-Trust 1.3) and Web Services Secure Conversation (WS-SecureConversation 1.3) specifications, which

together provide secure communication between Web services and their clients (either other Web services or standalone Java client applications).

The WS-Trust specification defines extensions that provide a framework for requesting and issuing security tokens, and to broker trust relationships.

The WS-SecureConversation specification defines mechanisms for establishing and sharing security contexts, and deriving keys from security contexts, to enable the exchange of multiple messages. Together, the security context and derived keys potentially increase the overall performance and security of the subsequent exchanges.

#### 2.1.1.2 Web Services SecurityPolicy 1.2

The WS-Policy specification defines a framework for allowing Web services to express their constraints and requirements. Such constraints and requirements are expressed as policy assertions.

WS-SecurityPolicy defines a set of security policy assertions for use with the WS-Policy framework to describe how messages are to be secured in the context of WSS: SOAP Message Security, WS-Trust and WS-SecureConversation.

You configure message-level security for a Web service by attaching one or more policy files that contain security policy statements, as specified by the WS-SecurityPolicy specification. See Section 2.3, "Using Policy Files for Message-Level Security Configuration" for detailed information about how the Web services runtime environment uses security policy files.

For information about the elements of the Web Services SecurityPolicy 1.2 that are not supported in this release of WebLogic Server, see Section 2.18, "Unsupported WS-SecurityPolicy 1.2 Assertions".

## 2.2 Main Use Cases of Message-Level Security

The implementation of the *Web Services Security: SOAP Message Security* specification supports the following use cases:

- Use X.509 certificates to sign and encrypt a SOAP message, starting from the client application that invokes the message-secured Web service, to the WebLogic Server instance that is hosting the Web service and back to the client application.
- Specify the SOAP message targets that are signed, encrypted, or required: the body, specific SOAP headers, or specific elements.
- Include a token (username, SAML, or X.509) in the SOAP message for authentication.
- Specify that a Web service and its client (either another Web service or a standalone application) establish and share a security context when exchanging multiple messages using WS-SecureConversation (WSSC).
- Derive keys for *each* key usage in a secure context, once the context has been established and is being shared between a Web service and its client. This means that a particular SOAP message uses two derived keys, one for signing and another for encrypting, and each SOAP message uses a different pair of derived keys from other SOAP messages. Because each SOAP message uses its own pair of derived keys, the message exchange between the client and Web service is extremely secure.

## 2.3 Using Policy Files for Message-Level Security Configuration

You specify the details of message-level security for a WebLogic Web service with one or more security policy files. The WS-SecurityPolicy specification provides a general purpose model and XML syntax to describe and communicate the security policies of a Web service.

---

**Note:** Previous releases of WebLogic Server, released before the formulation of the WS-SecurityPolicy specification, used security policy files written under the WS-Policy specification, using a proprietary schema for security policy. This proprietary schema for security policy is deprecated, and it is recommended that you use the WS-SecurityPolicy 1.2 format.

This release of WebLogic Server supports either security policy files that conform to the WS-SecurityPolicy 1.2 specification or the Web services security policy schema first included in WebLogic Server 9, but not both in the same Web service. The formats are mutually incompatible.

For information about the predefined WS-SecurityPolicy 1.2 security policy files, see Section 2.16, "Using WS-SecurityPolicy 1.2 Policy Files".

---

The security policy files used for message-level security are XML files that describe whether and how the SOAP messages resulting from an invoke of an operation should be digitally signed or encrypted. They can also specify that a client application authenticate itself using a username, SAML, or X.509 token.

You use the `@Policy` and `@Policies` JWS annotations in your JWS file to associate policy files with your Web service. You can associate any number of policy files with a Web service, although it is up to you to ensure that the assertions do not contradict each other. You can specify a policy file at both the class- and method level of your JWS file.

---

**Note:** If you specify a transport-level security policy for your Web service, it must be at the class level.

In addition, the transport-level security policy must apply to both the inbound and outbound directions. That is, you cannot have HTTPS for inbound and HTTP for outbound.

---

### 2.3.1 Using Policy Files With JAX-WS

For maximum portability, Oracle recommends that you use WS-Policy 1.2 and OASIS WS-SecurityPolicy 1.2 with JAX-WS.

### 2.3.2 WS-Policy Namespace

WebLogic Server supports WS-Policy 1.2 with the following namespace:

`http://schemas.xmlsoap.org/ws/2004/09/policy`

**Note:** WebLogic Server also supports WS-Policy 1.5 (now a W3C standard) with the following namespace:
`http://www.w3.org/ns/ws-policy`

### 2.3.3 WS-SecurityPolicy Namespace

The following OASIS WS-SX TC Web Services SecurityPolicy namespace is supported:

`http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702`

In addition to this new version of the namespace, WebLogic Server continues to support the following Web Services SecurityPolicy namespace:

`http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512`

In most of the cases, the policy assertions are identical for either namespaces, with the following exceptions.

- Trust10 and Trust13 assertion. Both Trust10 and Trust13 assertions are supported.
- SC10SecurityContextToken and SC13SecurityContextToken, as described in Section 2.9.1, "Specification Backward Compatibility".
- Derived Key using different WSSC versions (200502, 1.3).

### 2.3.4 Version-Independent Policy Supported

This version of WebLogic Server supports version-independent policy. You can combine protocol-specific policies such as WS-SecurityPolicy and WS-ReliableMessaging policy that are based on different versions of the WS-Policy specification. At runtime, the merged policy file then contains two or more different namespaces.

There are three versions of WS-SecurityPolicy in this release of WebLogic Server:

- (1) WS-SecurityPolicy 1.2 OASIS standard.
- (2) WS-SecurityPolicy 1.2, as included in WebLogic Server 10.0.
- (3) Proprietary format WebLogic Server 9.x-style policies (deprecated).

You can mix and match any version of WS-Policy with (1), (2), or a combination of (1) and (2). However, you cannot mix and match (3) with (1) or (2) and with different versions of WS-Policy.

The version match possibilities are shown in Table 2–1.

***Table 2–1 Version-Independent Matrix***

| Security Policy Versions | WS-Policy 1.5 | WS-Policy 1.2 | WS-Policy 1.5 AND WS-Policy 1.2 |
|---|---|---|---|
| WS-SecurityPolicy 1.2 OASIS standard | Y | Y | Y |
| WS-SecurityPolicy 1.2 (WebLogic Server 10.0) | Y | Y | Y |
| WS-SecurityPolicy 1.2 OASIS standard AND WS-SecurityPolicy 1.2 (WebLogic Server 10.0) | Y | Y | Y |
| WebLogic Server 9.x-style | Y | Y | N |

*Table 2–1 (Cont.) Version-Independent Matrix*

| Security Policy Versions | WS-Policy 1.5 | WS-Policy 1.2 | WS-Policy 1.5 AND WS-Policy 1.2 |
|---|---|---|---|
| WebLogic Server 9.x-style AND WS-SecurityPolicy 1.2 OASIS standard or WS-SecurityPolicy 1.2 (WebLogic Server 10.0) | N | N | N |

If the client program wants to know what version of the policy or security policy is used, use the versioning API to return the namespace and versioning information.

## 2.4 Configuring Simple Message-Level Security: Main Steps

The following procedure describes how to configure simple message-level security for the Web services security runtime, a particular WebLogic Web service, and a client application that invokes an operation of the Web service. In this document, *simple message-level security* is defined as follows:

- The message-secured Web service uses the predefined WS-SecurityPolicy files to specify its security requirements, rather than a user-created WS-SecurityPolicy file. See Section 2.3, "Using Policy Files for Message-Level Security Configuration" for a description of these files.
- The Web service makes its associated security policy files publicly available by attaching them to its deployed WSDL, which is also publicly visible.
- The Web services runtime uses the out-of-the-box private key and X.509 certificate pairs, store in the default keystores, for its encryption and digital signatures, rather than its own key pairs. These out-of-the-box pairs are also used by the core WebLogic Server security subsystem for SSL and are provided for demonstration and testing purposes. For this reason Oracle highly recommends you use your own keystore and key pair in production. To use key pairs other than out-of-the-box pairs, see Section 2.4.3, "Using Key Pairs Other Than the Out-Of-The-Box SSL Pair".

  **Note:** If you plan to deploy the Web service to a cluster in which different WebLogic Server instances are running on different computers, you must use a keystore and key pair other than the out-of-the-box ones, even for testing purposes. The reason is that the key pairs in the default WebLogic Server keystore, DemoIdentity.jks, are not guaranteed to be the same across WebLogic Servers running on different machines.

  If you were to use the default keystore, the WSDL of the deployed Web service would specify the public key from one of these keystores, but the invoke of the service might actually be handled by a server running on a different computer, and in this case the server's private key would not match the published public key and the invoke would fail. This problem only occurs if you use the default keystore and key pairs in a cluster, and is easily resolved by using your own keystore and key pairs.

- The client invoking the Web service uses a username token to authenticate itself, rather than an X.509 token.

- The client invoking the Web service is a stand-alone Java application, rather than a module running in WebLogic Server.

Later sections describe some of the preceding scenarios in more detail, as well as additional Web services security uses cases that build on the simple message-level security use case.

It is assumed in the following procedure that you have already created a JWS file that implements a WebLogic Web service and you want to update it so that the SOAP messages are digitally signed and encrypted. It is also assumed that you use Ant build scripts to iteratively develop your Web service and that you have a working `build.xml` file that you can update with new information. Finally, it is assumed that you have a client application that invokes the non-secured Web service. If these assumptions are not true, see:

- *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*
- *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*

To configure simple message-level security for a WebLogic Web service:

1. Update your JWS file, adding WebLogic-specific `@Policy` and `@Policies` JWS annotations to specify the predefined policy files that are attached to either the entire Web service or to particular operations.

   See Section 2.4.2, "Updating the JWS File with @Policy and @Policies Annotations", which describes how to specify *any* policy file.

2. Recompile and redeploy your Web service as part of the normal iterative development process.

   See "Developing WebLogic Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* and "Developing WebLogic Web Services" in *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*.

3. Create a keystore used by the client application. Oracle recommends that you create one client keystore per application user.

   You can use the Cert Gen utility or Sun Microsystem's keytool utility (`http://java.sun.com/javase/6/docs/tooldocs/solaris/keytool.html`) to perform this step. For development purposes, the keytool utility is the easiest way to get started.

   See "Obtaining Private Keys, Digital Signatures, and Trusted Certificate Authorities" in *Securing Oracle WebLogic Server*.

4. Create a private key and digital certificate pair, and load it into the client keystore. The same pair will be used to both digitally sign the client's SOAP request and encrypt the SOAP responses from WebLogic Server.

   Make sure that the certificate's key usage allows both encryption and digital signatures. Also see Section 2.4.1, "Ensuring That WebLogic Server Can Validate the Client's Certificate" for information about how WebLogic Server ensures that the client's certificate is valid.

   **Note:** Oracle requires a key length of 1024 bits or larger.

   You can use Sun Microsystem's Keytool utility (`http://java.sun.com/javase/6/docs/tooldocs/solaris/keytool.html`) to perform this step.

See "Obtaining Private Keys, Digital Signatures, and Trusted Certificate Authorities" in *Securing Oracle WebLogic Server*.

5. Using the Administration Console, create users for authentication in your security realm.

   See *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

6. Update your client application by adding the Java code to invoke the message-secured Web service.

   See Section 2.15, "Using a Client-Side Security Policy File".

7. Recompile your client application.

   See *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* and *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server* for general information.

See the following sections for information about additional Web service security uses cases that build on the basic message-level security use case:

- Section 2.4.3, "Using Key Pairs Other Than the Out-Of-The-Box SSL Pair"
- Section 2.7, "Creating and Using a Custom Policy File"
- Section 2.9, "Configuring and Using Security Contexts and Derived Keys"
- Section 2.10, "Associating Policy Files at Runtime Using the Administration Console"
- Section 2.11, "Using Security Assertion Markup Language (SAML) Tokens For Identity"
- Section 2.5.1, "Invoking a Web Service From a Client Running in a WebLogic Server Instance"
- Section 2.12, "Associating a Web Service with a Security Configuration Other Than the Default"

See Section 2.14, "Using System Properties to Debug Message-Level Security" for information on debugging problems with your message-secured Web service.

### 2.4.1 Ensuring That WebLogic Server Can Validate the Client's Certificate

You must ensure that WebLogic Server is able to validate the X.509 certificate that the client uses to digitally sign its SOAP request, and that WebLogic Server in turn uses to encrypt its SOAP responses to the client. Do one of the following:

- Ensure that the client application obtains a digital certificate that WebLogic Server automatically trusts, because it has been issued by a trusted certificate authority.
- Create a certificate registry that lists all the individual certificates trusted by WebLogic Server, and then ensure that the client uses one of these registered certificates.

For more information, see "SSL Certificate Validation" in *Securing WebLogic Server*.

### 2.4.2 Updating the JWS File with @Policy and @Policies Annotations

Use the `@Policy` and `@Policies` annotations in your JWS file to specify that the Web service has one or more policy files attached to it. You can use these annotations at either the class or method level.

> **Note:** If you specify a transport-level security policy for your Web service, it must be at the class level.
>
> In addition, the transport-level security policy must apply to both the inbound and outbound directions. That is, you cannot have HTTPS for inbound and HTTP for outbound.

See Section 2.4.2.1, "Loading a Policy From the CLASSPATH" for an additional policy option.

This release also support the `@SecurityPolicy` annotation that is used for integrating Oracle Web Services Manager (WSM) WS-Security policies into the WebLogic Server environment, as described in Appendix A, "Using Oracle Web Services Manager Security Policies".

The `@Policies` annotation simply groups two or more `@Policy` annotations together. Use the `@Policies` annotation if you want to attach two or more policy files to the class or method. If you want to attach just one policy file, you can use `@Policy` on its own.

The `@Policy` annotation specifies a single policy file, where it is located, whether the policy applies to the request or response SOAP message (or both), and whether to attach the policy file to the public WSDL of the service.

Use the `uri` attribute to specify the location of the policy file, as described below:

- To specify one of the predefined security policy files that are installed with WebLogic Server, use the `policy:` prefix and the name of one of the policy files, as shown in the following example:

  ```
  @Policy(uri="policy:Wssp1.2-2007-Https-BasicAuth.xml")
  ```

  If you use the predefined policy files, you do not have to create one yourself or package it in an accessible location. For this reason, Oracle recommends that you use the predefined policy files whenever you can.

  See Section 2.3, "Using Policy Files for Message-Level Security Configuration" for information on the various types of message-level security provided by the predefined policy files.

- To specify a user-created policy file, specify the path (relative to the location of the JWS file) along with its name, as shown in the following example:

  ```
  @Policy(uri="../policies/MyPolicy.xml")
  ```

  In the example, the `MyPolicy.xml` file is located in the `policies` sibling directory of the one that contains the JWS file.

- You can also specify a policy file that is located in a shared J2EE library; this method is useful if you want to share the file amongst multiple Web services packaged in different J2EE archives.

  > **Note:** In this case, it is assumed that the policy file is in the META-INF/policies or WEB-INF/policies directory of the shared J2EE library. Be sure, when you package the library, that you put the policy file in this directory.

To specify a policy file in a shared J2EE library, use the `policy` prefix and then the name of the policy file, as shown in the following example:

```
@Policy(uri="policy:MySharedPolicy.xml")
```

See "Creating Shared Java EE Libraries and Optional Packages" in *Developing Applications for Oracle WebLogic Server* for information on creating shared libraries and setting up your environment so the Web service can find the shared policy files.

You can also set the following attributes of the `@Policy` annotation:

- `direction` specifies whether the policy file should be applied to the request (inbound) SOAP message, the response (outbound) SOAP message, or both. The default value if you do not specify this attribute is both. The `direction` attribute accepts the following values:
  - Policy.Direction.both
  - Policy.Direction.inbound
  - Policy.Direction.outbound
- `attachToWsdl` specifies whether the policy file should be attached to the WSDL file that describes the public contract of the Web service. The default value of this attribute is `false`.

The following example shows how to use the `@Policy` and `@Policies` JWS annotations, with the relevant sections shown in bold:

***Example 2–1 Using @Policy and @Policies Annotations***

```
package wssp12.wss10;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Policy;
import weblogic.jws.Policies;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.Oneway;

/**
 * This Web service demonstrates how to use WS-SecurityPolicy 1.2
 * to enable message-level security specified in WS-Security 1.0.
 *
 * The service authenticates the client with a username token.
 * Both the request and response messages are signed and encrypted with X509
 certificates.
 *
*/
@WebService(name="Simple", targetNamespace="http://example.org")
@WLHttpTransport(contextPath="/wssp12/wss10",
 serviceUri="UsernameTokenPlainX509SignAndEncrypt")
@Policy(uri="policy:Wssp1.2-2007-Wss1.0-UsernameToken-Plain-X509-Basic256.xml")
public class UsernameTokenPlainX509SignAndEncrypt {

  @WebMethod
  @Policies({
      @Policy(uri="policy:Wssp1.2-2007-SignBody.xml"),
      @Policy(uri="policy:Wssp1.2-2007-EncryptBody.xml")})
  public String echo(String s) {
```

```
    return s;
  }

  @WebMethod
  @Policies({
      @Policy(uri="policy:Wssp1.2-2007-SignBody.xml"),
      @Policy(uri="policy:Wssp1.2-2007-Sign-Wsa-Headers.xml")})
  public String echoWithWsa(String s) {
    return s;
  }

  @WebMethod
  @Policy(uri="policy:Wssp1.2-2007-SignBody.xml",
 direction=Policy.Direction.inbound)
  @Oneway
  public void echoOneway(String s) {
    System.out.println("s = " + s);
  }

  @WebMethod
  @Policies({
      @Policy(uri="policy:Wssp1.2-2007-Wss1.0-X509-Basic256.xml",
direction=Policy.Direction.inbound),
      @Policy(uri="policy:Wssp1.2-2007-SignBody.xml",
direction=Policy.Direction.inbound)
  })
  @Oneway
  public void echoOnewayX509(String s) {
    System.out.println("X509SignEncrypt.echoOneway: " + s);
  }
}
```

The following section of the example is the binding policy for the Web service, specifying the policy:

```
@WebService(name="Simple", targetNamespace="http://example.org")
@WLHttpTransport(contextPath="/wssp12/wss10",
 serviceUri="UsernameTokenPlainX509SignAndEncrypt")
@Policy(uri="policy:Wssp1.2-2007-Wss1.0-UsernameToken-Plain-X509-Basic256.xml")
```

In the example, security policy files are attached to the Web service at the method level. The specified policy files are those predefined with WebLogic Server, which means that the developers do not need to create their own files or package them in the corresponding archive.

The `Wssp1.2-2007-SignBody.xml` policy file specifies that the body and WebLogic system headers of both the request and response SOAP message be digitally signed. The `Wssp1.2-2007-EncryptBody.xml` policy file specifies that the body of both the request and response SOAP messages be encrypted.

#### 2.4.2.1 Loading a Policy From the CLASSPATH

This release of WebLogic Server includes a 'load policy as resource from CLASSPATH' feature. This feature allows you to copy a policy file to the root directory of your Web application and then reference it directly by its name (for example, mypolicy.xml') from an @POLICY annotation in your JWS file.

To enable this feature, start WebLogic Server with `-Dweblogic.wsee.policy.LoadFromClassPathEnabled=true`

If you enable this feature, be aware of the following caveat: If you were to then move the policy file to the WEB-INF/policies directory, the same 'mypolicy.xml' reference in the @POLICY annotation will no longer work. You would need to add the policy prefix to the @POLICY annotation; for example, 'policy:mypolicy.xml'.

### 2.4.3 Using Key Pairs Other Than the Out-Of-The-Box SSL Pair

In the simple message-level configuration procedure, documented in Section 2.4, "Configuring Simple Message-Level Security: Main Steps", it is assumed that the Web services runtime uses the private key and X.509 certificate pair that is provided out-of-the-box with WebLogic Server; this same key pair is also used by the core security subsystem for SSL and is provided mostly for demonstration and testing purposes. In production environments, the Web services runtime typically uses its own two private key and digital certificate pairs, one for signing and one for encrypting SOAP messages.

The following procedure describes the additional steps you must take to enable this use case.

1. Obtain two private key and digital certificate pairs to be used by the Web services runtime. One of the pairs is used for digitally signing the SOAP message and the other for encrypting it.

   Although not required, Oracle recommends that you obtain two pairs that will be used *only* by WebLogic Web services. You must also ensure that both of the certificate's key usage matches what you are configuring them to do. For example, if you are specifying that a certificate be used for encryption, be sure that the certificate's key usage is specified as for encryption or is undefined. Otherwise, the Web services security runtime will reject the certificate.

   **Note:** Oracle requires that the key length be 1024 bits or larger.

   You can use the Cert Gen utility or Sun Microsystem's keytool utility (`http://java.sun.com/javase/6/docs/tooldocs/solaris/keytool.html`) to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   See "Obtaining Private Keys, Digital Signatures, and Trusted Certificate Authorities" in *Securing Oracle WebLogic Server*.

2. Create, if one does not currently exist, a custom identity keystore for WebLogic Server and load the private key and digital certificate pairs you obtained in the preceding step into the identity keystore.

   If you have already configured WebLogic Server for SSL, then you have already created an identity keystore that you can also use in this step.

   You can use WebLogic's `ImportPrivateKey` utility and Sun Microsystem's keytool utility (`http://java.sun.com/javase/6/docs/tooldocs/solaris/keytool.html`) to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   See "Creating a Keystore and Loading Private Keys and Trusted Certificate Authorities Into the Keystore" in *Securing Oracle WebLogic Server*.

3. Using the Administration Console, configure WebLogic Server to locate the keystore you created in the preceding step. If you are using a keystore that has

already been configured for WebLogic Server, you do not need to perform this step.

See "Configuring Keystores for Production" in *Securing Oracle WebLogic Server*.

4. Using the Administration Console, create the default Web service security configuration, which must be named `default_wss`. The default Web service security configuration is used by *all* Web services in the domain unless they have been explicitly programmed to use a different configuration.

   See "Create a Web Service Security Configuration" in the *Oracle WebLogic Server Administration Console Help*.

5. Update the default Web services security configuration you created in the preceding step to use one of the private key and digital certificate pairs for digitally signing SOAP messages.

   See "Specify the key pair used to sign SOAP messages" in *Oracle WebLogic Server Administration Console Help*. In the procedure, when you create the properties used to identify the keystore and key pair, enter the exact value for the Name of each property (such as `IntegrityKeyStore`, `IntegrityKeyStorePassword`, and so on), but enter the value that identifies your own previously-created keystore and key pair in the Value fields.

6. Similarly, update the default Web services security configuration you created in a preceding step to use the second private key and digital certificate pair for encrypting SOAP messages.

   See "Specify the key pair used to encrypt SOAP messages" in *Oracle WebLogic Server Administration Console Help*. In the procedure, when you create the properties used to identify the keystore and key pair, enter the exact value for the Name of each property (such as `ConfidentialityKeyStore`. `ConfidentialityKeyStorePassword`, and so on), but enter the value that identifies your own previously-created keystore and key pair in the Value fields.

## 2.5 Updating a Client Application to Invoke a Message-Secured Web Service

When you update your Java code to invoke a message-secured Web service, you must load a private key and digital certificate pair from the client's keystore and pass this information, along with a username and password for user authentication if so required by the security policy, to the secure WebLogic Web service being invoked.

If the security policy file of the Web service specifies that the SOAP request must be encrypted, then the Web services client runtime automatically gets the server's certificate from the policy file that is attached to the WSDL of the service, and uses it for the encryption. If, however, the policy file is not attached to the WSDL, or the entire WSDL itself is not available, then the client application must use a client-side copy of the policy file; for details, see Section 2.15, "Using a Client-Side Security Policy File".

Example 2–2 shows a Java client application under JAX-RPC that invokes the message-secured WebLogic Web service described by the JWS file in Section 4.2, "Updating the JWS File With the Security-Related Annotations". The client application takes five arguments:

- Client username for client authentication
- Client password for client authentication
- Client private key file

- Client digital certificate
- WSDL of the deployed Web service

The security-specific code in the sample client application is shown in bold (and described after the example):

***Example 2–2 Client Application Invoking a Message-Secured Web Service Under JAX-RPC***

```
package examples.webservices.security_jws.client;
import weblogic.security.SSL.TrustManager;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
import javax.xml.rpc.Stub;
import java.util.List;
import java.util.ArrayList;
import java.security.cert.X509Certificate;
/**
 *  Copyright © 1996, 2008, Oracle and/or its affiliates.
 *  All rights reserved.
 */
public class SecureHelloWorldClient {
  public static void main(String[] args) throws Throwable {
    //username or password for the UsernameToken
    String username = args[0];
    String password = args[1];
    //client private key file
    String keyFile = args[2];
    //client certificate
    String clientCertFile = args[3];
    String wsdl = args[4];
    SecureHelloWorldService service = new SecureHelloWorldService_Impl(wsdl +
"?WSDL" );
    SecureHelloWorldPortType port = service.getSecureHelloWorldServicePort();
    //create credential provider and set it to the Stub
    List credProviders = new ArrayList();
    //client side BinarySecurityToken credential provider -- x509
    CredentialProvider cp = new ClientBSTCredentialProvider(clientCertFile,
keyFile);
    credProviders.add(cp);
    //client side UsernameToken credential provider
    cp = new ClientUNTCredentialProvider(username, password);
    credProviders.add(cp);
    Stub stub = (Stub)port;
    stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);
    stub._setProperty(WSSecurityContext.TRUST_MANAGER,
      new TrustManager(){
         public boolean certificateCallback(X509Certificate[] chain, int
validateErr){
           return true;
         }
       } );
    String response = port.sayHello("World");
    System.out.println("response = " + response);
  }
}
```

The main points to note about the preceding code are:

- Import the WebLogic security `TrustManager` API:

```
import weblogic.security.SSL.TrustManager;
```

- Import the following WebLogic Web services security APIs to create the needed client-side credential providers, as specified by the policy files that are associated with the Web service:

```
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
```

- Use the `ClientBSTCredentialProvider` WebLogic API to create a binary security token credential provider from the client's certificate and private key:

```
 CredentialProvider cp =
    new ClientBSTCredentialProvider(clientCertFile, keyFile);
```

- Use the `ClientUNTCredentialProvider` WebLogic API to create a username token from the client's username and password, which are also known by WebLogic Server:

```
cp = new ClientUNTCredentialProvider(username, password);
```

- Use the `WSSecurityContext.CREDENTIAL_PROVIDER_LIST` property to pass a `List` object that contains the binary security and username tokens to the JAX-RPC Stub:

```
stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders)
```

  For JAX-WS, you might code this as follows:

```
import javax.xml.ws.BindingProvider;
:
Map<String, Object> requestContext = ((BindingProvider)
port).getRequestContext();
requestContext.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);
```

- Use the `weblogic.security.SSL.TrustManager` WebLogic security API to verify that the certificate used to encrypt the SOAP request is valid. The Web services client runtime gets this certificate from the deployed WSDL of the Web service, which in production situations is not automatically trusted, so the client application must ensure that it is okay before it uses it to encrypt the SOAP request:

```
stub._setProperty(WSSecurityContext.TRUST_MANAGER,
      new TrustManager(){
         public boolean certificateCallback(X509Certificate[] chain, int
validateErr){
           return true;
         }
       } );
```

  For JAX-WS, you might code this as follows:

```
requestContext.put(WSSecurityContext.TRUST_MANAGER,
     new TrustManager() {
         public boolean certificateCallback(X509Certificate[] chain,
int validateErr) {
           return true;
         }
```

```
        });
```

This example shows the TrustManager API on the client side. The Web service application must implement proper verification code to ensure security.

Example 2–3 shows the same Java client application under JAX-WS that invokes the message-secured Web service. The JAX-WS specific code in the sample client application is shown in bold.

***Example 2–3 Client Application Invoking a Message-Secured Web Service under JAX-WS***

```
package examples.webservices.security_jaxws.client;
import weblogic.security.SSL.TrustManager;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
import javax.xml.ws.BindingProvider;
import java.util.List;
import java.util.Map;
import java.util.ArrayList;
import java.security.cert.X509Certificate;/**
 *  Copyright © 1996, 2010, Oracle and/or its affiliates.
 *  All rights reserved.
 */
public class SecureHelloWorldJaxwsClient {
   public static void main(String[] args) throws Throwable {
      //username or password for the UsernameToken
      String username = args[0];
      String password = args[1];
      //client private key file
      String keyFile = args[2];
      //client certificate
      String clientCertFile = args[3];
      String wsdl = args[4];
      SecureHelloWorldService service = new SecureHelloWorldService_Impl(wsdl +
"?WSDL" );
      SecureHelloWorldPortType port = service.getSecureHelloWorldServicePort();
      //create credential provider and set it to the request context
      List credProviders = new ArrayList();
      //client side BinarySecurityToken credential provider -- x509
      CredentialProvider cp = new ClientBSTCredentialProvider(clientCertFile,
keyFile);
      credProviders.add(cp);
      //client side UsernameToken credential provider
      cp = new ClientUNTCredentialProvider(username, password);
      credProviders.add(cp);
      Map<String, Object> requestContext = ((BindingProvider)
port).getRequestContext();
      requestContext.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
credProviders);
      requestContext.put(WSSecurityContext.TRUST_MANAGER, new TrustManager() {
          public boolean certificateCallback(X509Certificate[] chain,
            int validateErr) {
           // need to validate if the server cert can be trusted
           return true;
         }
      });
      String response = port.sayHello("World");
```

```
        System.out.println("response = " + response);
    }
}
```

### 2.5.1 Invoking a Web Service From a Client Running in a WebLogic Server Instance

In the simple Web services configuration procedure, described in Section 2.4, "Configuring Simple Message-Level Security: Main Steps", it is assumed that a *stand-alone* client application invokes the message-secured Web service. Sometimes, however, the client is itself running in a WebLogic Server instance, as part of an EJB, a servlet, or another Web service. In this case, you can use the core WebLogic Server security framework to configure the credential providers and trust manager so that your EJB, servlet, or JWS code contains only the simple invoke of the secured operation and no other security-related API usage.

The following procedure describes the high level steps you must perform to make use of the core WebLogic Server security framework in this use case.

1. In your EJB, servlet, or JWS code, invoke the Web service operation as if it were *not* configured for message-level security. Specifically, do not create a `CredentialProvider` object that contains username or X.509 tokens, and do not use the `TrustManager` core security API to validate the certificate from the WebLogic Server hosting the secure Web service. The reason you should not use these APIs in your client code is that the Web services runtime will perform this work for you.
2. Using the Administration Console, configure the required credential mapping providers of the core security of the WebLogic Server instance that hosts your client application. The list of required credential mapper providers depends on the policy file that is attached to the Web service you are invoking. Typically, you must configure the credential mapper providers for both username/password and X.509 certificates. See Section 2.13, "Valid Class Names and Token Types for Credential Provider" for the possible values.

   **Note:** WebLogic Server includes a credential mapping provider for username/passwords and X.509. However, only username/password is configured by default.

3. Using the Administration Console, create the actual credential mappings in the credential mapping providers you configured in the preceding step. You must map the user principal, associated with the client running in the server, to the credentials that are valid for the Web service you are invoking. See "Configuring a WebLogic Credential Mapping Provider" in *Securing Oracle WebLogic Server*.
4. Using the Administration Console, configure the core WebLogic Server security framework to trust the X.509 certificate of the invoked Web service. See "Configuring the Certificate Lookup and Validation Framework" in *Securing Oracle WebLogic Server*.

You are not required to configure the core WebLogic Server security framework, as described in this procedure, if your client application does not want to use the out-of-the-box credential provider and trust manager. Rather, you can override all of this configuration by using the same APIs in your EJB, servlet, and JWS code as in the stand-alone Java code described in Section 2.15, "Using a Client-Side Security Policy File". However, using the core security framework standardizes the WebLogic Server configuration and simplifies the Java code of the client application that invokes the Web service.

## 2.6 Example of Adding Security to a JAX-WS Web Service

This section provides a simple example of adding security to a JAX-WS Web service. The example attaches four policies:

- Wssp1.2-2007-SignBody.xml
- Wssp1.2-2007-EncryptBody.xml
- Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml
- Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic128.xml

The examples include extensive inline comments in the code.

Example 2–4 shows the Web service code.

Note that this Web service implements *attachToWsdl=false*, and therefore the Web service client needs to load a client-side version of the policy, as shown in Example 2–5.

***Example 2–4 Web Service SignEncrypt.java***

```
package signencrypt;

import java.io.File;

import weblogic.jws.Policies;
import weblogic.jws.Policy;
import weblogic.jws.security.WssConfiguration;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.MTOM;

import com.sun.xml.ws.developer.SchemaValidation;

/**
 *
 * Webservice which accepts a SOAP Message which is Signed And
 *         Encrypted Uses the WS-Policy 1.2
 */


@WebService(name = "SignEncrypt", portName = "SignEncryptPort", serviceName =
"SignEncrypt", targetNamespace = "http://signencrypt")
@BindingType(value = "http://schemas.xmlsoap.org/wsdl/soap/http")
// Domain Level WebserviceSecurity Configuration
@WssConfiguration(value = "Basic-UNT")
@MTOM()
//@SchemaValidation

public class SignEncrypt {

  @Policies( {
  @Policy(uri = "policy:Wssp1.2-2007-SignBody.xml", attachToWsdl=false),
      @Policy(uri = "policy:Wssp1.2-2007-EncryptBody.xml", attachToWsdl=false),
    /*
     * WSS 1.1 X509 with symmetric binding and authentication with plain-text
     * Username Token which is encrypted and signed using the Symmetric key
```

```
     */
    /* Use Basic-UNT WssConfiguration */
      @Policy(uri =
"policy:Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml",
attachToWsdl=false)
  /*
   * The client side public certificate and private key is not required in
   * this scenario. Username token with plain text password is sent in the
   * request for authentication, and signed and encrypted with the symmetric
   * key. The symmetric key is encrypted by the server's public key. The client
   * also signs and encrypts the request header elements and body with the
   * symmetric key. The server signs and encrypts the response body with the
   * symmetric key. Both request and response messages include the signed time
   * stamps. The encryption method is Basic128.
   */
  /* Use untx509webservicesecurity WssConfiguration */

  /*
   * @Policy(uri =
   * "policy:Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic128.xml")
   */})
  @WebMethod()
  public String echoString(String input) {
    String result = "[SignEncrypt.echoString]: " + input;
    System.out.println(result);
    return result;
  }

   @WebMethod()
  public String echoStringWithoutSecurity(String input) {
    String result = "[SignEncrypt.echoString]: " + input;
    System.out.println(result);
    return result;
  }

  @WebMethod()
  public byte[] echoStringAsByteArray(String data) {
    System.out.println("echoByteArray data: " + data);
    byte[] output = data.getBytes();
    System.out.println("Output Length : " + output.length + " Output: " +
output.toString());
    return data.getBytes();
  }

   @Policies( {
  @Policy(uri = "policy:Wssp1.2-2007-SignBody.xml", attachToWsdl=false),
      @Policy(uri = "policy:Wssp1.2-2007-EncryptBody.xml", attachToWsdl=false),
    /*
     * WSS 1.1 X509 with symmetric binding and authentication with plain-text
     * Username Token which is encrypted and signed using the Symmetric key
     */
    /* Use Basic-UNT WssConfiguration */
      @Policy(uri =
"policy:Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml",
attachToWsdl=false)
  /*
   * The client side public certificate and private key is not required in
   * this scenario. Username token with plain text password is sent in the
   * request for authentication, and signed and encrypted with the symmetric
   * key. The symmetric key is encrypted by the server's public key. The client
```

```
    * also signs and encrypts the request header elements and body with the
    * symmetric key. The server signs and encrypts the response body with the
    * symmetric key. Both request and response messages include the signed time
    * stamps. The encryption method is Basic128.
    */
   /* Use untx509webservicesecurity WssConfiguration */

   /*
   * @Policy(uri =
   * "policy:Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic128.xml")
   */})
 @WebMethod()
 public byte[] echoByteArrayWithSecurity(byte[] inputData) {
    System.out.println("echoByteArrayWithSecurity data: " + inputData.length + "
bytes");
    return inputData;
 }

 @WebMethod()
 public byte[] echoByteArray(byte[] inputData) {
    System.out.println("echoByteArray data: " + inputData);
    return inputData;
 }

 @WebMethod()
 public DataHandler getDataHandler(String fileName) {

    DataHandler handler = null;
    try {
    File file = new File(fileName);
    System.out.println("file: " + file.getCanonicalPath() + ", " +
file.getPath());

    FileDataSource fileDataSource = new FileDataSource(file);
    handler = new DataHandler(fileDataSource);

    } catch(Exception e) {
      System.out.println("Error Creating Data Handelr:  " + e.getMessage());
    }

    return handler;

  }


  @WebMethod()
 @Policies( {
    @Policy(uri = "policy:Wssp1.2-2007-SignBody.xml", attachToWsdl=false),
    @Policy(uri = "policy:Wssp1.2-2007-EncryptBody.xml", attachToWsdl=false),
    /*
     * WSS 1.1 X509 with symmetric binding and authentication with plain-text
     * Username Token which is encrypted and signed using the Symmetric key
     */
    /* Use Basic-UNT WssConfiguration */
        @Policy(uri =
"policy:Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml",
attachToWsdl=false)
   /*
    * The client side public certificate and private key is not required in
    * this scenario. Username token with plain text password is sent in the
```

```
   * request for authentication, and signed and encrypted with the symmetric
   * key. The symmetric key is encrypted by the server's public key. The client
   * also signs and encrypts the request header elements and body with the
   * symmetric key. The server signs and encrypts the response body with the
   * symmetric key. Both request and response messages include the signed time
   * stamps. The encryption method is Basic128.
   */
 /* Use untx509webservicesecurity WssConfiguration */
  /*
  * @Policy(uri =
  * "policy:Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic128.xml")
  */})
 public DataHandler getDataHandlerWithSecurity(String fileName) {

   DataHandler handler = null;
   try {
   File file = new File(fileName);
   System.out.println("file: " + file.getCanonicalPath() + ", " +
file.getPath());

   FileDataSource fileDataSource = new FileDataSource(file);
   handler = new DataHandler(fileDataSource);

   } catch(Exception e) {
     System.out.println("Error Creating Data Handelr:  " + e.getMessage());
   }

   return handler;

 }

}
```

Example 2–5 shows an example of using the *weblogic.jws.jaxws.ClientPolicyFeature* class to load client-side policies.

The example includes extensive inline comments.

***Example 2–5 SOAClient.java***

```
package signencrypt.client;
import weblogic.jws.jaxws.ClientPolicyFeature;
import weblogic.jws.jaxws.policy.InputStreamPolicySource;
import weblogic.security.SSL.TrustManager;
import weblogic.wsee.policy.runtime.BuiltinPolicyFinder;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.bst.StubPropertyBSTCredProv;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
import weblogic.wsee.security.util.CertUtils;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.xml.crypto.wss.provider.CredentialProvider;

import soa.client.Bpelprocess1ClientEp;
import soa.client.BPELProcess1;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.InputStream;
```

```
import java.io.OutputStream;
import java.net.URL;
import java.security.cert.X509Certificate;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.WebServiceFeature;
import javax.xml.ws.soap.MTOMFeature;

public class SOAClient {

  private final static boolean debug = true;

  private final static String endpointURL =
    "http://....com:8001/soa-infra/services/default/soa/bpelprocess1_client_ep";
  private final static String certsDir = "C:/webservices/server/keystores";

  private final static String serverKeyStoreName = "default-keystore.jks";
  private final static String serverKeyStorePass = "...";
  private final static String serverCertAlias = "alice";
  private final static String serverKeyPass = "...";

  private final static String username = "weblogic";
  private final static String password = "...";

  private final static String fileName =
    "C:/webservices/farallon/owsm-interop/mtom.JPG";

  private final static String outputFileName =
    "C:/webservices/farallon/owsm-interop/output.jpg";

  private final static String[] clientPolicyFileNames =
    {

"./policy/Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml",
      "./policy/Wssp1.2-2007-SignBody.xml",
      "./policy/Wssp1.2-2007-EncryptBody.xml" };

  private BPELProcess1 port = null;

  /**
   * Create the Stub/Port and set the Stub/Port with Client Side Security Policy
   * Feature and MTOM Feature.
   * @throws Exception
   */

  private void createStubWithClientPolicy() throws Exception {

    URL url = new URL(endpointURL + "?WSDL");

    QName serviceName =
      new QName("http://xmlns.oracle.com/SOASecurity/soa/BPELProcess1",
        "bpelprocess1_client_ep");

    Bpelprocess1ClientEp service = new Bpelprocess1ClientEp(url, serviceName);
```

```
    QName operationName =
      new QName("http://xmlns.oracle.com/SOASecurity/soa/BPELProcess1",
"process");

    ClientPolicyFeature policyFeature = new ClientPolicyFeature();

    // Set the Client Side Policy on the operation with QName <operationName>

policyFeature.setEffectivePolicyForOperation(operationName, new
InputStreamPolicySource(getPolicyInputStreamArray(clientPolicyFileNames)
));
    MTOMFeature mtomFeature = new MTOMFeature();

    WebServiceFeature[] features = { policyFeature, mtomFeature };
    // WebServiceFeature[] features = { mtomFeature };
    //WebServiceFeature[] features = {policyFeature};

    port = service.getBPELProcess1Pt(features);
  }

  /**
  * Setup the Client Port/Stub used to invoke the webservice with Security
  *
  * @throws Exception
  */
 private void setUp() throws Exception {
   createStubWithClientPolicy();
   /**
    * Get the Server Public Certificate to Encrypt the Symmetric Key or the
    * SOAP Message
    */
   /**
    * Get the Server Public Certificate to Verify the Signature of the
    * Symmetric Key or the SOAP Message
    */
    X509Certificate serverCert =
     (X509Certificate) CertUtils.getCertificate(
       certsDir + "/" + serverKeyStoreName, serverKeyStorePass,
       serverCertAlias, "JKS").get(0);
   List<CredentialProvider> credProviders =
     new ArrayList<CredentialProvider>();
   /*
    * Set up UserNameToken
    */
   credProviders.add(new ClientUNTCredentialProvider(username.getBytes(),
     password.getBytes()));
   Map<String, Object> rc = ((BindingProvider) port).getRequestContext();
    /*
    * For Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml
    * there is no need to specify the client side public certificate and
    * private key as this is a symmetric key use case. serverCert is used to
    * encrypt the Symmetric Key/Keys
    */
   rc.put(StubPropertyBSTCredProv.SERVER_ENCRYPT_CERT, serverCert);
   rc.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);
   rc.put(WSSecurityContext.TRUST_MANAGER, new TrustManager() {
     public boolean certificateCallback(X509Certificate[] chain,
       int validateErr) {
       // need to validate if the server cert can be trusted
```

```
      System.out.println("Validating Server Certificate");
      return true;
    }
  });

 }
/**
 * Returns an array of InputStreams of the policy files
 *
 * @param policyNames
 * @return array of InputStreams of Policy's
 * @throws FileNotFoundException
 */
private InputStream[] getPolicyInputStreamArray(String[] policyNames)
  throws FileNotFoundException {
  InputStream[] inpStreams = new InputStream[policyNames.length];
  for (int k = 0; k < policyNames.length; k++) {
    System.out.println("policy name: " + policyNames[k]);
    inpStreams[k] = getPolicyInputStream(policyNames[k]);
  }
  return inpStreams;
}
 /**
 * Returns an InputStream of the policy file
 *
 * @param myPolicyName
 * @return
 * @throws FileNotFoundException
 */
private InputStream getPolicyInputStream(String myPolicyName)
  throws FileNotFoundException {
  return new FileInputStream(myPolicyName);
}
/**
 * Invoke the webservice at endpointURL
 *
(http://....:9003/soa-infra/services/default/soa/bpelprocess1_client_ep)
 *
 * @throws Exception
 */
private void invokeProcess() throws Exception {
  InputStream inputstream = null;
  OutputStream outputstream = null;
  try {

    File file = new File(fileName);
    File outputFile = new File(outputFileName);

    inputstream = new BufferedInputStream(new FileInputStream(file));
    int bytesAvailable = -1;
    int counter = 0;
    int bytesRead = 0;
    int fileSize = (int) file.length();

    byte[] fileInBytes = new byte[fileSize];

    bytesRead = inputstream.read(fileInBytes);
    System.out.println("bytesRead: " + bytesRead + ", fileSize: " + fileSize + "
fileInBytes: " + fileInBytes.length);
```

```
      byte[] result = port.process(fileInBytes);
      /*byte[] input = "Hello".getBytes();
      System.out.println("input length : "+ input.length);

      byte[] result = port.process(input);*/
      if (!outputFile.exists()) {
        outputFile.createNewFile();
      }

      outputstream = new BufferedOutputStream(new FileOutputStream(outputFile));

      if (result != null) {
        System.out.println("Result Length: " + result.length);
      } else {
        System.out.println("result is null");
      }
      outputstream.write(result);

      // System.out.println(result);
    } catch (Exception e) {
      System.out.println("Error Creating Data Handler:  " + e.getMessage());
    } finally {

      if (inputstream != null) {
        inputstream.close();
      }

     if (outputstream != null) {
        outputstream.close();
      }
    }
  }
  public static void main(String[] args) {
    try {
      SOAClient client = new SOAClient();
      client.setUp();
      //client.createStubWithClientPolicy();
      client.invokeProcess();
    } catch (Exception e) {
      System.out.println("Error calling SOA Webservice: " + e.getMessage());
      if (debug) {
        e.printStackTrace();
      }
    }
  }
}
```

## 2.7 Creating and Using a Custom Policy File

Although WebLogic Server includes a number of predefined Web services security policy files that typically satisfy the security needs of most programmers, you can also create and use your own WS-SecurityPolicy file if you need additional configuration. See Section 2.3, "Using Policy Files for Message-Level Security Configuration" for general information about security policy files and how they are used for message-level security configuration.

**Note:** Use of element-level security always requires one or more custom policy files to specify the particular element path and name to be secured.

When you create a custom policy file, you can separate out the three main security categories (authentication, encryption, and signing) into three separate policy files, as do the predefined files, or create a single policy file that contains all three categories. You can also create a custom policy file that changes just one category (such as authentication) and use the predefined files for the other categories (`Wssp1.2-2007-SignBody.xml, Wssp1.2-SignBody.xml` and `Wssp1.2-2007-EncryptBody, Wssp1.2-EncryptBody`). In other words, you can mix and match the number and content of the policy files that you associate with a Web service. In this case, however, you must always ensure yourself that the multiple files do not contradict each other.

Your custom policy file needs to comply with the standard format and assertions defined in WS-SecurityPolicy 1.2. Note, however, that this release of WebLogic Server does not completely implement WS-SecurityPolicy 1.2. For more information, see Section 2.18, "Unsupported WS-SecurityPolicy 1.2 Assertions". The root element of your WS-SecurityPolicy file must be `<Policy>`.

The following namespace declaration is recommended in this release:

```
<wsp:Policy
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
>
. . .
</wsp:Policy>
```

WLS also supports other namespaces for Security Policy. For example, the following two namespaces are also supported:

```
<wsp:Policy
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512"
>
. . .
</wsp:Policy>
```

or

```
<wsp:Policy
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
>
. . .
</wsp:Policy>
```

You can also use the predefined WS-SecurityPolicy files as templates to create your own custom files. See Section 2.16, "Using WS-SecurityPolicy 1.2 Policy Files".

## 2.8 Configuring the WS-Trust Client

WebLogic Server implements a WS-Trust client that retrieves security tokens from a Security Token Service (STS) for use in Web Services Security. This WS-Trust client is used internally by the client side WebLogic Server Web service runtime.

You can configure the WS-Trust client as follows:

- Through properties on the Web service client stub for a standalone Web service client.
- Through MBean properties for a Web service client running on the server.

In releases prior to 10g Release 3 (10.3) of WebLogic Server, the WS-Trust client could use only security tokens from an STS that was co-located with a Web service and hosted by WebLogic Server. However, the STS now need only be accessible to the WS-Trust client; it does not need to be co-located.

The WS-Trust client in prior releases supported only WS-SecureConversation tokens. It now also supports SAML tokens.

### 2.8.1 Supported Token Types

Web Service Secure Conversation Language (WS-SecureConversation) and SAML tokens are supported. The tokens have the following namespace and URI:

- For WS-SecureConversation 1.3:

  ```
  http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512
  http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
  ```

- For WS-SecureConversation 1.2:

  ```
  http://schemas.xmlsoap.org/ws/2005/02/sc
  http://schemas.xmlsoap.org/ws/2005/02/sc/sct
  ```

- For SAML 1.1:

  ```
  urn:oasis:names:tc:SAML:1.0:assertion
  ```

  Supported confirmation method are sender-vouches and holder-of-key. Bearer confirmation and Symmetric holder-of-key are not supported.

- For SAML 2.0:

  ```
  urn:oasis:names:tc:SAML:2.0:assertion
  ```

  Supported confirmation methods are sender-vouches, holder-of-key and bearer. Symmetric holder-of-key is not supported.

### 2.8.2 Configuring WS-Trust Client Properties

You set some of the configuration properties specifically for the WS-Trust client; others are determined through configuration information generally present for a Web service client. For example, the type of token retrieved is determined by the security policy of the Web service that the Web service client is invoking.

The properties that you can explicitly set and the token type they apply to are as follows.

- STS URI (WS-SecureConversation and SAML)
- STS security policy (SAML)
- STS SOAP version (SAML)
- STS WS-Trust version (SAML)
- STS Server Certificate (SAML)

This section describes the following topics:

- "Obtaining the URI of the Secure Token Service" on page 2-28
- "Configuring STS URI for WS-SecureConversation: Standalone Client" on page 2-28
- "Configuring STS URI for SAML: Standalone Client" on page 2-29
- "Configuring STS URI Using WLST: Client On Server Side" on page 2-29
- "Configuring STS URI Using Console: Client On Server Side" on page 2-30
- "Configuring STS Security Policy: Standalone Client" on page 2-31
- "Configuring STS Security Policy Using WLST: Client On Server Side" on page 2-31
- "Configuring STS Security Policy: Using the Console" on page 2-32
- "Configuring the STS SOAP and WS-Trust Version: Standalone Client" on page 2-32
- "Configuring the SAML STS Server Certificate: Standalone Client" on page 2-33


#### 2.8.2.1 Obtaining the URI of the Secure Token Service

There are three sources from which the WS-Trust client can obtain the URI of the secure token service (STS). The order of precedence is as follows:

- The URI for the STS, as contained in the `sp:Issuer/wsa:Address` element of the token assertion in the Web service's security policy.
- A configured STS URI.
- The co-located STS URI. This is the default if there is no other source (WS-SecureConversation only).

---

**Note:** The URI for the STS, as contained in the *sp:IssuedToken/sp:Issuer/wsa:Address* element of the token assertion in the Web service's security policy is supported on the STS URI only for getting the SAML token, and is not supported for getting the Secure Conversation token in this release.

For example, the following assertion for STS URI is **not** supported for obtaining the Secure Conversation token (SCT):

```
<sp:IssuedToken
IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/20
0702/IncludeToken/AlwaysToRecipient">
<sp:Issuer>
<a:Address>http://example.com/STS</a:Address>
</sp:Issuer>
. . .
</sp:IssuedToken>
```

---

#### 2.8.2.2 Configuring STS URI for WS-SecureConversation: Standalone Client

For WS-SecureConversation, if the STS is co-located with the service there is no need to configure the STS URI. However, when the STS and the service do not share the same port, for example the service uses an HTTP port and the STS uses an HTTPs port, you need to configure the STS URI.

The following code example demonstrates setting the STS URI on a client stub under JAX-RPC. The example assumes that the location of the STS URI is already known to the client.

```
String wsdl = "http://myserver/wsscsecuredservice?wsdl";
WsscSecuredService service = new WsscSecuredService_Impl(wsdl);
WsscSecured port = service.getWsscSecuredSoapPort();
Stub stub = (Stub) port;
String sts = "https://myserver/wsscsecuredservice";
stub._setProperty(weblogic.wsee.jaxrpc.WLStub.WST_STS_ENDPOINT_ON_WSSC, sts);
```

The following code example demonstrates setting the STS URI on a client stub under JAX-WS.

```
String wsdl = "http://myserver/wsscsecuredservice?wsdl";
WsscSecuredService service = new WsscSecuredService_Impl(wsdl);
String sts = "https://myserver/wsscsecuredservice";
WsscSecured port = service.getWsscSecuredSoapPort();
BindingProvider provider = (BindingProvider) port;
Map context = provider.getRequestContext();
context.put(weblogic.wsee.jaxrpc.WLStub.WST_STS_ENDPOINT_ON_WSSC, sts)
```

### 2.8.2.3 Configuring STS URI for SAML: Standalone Client

When the STS is used for retrieving the SAML token, the STS is not co-located with the service and there is no default STS URI. You must configure the STS URI in this case.

The following code example demonstrates setting the STS URI for SAML on a client stub under JAX-RPC. The example assumes that the location of the STS URI is already known to the client.

```
String wsdl = "http://myserver/wssecuredservice?wsdl";
WssecuredService service = new  WsSecuredService_Impl(wsdl);
WsSecured port = service.getWsSecuredSoapPort();
Stub stub = (Stub) port;
String sts = "https://stsserver/standaloneSTS/saml/STS";
stub._setProperty(weblogic.wsee.jaxrpc.WLStub.WST_STS_ENDPOINT_ON_SAML, sts);
```

The following code example demonstrates setting the STS URI for SAML on a client stub under JAX-WS.

```
String wsdl = "http://myserver/wsssecuredservice?wsdl";
WsSecuredService service = new WsSecuredService_Impl(wsdl);
String sts = "https://stsserver/standaloneSTS/saml/STS";
WsscSecured port = service.getWsSecuredSoapPort();
BindingProvider provider = (BindingProvider) port;
Map context = provider.getRequestContext();
context.put(weblogic.wsee.jaxrpc.WLStub.WST_STS_ENDPOINT_ON_SAML, sts)
```

### 2.8.2.4 Configuring STS URI Using WLST: Client On Server Side

Example 2–6 demonstrates using the WebLogic Scripting Tool (WLST) to create a credential provider for the WS-Trust client and then configuring the STS URI, as indicated by bold text.

The provider class name can be one of the following:

- `weblogic.wsee.security.wssc.v200502.sct.ClientSCCredentialProvider`
- `weblogic.wsee.security.wssc.v13.sct.ClientSCCredentialProvider`

- `weblogic.wsee.security.saml.SAMLTrustCredentialProvider`

***Example 2–6 Configuring STS URI Using WLST***

```
userName = sys.argv[1]
passWord = sys.argv[2]
host = sys.argv[3]+":"+sys.argv[4]
sslhost = sys.argv[3]+":"+sys.argv[5]
url="t3://"+ host connect(userName, passWord, url)
edit()
startEdit()
defaultWss = cmo.lookupWebserviceSecurity('default_wss')
#Create credential provider for SCT Trust Client
wtm = defaultWss.createWebserviceCredentialProvider('trust_client_sct_cp')
wtm.setClassName('weblogic.wsee.security.wssc.v13.sct.ClientSCCredentialProvider')

wtm.setTokenType('sct_trust')
cpm = wtm.createConfigurationProperty('StsUri')
cpm.setValue("https://" + sslhost + "/standaloneSTS/wssc13/STS")
save()
activate(block="true")
disconnect()
exit()
```

### 2.8.2.5 Configuring STS URI Using Console: Client On Server Side

Configuring the STS URI through the WebLogic Server Administration Console allows the decision about which URI to use to be made at runtime, and not during the Web service development cycle.

Follow these steps to configure the STS URI through the Console:

1. Create a Web services security configuration, as described in the *Oracle WebLogic Server Administration Console Help*. This creates an empty configuration.
2. Edit the Web services security configuration to create a credential provider, as described in the *Oracle WebLogic Server Administration Console Help*:
   - On the Create Credential Provider tab, enter the following:
     - A provider name, which is your name for this MBean instance.
     - The provider class name, which can be

       `weblogic.wsee.security.wssc.v200502.sct.ClientSCCredentialProvider`
       or

       `weblogic.wsee.security.wssc.v13.sct.ClientSCCredentialProvider`
       or

       `weblogic.wsee.security.saml.SAMLTrustCredentialProvider`

     - The token type, which is a short name to identify the token. For example, sct or saml.
3. Select Next.
4. Enter the name/value pairs for the STS URI.
5. Select Finish.
6. On the Security Configuration General tab, set the value of the Default Credential Provider STS URI.

The Default Credential Provider STS URL is the default STS endpoint URL for all WS-Trust enabled credential providers of this Web service security configuration.

#### 2.8.2.6 Configuring STS Security Policy: Standalone Client

The following code example demonstrates setting the STS security policy on a client stub, under JAX-RPC, as indicated in bold.

```
import weblogic.wsee.message.WlMessageContext;
. . .
String wsdl = "http://myserver/samlsecuredservice?wsdl";
SamlSecuredService service = new SamlSecuredService_Impl(wsdl);
SamlSecured port = service.getSamlSecuredSoapPort();
Stub stub = (Stub) port;
InputStream policy = loadPolicy();
stub._setProperty(WlMessageContext.WST_BOOT_STRAP_POLICY, policy);
```

The following code example demonstrates setting the STS security policy on a client stub, under JAX-WS, as indicated in bold.

```
import weblogic.wsee.message.WlMessageContext;
. . .
String wsdl = "http://myserver/wsssecuredservice?wsdl";
WsSecuredService service = new WsSecuredService_Impl(wsdl);
WsscSecured port = service.getWsSecuredSoapPort();
BindingProvider provider = (BindingProvider) port;
Map context = provider.getRequestContext();
InputStream policy = loadPolicy();
context._setProperty(WlMessageContext.WST_BOOT_STRAP_POLICY, policy);
```

#### 2.8.2.7 Configuring STS Security Policy Using WLST: Client On Server Side

Example 2–7 demonstrates using WLST to create a credential provider for the default Web services security configuration, and then configuring the STS security policy, as indicated by bold text. The value for the `StsPolicy` property must be either a policy included in WebLogic Server (see Section 2.16, "Using WS-SecurityPolicy 1.2 Policy Files") or a custom policy file in a J2EE library (see Section 2.7, "Creating and Using a Custom Policy File").

***Example 2–7 Configuring STS Security Policy Using WLST***

```
userName = sys.argv[1]
passWord = sys.argv[2]
host = sys.argv[3]+":"+sys.argv[4]
sslhost = sys.argv[3]+":"+sys.argv[5]
samlstsurl = sys.argv[6]
url="t3://"+ host
print "Connect to the running adminSever"
connect(userName, passWord, url)
edit()
startEdit()
defaultWss = cmo.lookupWebserviceSecurity('default_wss')

#Create credential provider for SAML Trust Client

wtm = defaultWss.createWebserviceCredentialProvider('trust_client_saml_cp')
wtm.setClassName('weblogic.wsee.security.saml.SAMLTrustCredentialProvider')
wtm.setTokenType('saml_trust')
cpm = wtm.createConfigurationProperty('StsUri')
cpm.setValue(samlstsurl)
cpm = wtm.createConfigurationProperty('StsPolicy')
```

```
cpm.setValue("Wssp1.2-2007-Https-UsernameToken-Plain")
save()
activate(block="true")
disconnect()
exit()
```

#### 2.8.2.8 Configuring STS Security Policy: Using the Console

Perform the following steps to configure the STS security policy using the console:

1. Create a Web services security configuration, as described in the *Oracle WebLogic Server Administration Console Help*. This creates an empty configuration.
2. Edit the Web services security configuration to create a credential provider, as described in the *Oracle WebLogic Server Administration Console Help*:
   - On the Create Credential Provider tab, enter the following:
     - A provider name, which is your name for this MBean instance.
     - The provider class name, which can be

       `weblogic.wsee.security.wssc.v200502.sct.ClientSCCredentialProvider`
       or

       `weblogic.wsee.security.wssc.v13.sct.ClientSCCredentialProvider`
       or

       `weblogic.wsee.security.saml.SAMLTrustCredentialProvider`

     - The token type, which is a short name to identify the token. For example, sct or saml.
3. Select Next.
4. Enter the name/value pairs for the STS policy.
5. Select Finish.

#### 2.8.2.9 Configuring the STS SOAP and WS-Trust Version: Standalone Client

For a SAML STS, you need to configure the WS-Trust version only if it is not the default (WS-Trust 1.3). The supported values for `WSEESecurityConstants.TRUST_VERSION` are as follows:

- `http://docs.oasis-open.org/ws-sx/ws-trust/200512` (WS-Trust 1.3)
- `http://schemas.xmlsoap.org/ws/2005/02/trust`

You also need to configure the SOAP version if it is different from the SOAP version of the target Web service for which you generated the standalone client. (See Interface SOAPConstants (`http://java.sun.com/javaee/5/docs/api/javax/xml/soap/SOAPConstants.html`) for the definitions of the constants.) The supported values for `WSEESecurityConstants.TRUST_SOAP_VERSION` are as follows:

- `javax.xml.soap.SOAPConstants.URI_NS_SOAP_1_1_ENVELOPE` (as per `http://schemas.xmlsoap.org/soap/envelope/`)
- `javax.xml.soap.SOAPConstants.URI_NS_SOAP_1_2_ENVELOPE` ( as per `http://www.w3.org/2003/05/soap-envelope`)

Example 2–8 shows an example of setting the WS-Trust and SOAP versions.

***Example 2–8 Setting the WS-Trust and SOAP Versions***

```
// set WS-Trust version
stub._setProperty(WSEESecurityConstants.TRUST_VERSION,
"http://docs.oasis-open.org/ws-sx/ws-trust/200512");
// set SOAP version
stub._setProperty(WSEESecurityConstants.TRUST_SOAP_VERSION, SOAPConstants.URI_NS_
SOAP_1_1_ENVELOPE);
```

#### 2.8.2.10 Configuring the SAML STS Server Certificate: Standalone Client

For a SAML STS, you need to configure the STS server X.509 certificate if you use a message-level policy to protect the request and response between the STS server and the WS-Trust client. (If you use a transport-level policy, you do not need to configure the STS server certificate.)

Example 2–9 shows an example of setting the STS server certificate under JAX-RPC, assuming the location of the STS sever certificate is known.

***Example 2–9 Setting STS Server Certificate under JAX-RPC***

```
// import
import weblogic.wsee.security.util.CertUtils;
import java.security.cert.X509Certificate;
import weblogic.wsee.jaxrpc.WLStub;
. . .

// get X509 Certificate
String stsCertLocation = "../../cert/WssIP.cer";
X509Certificate stsCert = CertUtils.getCertificate(stsCertLocation);
// set STS Server Cert
stub._setProperty(WLStub.STS_ENCRYPT_CERT,stsCert);
```

Example 2–10 shows the same example of setting the STS server certificate under JAX-WS. The JAX-WS specific code in the example is shown in bold.

***Example 2–10 Setting STS Server Certificate under JAX-WS***

```
// import
import weblogic.wsee.security.util.CertUtils;
import java.security.cert.X509Certificate;
import weblogic.wsee.jaxrpc.WLStub;
. . .

// get X509 Certificate
String stsCertLocation = "../../cert/WssIP.cer";
X509Certificate stsCert = CertUtils.getCertificate(stsCertLocation);
// set STS Server Cert
context.put(WLStub.STS_ENCRYPT_CERT,stsCert);
```

### 2.8.3 Sample WS-Trust Client for SAML 2.0 Bearer Token over HTTPS

You can configure a client application to use WS-Trust to retrieve the SAML 2.0 bearer token from STS, and then use the SAML token for authentication on the bootstrap message on secure conversation.

> **Note:** When using the bearer confirmation method for the SAML assertion, use SAML 2.0 in this release. SAML 1.1 for bearer confirmation method is not supported.

In this scenario, transport-level message protection is used for WS-Trust message exchange between a client and the SAML STS, as well as the bootstrap message on secure conversation. A public key and private key are not required for this standalone client.

The policy for the service side is similar to the predefined WS-Policy file `Wssp1.2-2007-Wssc1.3-Bootstrap-Https-UNT.xml`, except the following `<sp:SupportingTokens>` is used in the policy instead:

```
<sp:SupportingTokens>
   <wsp:Policy>
      <sp:SamlToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/Include
Token/AlwaysToRecipient">
         <wsp:Policy>
              <sp:WssSamlV20Token11/>
         </wsp:Policy>
      </sp:SamlToken>
   </wsp:Policy>
</sp:SupportingTokens>
```

The policy that is used to protect the WS-Trust message between the WS-Trust client and the remote STS server is a copy of the packaged security policy file `Wssp1.2-2007-Https-UsernameToken-Plain.xml`, which uses username token for authentication in transport-level message protection.

> **Note:** When using transport-level security policy to protect the bootstrap message of secure conversation, the WS-Trust messages exchanged between the WS-Trust client and the remote STS must also use transport-level security policy to protect the WS-Trust messages.

When invoking the Web service from the client, it is similar to a standard client application that invokes a message-secured Web service, as described in "Using a Client-Side Security Policy File" on page 2-54. The major difference is that you need to configure two STS endpoints: one for the retrieved SAML token, and another for getting the Security Context Token (SCT) for Secure Conversation.

Example 2–11 shows a simple example of a client application invoking a Web service under JAX-WS that is retrieving a SAML token via WS-Trust. It is associated with a security policy that enables secure conversations by using HTTPS transport-level protection. The sections in bold are relevant to security contexts and are described after the example:

***Example 2–11 Client Application Using WS-Trust and WS-SecureConversation with HTTPS***

```
package examples.webservices.samlwsschttps.client;

import weblogic.security.SSL.TrustManager;
import weblogic.wsee.message.WlMessageContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.saml.SAMLTrustCredentialProvider;
```

```
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.wsee.jaxrpc.WLStub;
import weblogic.wsee.security.util.CertUtils;
import com.sun.xml.ws.developer.MemberSubmissionAddressingFeature;
import java.security.cert.X509Certificate;
import javax.xml.ws.*;
import javax.xml.namespace.*;
import javax.net.ssl.HttpsURLConnection;
import java.net.URL;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class TravelAgencyClient {

  public static final String STS_POLICY = "StsHttpsUntPolicy.xml";
  static {
    HttpsURLConnection.setDefaultHostnameVerifier(new MyHostnameVerifier());
    try {
      String defaultTrustStore = new
File(TravelAgencyClient.class.getResource("/cacerts").getFile()).getCanonicalPath(
);
      System.out.println("Default trustStore:\t" + defaultTrustStore);
      System.setProperty("javax.net.ssl.trustStore", defaultTrustStore);
    } catch (IOException e) {
      System.out.printf("can't find default trusted keystore");
    }
   }

  public static void main(String[] args) throws Exception {
      TravelAgencyClient client  = new TravelAgencyClient();
      String wsscStsURL = System.getProperty("wsscStsURL");
   System.out.println("WSSC StS URL \t" + wsscStsURL);
      String samlStsURL = System.getProperty("samlStsURL");
      System.out.println("StS URL \t" + samlStsURL);
      String hotelWsdlURL = System.getProperty("hotelWsdlURL");
      System.out.println("Hotel Service WSDL URL \t" + hotelWsdlURL);

      String hotelResult = client.callWsscHotelService("Travel Agency client to
Hotel Service", wsscStsURL,hotelWsdlURL, samlStsURL);
      System.out.println("Hotel Service return value: -->"+hotelResult);
  }

  public String callWsscHotelService(String hello,
                    String wsscStsURL,
                    String hotelWsdlURL,
                    String samlStsURL) throws Exception{

    HotelService service = new HotelService(new URL(hotelWsdlURL),
            new QName("http://wsinterop.org/samples", "HotelService"));

    IHotelService port = service.getIHotelServicePort(new
MemberSubmissionAddressingFeature());

    BindingProvider provider = (BindingProvider)port;
```

```
    this.configurePort(provider, wsscStsURL, samlStsURL);

    try {
          // for securie conversation, it can call twice
      String s1 = port.getName(hello);
      String s2 = port.getName(hello + " --- " + s1) ;
      WSSCClientUtil.terminateWssc((BindingProvider)port);
      return s2;
    } catch (Exception ex) {
        ex.printStackTrace();
      throw new RuntimeException("fail to call the remote hotel service!", ex);
    }
  }

 private void configurePort(BindingProvider provider, String wsscStsURL, String
samlStsURL) throws Exception {

    Map context = provider.getRequestContext();
    InputStream policy = getPolicy(STS_POLICY);
    context.put(WlMessageContext.WST_BOOT_STRAP_POLICY, policy);
    if (null != wsscStsURL) {
        context.put(WLStub.WST_STS_ENDPOINT_ON_WSSC, wsscStsURL);
    }
    context.put(WLStub.WST_STS_ENDPOINT_ON_SAML, samlStsURL);
    context.put(WSSecurityContext.TRUST_MANAGER,
        new TrustManager() {
          public boolean certificateCallback(X509Certificate[] chain,
                                             int validateErr) {
            // need to validate if the server cert can be trusted
            return true;
          }
        });
    List credProviders = buildCredentialProviderList();
    context.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);

    context.put(com.sun.xml.ws.developer.JAXWSProperties.HOSTNAME_VERIFIER, new
MyHostnameVerifier());
  }

  private static List buildCredentialProviderList() throws Exception  {
    List credProviders = new ArrayList();
    credProviders.add(new SAMLTrustCredentialProvider());
    credProviders.add(getClientUNTCredentialProvider());
    return credProviders;
  }
  private static CredentialProvider getClientUNTCredentialProvider() throws
Exception {
    String username = System.getProperty("target.username", "Alice");
    String password = System.getProperty("target.password", "Password1");
    return new ClientUNTCredentialProvider(username.getBytes(),
        password.getBytes());
  }
  private  InputStream getPolicy(String policyName) {
    String resName = '/' + this.getClass().getPackage().getName().replace('.',
'/') + '/' + policyName;
    InputStream  stsPolicy = this.getClass().getResourceAsStream(resName);
    if(stsPolicy == null)    {
        throw new RuntimeException("STS policy is not correctly set!");
    }
    return stsPolicy;
```

```
  }
  public static class MyHostnameVerifier implements javax.net.ssl.HostnameVerifier
{
      public boolean verify(String hostname, javax.net.ssl.SSLSession session) {
        return(true);
     }
    }
}
```

Note the following points in this example:

- Configure the policy for message protection between the remote STS and WS-Trust client:

  ```
  context.put(WlMessageContext.WST_BOOT_STRAP_POLICY,
  policy);
  ```

- The bootstrap is protected by transport-level policy, and you need to set the STS endpoint address for secure conversation:

  ```
  context.put(WLStub.WST_STS_ENDPOINT_ON_WSSC, wsscStsURL);
  ```

- Set the STS endpoint address for SAML STS:

  ```
  context.put(WLStub.WST_STS_ENDPOINT_ON_SAML, samlStsURL);
  ```

- For transport-level protection, you need to configure the hostname verifier:

  ```
  context.put(com.sun.xml.ws.developer.JAXWSProperties.HOSTNAME
  _VERIFIER, new MyHostnameVerifier());
  ```

- Set the SAML Trust Credential Provider to handle the remote SAML token retrieval:

  ```
  credProviders.add(new SAMLTrustCredentialProvider());
  ```

- Set the client user name token provider to use the client's user name and password to exchange the SAML token via the WS-Trust call:

  ```
  credProviders.add(getClientUNTCredentialProvider());
  ```

## 2.8.4 Sample WS-Trust Client for SAML 2.0 Bearer Token with WSS 1.1 Message Protections

Similar to Example 2–11, you can configure a client application to use WS-Trust to retrieve the SAML 2.0 bearer token from STS, and then use the SAML token for authentication on the bootstrap message on secure conversation. However, instead of using HTTPS transport-level message protection, it uses WS-Security 1.1 message-level protection, and HTTPS configuration is not required.

In this scenario, the STS server's X.509 certificate is used to protect the WS-Trust message exchange between the client and the SAML STS, and the server's X.509 certificate is used to protect the bootstrap message on secure conversation. A public key and private key are not required for this standalone client.

The policy for the service side is similar to the packaged WS-Policy file `Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.1.xml`, except that it uses a SAML 2.0 token for authentication in the bootstrap message instead of the client's X.509 certificate. That is, it uses a `<sp:SignedSupportingTokens>` assertion with a SAML token inside the policy instead of using a `<sp:SignedEndorsingSupportingTokens>` assertion.

The entire secure conversation policy is as follows:

```
<?xml version="1.0"?>
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-util
ity-1.0.xsd">
  <sp:SymmetricBinding>
    <wsp:Policy>
      <sp:ProtectionToken>
        <wsp:Policy>
          <sp:SecureConversationToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/Include
Token/AlwaysToRecipient">
            <wsp:Policy>
              <sp:RequireDerivedKeys/>
              <sp:BootstrapPolicy>
                <wsp:Policy>
                  <sp:SignedParts>
                    <sp:Body/>
                    <sp:Header
Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
                    <sp:Header Namespace="http://www.w3.org/2005/08/addressing"/>
                  </sp:SignedParts>
                  <sp:EncryptedParts>
                    <sp:Body/>
                  </sp:EncryptedParts>
                  <sp:SymmetricBinding>
                    <wsp:Policy>
                      <sp:ProtectionToken>
                        <wsp:Policy>
                          <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/Include
Token/Never">
                            <wsp:Policy>
                              <sp:RequireDerivedKeys/>
                              <sp:RequireThumbprintReference/>
                              <sp:WssX509V3Token11/>
                            </wsp:Policy>
                          </sp:X509Token>
                        </wsp:Policy>
                      </sp:ProtectionToken>
                      <sp:AlgorithmSuite>
                        <wsp:Policy>
                          <sp:Basic256/>
                        </wsp:Policy>
                      </sp:AlgorithmSuite>
                      <sp:Layout>
                        <wsp:Policy>
                          <sp:Lax/>
                        </wsp:Policy>
                      </sp:Layout>
                      <sp:IncludeTimestamp/>
                      <sp:OnlySignEntireHeadersAndBody/>
                    </wsp:Policy>
                  </sp:SymmetricBinding>
                  <sp:SignedSupportingTokens>
                    <wsp:Policy>
                      <sp:SamlToken
  sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/
IncludeToken/AlwaysToRecipient">
                        <wsp:Policy>
```

```
                        <sp:WssSamlV20Token11/>
                      </wsp:Policy>
                    </sp:SamlToken>
                  </wsp:Policy>
                </sp:SignedSupportingTokens>
                <sp:Wss11>
                  <wsp:Policy>
                    <sp:MustSupportRefKeyIdentifier/>
                    <sp:MustSupportRefIssuerSerial/>
                    <sp:MustSupportRefThumbprint/>
                    <sp:MustSupportRefEncryptedKey/>
                    <sp:RequireSignatureConfirmation/>
                  </wsp:Policy>
                </sp:Wss11>
              </wsp:Policy>
            </sp:BootstrapPolicy>
          </wsp:Policy>
        </sp:SecureConversationToken>
      </wsp:Policy>
    </sp:ProtectionToken>
    <sp:AlgorithmSuite>
      <wsp:Policy>
        <sp:Basic256/>
      </wsp:Policy>
    </sp:AlgorithmSuite>
    <sp:Layout>
      <wsp:Policy>
        <sp:Lax/>
      </wsp:Policy>
    </sp:Layout>
    <sp:IncludeTimestamp/>
    <sp:ProtectTokens/>
    <sp:OnlySignEntireHeadersAndBody/>
  </wsp:Policy>
</sp:SymmetricBinding>
<sp:Wss11>
  <wsp:Policy>
    <sp:MustSupportRefKeyIdentifier/>
    <sp:MustSupportRefIssuerSerial/>
    <sp:MustSupportRefThumbprint/>
    <sp:MustSupportRefEncryptedKey/>
    <sp:RequireSignatureConfirmation/>
  </wsp:Policy>
</sp:Wss11>
<sp:Trust13>
  <wsp:Policy>
    <sp:MustSupportIssuedTokens/>
    <sp:RequireClientEntropy/>
    <sp:RequireServerEntropy/>
  </wsp:Policy>
</sp:Trust13>
</wsp:Policy>
```

The policy that is used to protect the WS-Trust message between the WS-Trust client and the remote STS server is a copy of packaged security policy `Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey.xml`, which uses the username token for authentication and WS-Security 1.1 message-level security.

The entire security policy is as follows:

```
<?xml version="1.0"?>
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <sp:SymmetricBinding>
    <wsp:Policy>
      <sp:ProtectionToken>
        <wsp:Policy>
          <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/Include
Token/Never">
            <wsp:Policy>
              <sp:RequireThumbprintReference/>
              <sp:WssX509V3Token11/>
            </wsp:Policy>
          </sp:X509Token>
        </wsp:Policy>
      </sp:ProtectionToken>
      <sp:AlgorithmSuite>
        <wsp:Policy>
          <sp:Basic256/>
        </wsp:Policy>
      </sp:AlgorithmSuite>
      <sp:Layout>
        <wsp:Policy>
          <sp:Lax/>
        </wsp:Policy>
      </sp:Layout>
      <sp:IncludeTimestamp/>
      <sp:OnlySignEntireHeadersAndBody/>
    </wsp:Policy>
  </sp:SymmetricBinding>
  <sp:SignedEncryptedSupportingTokens>
    <wsp:Policy>
      <sp:UsernameToken
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/Include
Token/AlwaysToRecipient">
        <wsp:Policy>
          <sp:WssUsernameToken10/>
        </wsp:Policy>
      </sp:UsernameToken>
    </wsp:Policy>
  </sp:SignedEncryptedSupportingTokens>
  <sp:Wss11>
    <wsp:Policy>
      <sp:MustSupportRefKeyIdentifier/>
      <sp:MustSupportRefIssuerSerial/>
      <sp:MustSupportRefThumbprint/>
      <sp:MustSupportRefEncryptedKey/>
      <sp:RequireSignatureConfirmation/>
    </wsp:Policy>
  </sp:Wss11>
  <sp:SignedParts>
    <sp:Header Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
    <sp:Header Namespace="http://www.w3.org/2005/08/addressing"/>
    <sp:Body/>
  </sp:SignedParts>
  <sp:EncryptedParts>
    <sp:Body/>
  </sp:EncryptedParts>
</wsp:Policy>
```

**Note:** When using message-level security policy to protect the bootstrap message of secure conversation, the WS-Trust messages exchanged between the WS-Trust client and the remote STS must also use message-level security policy to protect the WS-Trust messages. Mixing transport- and message-level security policy is not supported.

When invoking a Web service from the WS-Trust client, the configurations are mostly similar to the previous example. The major differences are:

- You need to configure two encryption certificates: one is the certificate of the STS for SAML token retrieval, and the other is the certificate for the server.
- Configuring the service STS endpoint address for secure conversation is not required. When the bootstrap message is not protected by transport-level security, by default the STS endpoint address is the same as the service endpoint address for security conversation.
- The SSL configuration is not required.

Example 2–12 shows a simple example of a client application invoking a Web service under JAX-WS that is retrieving a SAML token via WS-Trust. It is associated with a security policy that enables secure conversations by using WS-Security 1.1 message-level security. The sections in bold are relevant to security contexts and are described after the example:

***Example 2–12 Client Application Using WS-Trust and WS-SecureConversation without HTTPS***

```
package examples.webservices.samlwssc.client;

import weblogic.security.SSL.TrustManager;
import weblogic.wsee.message.WlMessageContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.saml.SAMLTrustCredentialProvider;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.wsee.jaxrpc.WLStub;
import weblogic.wsee.security.util.CertUtils;
import weblogic.wsee.security.wssc.utils.WSSCClientUtil;
import com.sun.xml.ws.developer.MemberSubmissionAddressingFeature;

. . .

public class TravelAgency1Client {

    public static final String STS_POLICY = "StsWss11UntPolicy.xml";

    public static void main(String[] args) throws Exception {
        TravelAgencyClient client = new TravelAgencyClient();
        String stsURL = System.getProperty("stsURL");
        System.out.println("StS URL \t" + stsURL);

        String hotelWsdlURL = System.getProperty("hotelWsdlURL");
        System.out.println("Hotel Service WSDL URL \t" + hotelWsdlURL);
        String hotelResult = client.callWsscHotelService("Travel Agency client to
Hotel Service", stsURL, hotelWsdlURL);
        System.out.println("Hotel Service return value: -->" + hotelResult);
    }
```

```
    public String callWsscHotelService(String hello,
                                       String stsurl,
                                       String hotelWsdlURL) throws Exception {

        HotelService service = new HotelService(new URL(hotelWsdlURL),
                new QName("http://wsinterop.org/samples", "HotelService"));

        IHotelService port = service.getIHotelServicePort(new
MemberSubmissionAddressingFeature());

        BindingProvider provider = (BindingProvider) port;
        this.configurePort(provider, stsurl);

        try {
            // for securie conversation, it can call twice
            String s1 = port.getName(hello);
            String s2 = port.getName(hello + " --- " + s1);
            WSSCClientUtil.terminateWssc((BindingProvider)port);
            return s2;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException("fail to call the remote hotel service!",
ex);
        }
    }

    private void configurePort(BindingProvider provider, String stsurl) throws
Exception {

        Map context = provider.getRequestContext();
        InputStream policy = getPolicy(STS_POLICY);
        context.put(WlMessageContext.WST_BOOT_STRAP_POLICY, policy);
        context.put(WLStub.WST_STS_ENDPOINT_ON_SAML, stsurl);
        context.put(WLStub.STS_ENCRYPT_CERT, getStsCert());
        context.put(WLStub.SERVER_ENCRYPT_CERT, getServerCert());
        List credProviders = buildCredentialProviderList();
        context.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);
        context.put(WLStub.POLICY_COMPATIBILITY_PREFERENCE, WLStub.POLICY_
COMPATIBILITY_MSFT);
    }

    private static List buildCredentialProviderList() throws Exception {
        List credProviders = new ArrayList();
        credProviders.add(new SAMLTrustCredentialProvider());
        credProviders.add(getClientUNTCredentialProvider());
        return credProviders;
    }

. . .

    private static X509Certificate getServerCert() throws Exception {
        String defaultServerCert = new File(
TravelAgency1Client.class.getResource("/Bob.cer").getFile()).getCanonicalPath();
        String certName = System.getProperty("target.serverCert",
 defaultServerCert);
        X509Certificate cert = CertUtils.getCertificate(certName);
        return cert;
    }
}
```

Note the following points in this example:

- Configure the STS Server certificate for message protection between the remote STS and WS-Trust client:

  ```
  context.put(WLStub.STS_ENCRYPT_CERT, getStsCert());
  ```

- Configure the STS Server certificate for message protection of the bootstrap message of secure conversation:

  ```
  context.put(WLStub.SERVER_ENCRYPT_CERT, getServerCert());
  ```

- Optionally, if the service is a Microsoft .NET WCF service, then set the `WLStub.POLICY_COMPATIBILITY_PREFERENCE` flag to `WLStub.POLICY_COMPATIBILITY_MSFT` for interoperability:

  ```
  context.put(WLStub.POLICY_COMPATIBILITY_PREFERENCE,
  WLStub.POLICY_COMPATIBILITY_MSFT);
  ```

## 2.9 Configuring and Using Security Contexts and Derived Keys

Oracle provides the following predefined WS-SecurityPolicy files to configure security contexts and derived keys:

- WS-SecureConversation 1.2 (2005/2) specification:
  - Wssp1.2-Wssc200502-Bootstrap-Https.xml
  - Wssp1.2-Wssc200502-Bootstrap-Wss1.0.xml
  - Wssp1.2-Wssc200502-Bootstrap-Wss1.1.xml
- WS-SecureConversation 1.3 versions of the WS-SecureConversation 1.2 (2005/2) policy files:
  - Wssp1.2-Wssc1.3-Bootstrap-Https.xml
  - Wssp1.2-Wssc1.3-Bootstrap-Wss1.0.xml
  - Wssp1.2-Wssc1.3-Bootstrap-Wss1.1.xml
- Additional WS-SecureConversation 1.3 policy files:
  - Wssp1.2-Wssc1.3-Bootstrap-Https-BasicAuth.xml
  - Wssp1.2-Wssc1.3-Bootstrap-Https-ClientCertReq.xml

It is recommended that you use the predefined files if you want to configure security contexts, because these security policy files provide most of the required functionality and typical default values. See Section 2.16.5, "WS-SecureConversation Policies" for more information about these files.

> **Note:** If you are deploying a Web service that uses shared security contexts to a cluster, then you are required to also configure cross-cluster session state replication. For details, see "Failover and Replication in a Cluster" in *Using Clusters for Oracle WebLogic Server*.

Code or configure your application to use the policy through policy annotations, policy attached to the application's WSDL, or runtime policy configuration.

### 2.9.1 Specification Backward Compatibility

WebLogic Web services implement the Web Services Trust (WS-Trust 1.3) and Web Services Secure Conversation (WS-SecureConversation 1.3) specifications. Take note of the following differences from the WS-SecureConversation version of 02/2005:

- The Web Services Secure Conversation (WS-SecureConversation 1.3) specification requires a token service to return wst:RequestedSecurityToken to the initiating party in response to a wst:RequestSecurityToken. One or more wst:RequestSecurityTokenResponse elements are contained within a single wst:RequestSecurityTokenResponseCollection.

  This differs from the previous version of the specification, in which wst:RequestSecurityTokenResponse was returned by the token service.

  The token service can return wst:RequestSecurityTokenResponse if the service policy specifies the SC10SecurityContextToken, as described in the next bullet item.

- The WS-SecurityPolicy 1.2 Errata document describes the following change to SecureConversationToken Assertion:

  <sp:SC10SecurityContextToken />

  changes to

  <sp:SC13SecurityContextToken />

  sp:SC10SecurityContextToken continues to be supported only when used with the WS-SecureConversation version of 02/2005.

### 2.9.2 WS-SecureConversation and Clusters

WS-SecureConversation is pinned to a particular WebLogic Server instance in the cluster. If a SecureConversation request lands in the wrong server, it is automatically rerouted to the correct server. If the server instance hosting the WS-SecureConversation fails, the SecureConversation will not be available until the server instance is brought up again.

### 2.9.3 Updating a Client Application to Negotiate Security Contexts

A client application that negotiates security contexts when invoking a Web service is similar to a standard client application that invokes a message-secured Web service, as described in Section 2.15, "Using a Client-Side Security Policy File". The only real difference is that you can use the *weblogic.wsee.security.wssc.utils.WSSCClientUtil* API to explicitly cancel the secure context token.

You can configure the SCT expiration value by setting SCT lifetime property. The SCT expiration value is then used to time out the SCT. When the timeout is reached, the Web services runtime on the client side automatically renews the SCT. The Web services runtime automatically cancels the unused secure context token when the timeout is reached.

---

**Note:** WebLogic Server provides the WSSCCLientUtil API for your convenience only; the Web services runtime automatically cancels the secure context token when the configured timeout is reached. Use the API only if you want to have more control over when the token is cancelled.

---

Example 2–13 shows a simple example of a client application invoking a Web service under JAX-RPC that is associated with a predefined security policy file that enables secure conversations; the sections in bold that are relevant to security contexts are discussed after the example:

***Example 2–13 Client Application Using WS-SecureConversation***

```
package examples.webservices.wssc.client;
import weblogic.security.SSL.TrustManager;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.bst.StubPropertyBSTCredProv;
import weblogic.wsee.security.wssc.utils.WSSCClientUtil;
import weblogic.wsee.security.util.CertUtils;
import javax.xml.rpc.Stub;
import java.util.List;
import java.util.ArrayList;
import java.security.cert.X509Certificate;

/**
 * Copyright © 1996, 2008, Oracle and/or its affiliates.
 * All rights reserved.
 */
public class WSSecureConvClient {
  public static void main(String[] args) throws Throwable {

    String clientKeyStore = args[0];
    String clientKeyStorePass = args[1];
    String clientKeyAlias = args[2];
    String clientKeyPass = args[3];
    String serverCert = args[4];
    String wsdl = args[5];

    WSSecureConvService service = new WSSecureConvService_Impl(wsdl);
    WSSecureConvPortType port = service.getWSSecureConvServicePort();

    //create credential provider and set it to the Stub
    List credProviders = new ArrayList();

    //use x509 to secure wssc handshake
    credProviders.add(new ClientBSTCredentialProvider(clientKeyStore,
clientKeyStorePass, clientKeyAlias, clientKeyPass));

    Stub stub = (Stub)port;

    stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);
    stub._setProperty(StubPropertyBSTCredProv.SERVER_ENCRYPT_CERT,
CertUtils.getCertificate(serverCert));
    stub._setProperty(WlMessageContext.SCT_LIFETIME_PROPERTY, new Long( 2 * 60 *
60 * 1000L));
// set to 2 hrs (Default is 30 minutes.)

    stub._setProperty(WSSecurityContext.TRUST_MANAGER,
        new TrustManager(){
          public boolean certificateCallback(X509Certificate[] chain, int
validateErr){
            //need to validate if the server cert can be trusted
            return true;
          }
```

```
        }
    );

    System.out.println (port.sayHelloWithWSSC("Hello World, once"));
    System.out.println (port.sayHelloWithWSSC("Hello World, twice"));
    System.out.println (port.sayHelloWithWSSC("Hello World, thrice"));

    //cancel SecureContextToken after done with invocation
    WSSCClientUtil.terminateWssc(stub);
    System.out.println("WSSC terminated!");

  }
}
```

The points to notice in the preceding example are:

- Import the WebLogic API used to explicitly terminate the secure context token:

  ```
  import weblogic.wsee.security.wssc.utils.WSSCClientUtil;
  ```

- Set a property on the JAX-RPC stub that specifies that the client application must encrypt its request to WebLogic Server with the given WebLogic Server's public key:

  ```
  stub._setProperty(StubPropertyBSTCredProv.SERVER_ENCRYPT_CERT,
  CertUtils.getCertificate(serverCert));
  ```

- Set a property on the JAX-RPC stub that specifies the Security Context Token (SCT) timeout value:

  ```
  stub._setProperty(WlMessageContext.SCT_LIFETIME_PROPERTY, new Long( 2 * 60 * 60
  * 1000L));
  ```

  > **Note:** Setting the SCT lifetime value is optional. The default value is set to 30 minutes. Setting a shorter SCT lifetime value is more secure, but requires renewing the SCT more frequently. Setting a longer SCT lifetime requires renewing the SCT less frequently, and it stays in memory longer if not explicitly terminated.

- Use the `terminateWssc()` method of the `WSSClientUtil` class to terminate the secure context token:

  ```
  WSSCClientUtil.terminateWssc(stub);
  ```

## 2.10 Associating Policy Files at Runtime Using the Administration Console

The simple message-level configuration procedure, documented in Section 2.4, "Configuring Simple Message-Level Security: Main Steps", describes how to use the `@Policy` and `@Policies` JWS annotations in the JWS file that implements your Web service to specify one or more policy files that are associated with your service. This of course implies that you must already know, at the time you program your Web service, which policy files you want to associate with your Web service and its operations. This might not always be possible, which is why you can also associate policy files at *runtime*, after the Web service has been deployed, using the Administration Console.

You can use no `@Policy` or `@Policies` JWS annotations at all in your JWS file and associate policy files only at runtime using the Administration Console, or you can

specify some policy files using the annotations and then associate additional ones at runtime.

At runtime, the Administration Console allows you to associate as many policy files as you want with a Web service and its operations, even if the policy assertions in the files contradict each other or contradict the assertions in policy files associated with the JWS annotations. It is up to you to ensure that multiple associated policy files work together. If any contradictions do exist, WebLogic Server returns a runtime error when a client application invokes the Web service operation.

To use the Console to associate one or more WS-Policy files to a Web service, the WS-Policy XML files must be located in either the META-INF/policies or WEB-INF/policies directory of the EJB JAR file (for EJB implemented Web services) or WAR file (for Java class implemented Web services), respectively.

See "Associate a WS-Policy file with a Web Service" in the *Oracle WebLogic Server Administration Console Help* for detailed instructions on using the Administration Console to associate a policy file at runtime.

## 2.11 Using Security Assertion Markup Language (SAML) Tokens For Identity

The SAML Token Profile 1.1 (http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SAMLTokenProfile.pdf) is part of the core set of WS-Security standards, and specifies how SAML assertions can be used for Web services security. WebLogic Server supports SAML Token Profile 1.1, including support for SAML 2.0 and SAML 1.1 assertions. SAML Token Profile 1.1 is backwards compatible with SAML Token Profile 1.0.

---

**Note:** SAML Token Profile 1.1 is supported only through WS-SecurityPolicy.

Previous releases of WebLogic Server, released before the formulation of the WS-SecurityPolicy specification, used security policy files written under the WS-Policy specification, using a proprietary schema for security policy. These earlier security policy files support SAML Token Profile 1.0 and SAML 1.1 only.

---

In the simple Web services configuration procedure, described in Section 2.4, "Configuring Simple Message-Level Security: Main Steps", it is assumed that users use username tokens to authenticate themselves. Because WebLogic Server implements the SAML Token Profile 1.1 (http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SAMLTokenProfile.pdf) of the Web Services Security specification, users can also use SAML tokens in the SOAP messages to authenticate themselves when invoking a Web service operation, as described in this section.

Use of SAML tokens works server-to-server. This means that the client application is running inside of a WebLogic Server instance and then invokes a Web service running in another WebLogic Server instance using SAML for identity. Because the client application is itself a Web service, the Web services security runtime takes care of all the SAML processing.

In addition to this server-to-server usage, you can also use SAML tokens from a standalone client via WS-Trust, as described in Section 2.8, "Configuring the WS-Trust Client".

**Note:** It is assumed in this section that you understand the basics of SAML and how it relates to core security in WebLogic Server. For general information, see "Security Assertion Markup Language (SAML)" in *Understanding Security for Oracle WebLogic Server*.

It is also assumed in the following procedure that you have followed the steps in Section 2.4, "Configuring Simple Message-Level Security: Main Steps" and now want to enable the additional use case of using SAML tokens, rather than username tokens, for identity.

### 2.11.1 Using SAML Tokens for Identity: Main Steps

1. Make sure that the SAML providers you need are configured and add the appropriate partner entries. This step configures the core WebLogic Server security subsystem. For details, see the following sections in *Securing Oracle WebLogic Server*:
   - *Configuring a SAML Identity Assertion Provider*
   - *Configuring a SAML Credential Mapping Provider*

   **Note:** You will need to configure both SAML 1.1 and SAML 2.0 security providers if you want to enable both versions of SAML for use with the SAML Token Profile.

   When configuring SAML 2.0 partner entries, you must use the endpoint URL of the target Web service as the name of the partner for both WSSIdPPartner and WSSSPPartner entries. Specify the URL as HTTPS if SSL will be used.

2. If you will be using policies that involve signatures related to SAML assertions (for example, SAML Holder-of-Key policies) where a key referenced by the assertion is used to sign the message, or Sender-Vouches policies where the sender's key is used to sign the message, you need to configure keys and certificates for signing and verification.

   For the Holder-of-Key scenarios, the signature from the client certificate is to prove that the client has possession of the private key that the SAML token references. For the Sender Vouches scenarios, the signature from the client certificate is to guarantee that the message with the SAML token is generated by the sender.

   **Note:** These keys and certificates are not used to create or verify signatures on the assertions themselves. Creating and verifying signatures on assertions is done using keys and certificates configured on the SAML security providers.

   If you are using SAML Bearer policies, protection is provided by SSL and the PKI Credential Mapping provider is not needed.

   If you are using SAML tokens from a standalone client via WS-TRUST, the tokens are passed in via the Web service client stub, not via the PKI Credential Mapping provider.

   a. Configure a PKI Credential Mapping provider on the sending side, and populate it with the keys and certificates to be used for signing.

`setKeypairCredential` creates a keypair mapping between the `principalName`, `resourceid` and credential action and the keystore alias and the corresponding password.

```
pkiCM.setKeypairCredential(
type=<remote>, protocol=http,
remoteHost=hostname, remotePort=portnumber, path=/ContextPath/ServicePath,
username, Boolean('true'), None,
alias, passphrase)
```

The first (String) parameter is used to construct a Resource object that represents the endpoint of the target Web service. The `userName` parameter is the user on whose behalf the signed Web service message will be generated. The `alias` and `passphrase` parameters are the alias and passphrase used to retrieve the key/certificate from the keystore configured for the PKI Credential Mapping provider. The actual key and certificate should be loaded into the keystore before creating the `KeypairCredential`.

**b.** Add the same certificates to the Certificate Registry on the receiving side, so they can be validated by the Web service security runtime:

```
reg.registerCertificate(certalias, certfile)
```

## 2.11.2 Specifying the SAML Confirmation Method

The WS-SecurityPolicy implies, but does not explicitly specify, the confirmation method for SAML assertions. Consider the following general guidelines:

- For WSS1.0 Asymmetric Binding, if the SamlToken assertion is inside the `<sp:AsymmerticBinding>` assertion, then the Holder of Key confirmation method is used.

  For WSS1.1 Symmetric Binding, if the SamlToken assertion is inside the `<sp:EndorsingSupportingTokens>` assertion, then the Holder of Key confirmation method is used.

  See Table 2–8 for examples of predefined policies that use Holder of Key confirmation.

- For WSS1.0 Asymmetric Binding, if the SamlToken assertion is inside `<sp:SignedSupportingTokens>`, then the Sender Vouches confirmation method is used.

  For WSS1.1 Symmetric Binding, if the SamlToken assertion is inside the `<sp:SignedSupportingTokens>` assertion, and the `<sp:X509Token>` is used in the `<sp:EndorsingSupportingTokens>` assertion , then the Sender Vouches confirmation method is used.

  For Transport Binding, two-way SSL with client certification is required for the Sender Vouches confirmation method. Use transport-level security as described in Chapter 3, "Configuring Transport-Level Security" in this case.

  See Table 2–8 for examples of predefined policies that use Sender Vouches confirmation.

- For transport-level security, if the SamlToken assertion is inside `<sp:SupportingTokens>`, then the Bearer confirmation method is used. Use transport-level security as described in Chapter 3, "Configuring Transport-Level Security" in this case.

  For WSS1.1 Symmetric Binding, if the SamlToken assertion is inside the `<sp:SignedSupportingTokens>` assertion, and there is no

<sp:EndorsingSupportingTokens> assertion, then the Bearer confirmation method is used.

See Table 2–8 for examples of predefined policies that use Bearer confirmation.

### 2.11.2.1 Specifying the SAML Confirmation Method (Proprietary Policy Only)

This section describes how to specify the SAML confirmation method in a policy file that uses the proprietary schema for security policy.

---

**Note:** SAML V1.1 and V2.0 assertions use <saml:SubjectConfirmation> and <saml2:SubjectConfimation> elements, respectively, to specify the confirmation method; the confirmation method is not directly specified in the policy file.

---

When you configure a Web service to require SAML tokens for identity, you can specify one of the following confirmation methods:

- `sender-vouches`
- `holder-of-key`

See "SAML Token Profile Support in WebLogic Web services", as well as the *Web Services Security: SAML Token Profile* (http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SAMLTokenProfile.pdf) specification itself, for details about these confirmation methods.

1. Use a security policy file that specifies that SAML should be used for identity. The exact syntax depends on the type of confirmation method you want to configure (`sender-vouches`, `holder-of-key`).

   **To specify the sender-vouches confirmation method:**

   a. Create a <SecurityToken> child element of the <Identity><SupportedTokens> elements and set the TokenType attribute to a value that indicates SAML token usage.

   b. Add a <Claims><Confirmationmethod> child element of <SecurityToken> and specify `sender-vouches`.

   For example:

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecuri
ty-utility-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
  >
  <wssp:Identity>
    <wssp:SupportedTokens>
      <wssp:SecurityToken

TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-
profile-1.0#SAMLAssertionID">
        <wssp:Claims>
          <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
        </wssp:Claims>
      </wssp:SecurityToken>
```

```
    </wssp:SupportedTokens>
  </wssp:Identity>
</wsp:Policy>
```

**To specify the holder-of-key confirmation method:**

**a.** Create a `<SecurityToken>` child element of the `<Integrity><SupportedTokens>` elements and set the `TokenType` attribute to a value that indicates SAML token usage.

The reason you put the SAML token in the `<Integrity>` assertion for the `holder-of-key` confirmation method is that the Web service runtime must prove the integrity of the message, which is not required by `sender-vouches`.

**b.** Add a `<Claims><Confirmationmethod>` child element of `<SecurityToken>` and specify `holder-of-key`.

For example:

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecuri
ty-utility-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part">
  <wssp:Integrity>
    <wssp:SignatureAlgorithm
       URI="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <wssp:CanonicalizationAlgorithm
       URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    <wssp:Target>
      <wssp:DigestAlgorithm
         URI="http://www.w3.org/2000/09/xmldsig#sha1" />
      <wssp:MessageParts
         Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
         wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:SupportedTokens>
      <wssp:SecurityToken
          IncludeInMessage="true"

TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-
profile-1.0#SAMLAssertionID">
        <wssp:Claims>
          <wssp:ConfirmationMethod>holder-of-key</wssp:ConfirmationMethod>
        </wssp:Claims>
      </wssp:SecurityToken>
    </wssp:SupportedTokens>
  </wssp:Integrity>
</wsp:Policy>
```

**c.** By default, the WebLogic Web services runtime always validates the X.509 certificate specified in the `<KeyInfo>` assertion of any associated WS-Policy file. To disable this validation when using SAML holder-of-key assertions, you must configure the Web service security configuration associated with the Web service by setting a property on the SAML token handler. See "Disable X.509 certificate validation when using SAML holder_of_key assertions" in *Oracle*

*WebLogic Server Administration Console Help* for information on how to do this using the Administration Console.

See Section 2.7, "Creating and Using a Custom Policy File" for additional information about creating your own security policy file. See "Web Services Security Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server* for reference information about the assertions.

2. Update the appropriate `@Policy` annotations in the JWS file that implements the Web service to point to the security policy file from the preceding step. For example, if you want invokes of *all* the operations of a Web service to SAML for identity, specify the `@Policy` annotation at the class-level.

   You can mix and match the policy files that you associate with a Web service, as long as they do not contradict each other and as long as you do not combine OASIS WS-SecurityPolicy 1.2 files with security policy files written under Oracle's security policy schema.

   For example, you can create a simple `MyAuth.xml` file that contains only the `<Identity>` security assertion to specify use of SAML for identity and then associate it with the Web service together with the predefined `Wssp1.2-2007-EncryptBody.xml` and `Wssp1.2-2007-SignBody.xml` files. It is, however, up to you to ensure that multiple associated policy files do not contradict each other; if they do, you will either receive a runtime error or the Web service might not behave as you expect.

3. Recompile and redeploy your Web service as part of the normal iterative development process.

   See "Developing WebLogic Web Services" in *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*.

4. Create a client application that runs in a WebLogic Server instance to invoke the main Web service using SAML as identity. See Section 2.5.1, "Invoking a Web Service From a Client Running in a WebLogic Server Instance" for details.

## 2.12 Associating a Web Service with a Security Configuration Other Than the Default

Many use cases previously discussed require you to use the Administration Console to create the default Web service security configuration called `default_wss`. After you create this configuration, it is applied to all Web services that either do *not* use the `@weblogic.jws.security.WssConfiguration` JWS annotation or specify the annotation with no attribute.

There are some cases, however, in which you might want to associate a Web service with a security configuration *other* than the default; such use cases include specifying different timestamp values for different services.

To associate a Web service with a security configuration other than the default:

1. "Create a Web Service Security Configuration" in the *Oracle WebLogic Server Administration Console Help* with a name that is *not* `default_wss`.

2. Update your JWS file, adding the `@WssConfiguration` annotation to specify the name of this security configuration. See "weblogic.jws.security.WssConfiguration" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information and an example.

**Note:** If you are going to package additional Web services in the same Web application, and these Web services also use the @WssConfiguration annotation, then you must specify the same security configuration for each Web service. See "weblogic.jws.security.WssConfiguration" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for more details.

3. Recompile and redeploy your Web service as part of the normal iterative development process.

   See "Invoking Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* and "Developing WebLogic Web Services" in *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*.

**Note:** All Web services security configurations are required to specify the same password digest use. Inconsistent password digest use in different Web service security configurations will result in a runtime error.

## 2.13 Valid Class Names and Token Types for Credential Provider

When you create a security configuration, you need to supply the class name of the credential provider for this configuration. The valid class names and token types you can use are as follows:

- `weblogic.wsee.security.bst.ClientBSTCredentialProvider`. The token type is `x509`.
- `weblogic.wsee.security.unt.ClientUNTCredentialProvider`. The token type is `ut`.
- `weblogic.wsee.security.wssc.v13.sct.ClientSCCredentialProvider`. The token type is `sct`.
- `weblogic.wsee.security.wssc.v200502.sct.ClientSCCredentialProvider`. The token type is `sct`.
- `weblogic.wsee.security.saml.SAMLTrustCredentialProvider`. The token type is `saml`.

## 2.14 Using System Properties to Debug Message-Level Security

The following table lists the system properties you can set to debug problems with your message-secured Web service.

***Table 2–2 System Properties for Debugging Message-Level Security***

| System Property | Data Type | Description |
|---|---|---|
| weblogic.xml.crypto.dsig.verbose | Boolean | Prints information about digital signature processing. |
| weblogic.xml.crypto.encrypt.verbose | Boolean | Prints information about encryption processing. |
| weblogic.xml.crypto.keyinfo.verbose | Boolean | Prints information about key resolution processing. |

*Table 2–2 (Cont.) System Properties for Debugging Message-Level Security*

| System Property | Data Type | Description |
|---|---|---|
| weblogic.xml.crypto.wss.verbose | Boolean | Prints information about Web service security token and token reference processing. |

## 2.15 Using a Client-Side Security Policy File

The section Section 2.3, "Using Policy Files for Message-Level Security Configuration" describes how a WebLogic Web service can be associated with one or more security policy files that describe the message-level security of the Web service. These policy files are XML files that describe how a SOAP message should be digitally signed or encrypted and what sort of user authentication is required from a client that invokes the Web service. Typically, the policy file associated with a Web service is attached to its WSDL, which the Web services client runtime reads to determine whether and how to digitally sign and encrypt the SOAP message request from an operation invoke from the client application.

Sometimes, however, a Web service might not attach the policy file to its deployed WSDL or the Web service might be configured to not expose its WSDL at all. In these cases, the Web services client runtime cannot determine from the service itself the security that must be enabled for the SOAP message request. Rather, it must load a client-side copy of the policy file. This section describes how to update a client application to load a local copy of a policy file.

Example 2–5 shows an example of using a client-side policy file from a JAX-WS Web service.

The client-side policy file is typically exactly the same as the one associated with a deployed Web service. If the two files are different, and there is a conflict in the security assertions contained in the files, then the invoke of the Web service operation returns an error.

You can specify that the client-side policy file be associated with the SOAP message request, response, or both. Additionally, you can specify that the policy file be associated with the entire Web service, or just one of its operations.

### 2.15.1 Associating a Policy File with a Client Application: Main Steps

The following procedure describes the high-level steps to associate a security policy file with the client application that invokes a Web service operation.

It is assumed that you have created the client application that invokes a deployed Web service, and that you want to update it by associating a client-side policy file. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `clientgen` Ant task.

See "Invoking Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* and "Invoking a Web Service from a Stand-alone Client: Main Steps" in *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*.

1. Create the client-side security policy files and save them in a location accessible by the client application. Typically, the security policy files are the same as those configured for the Web service you are invoking, but because the server-side files are not exposed to the client runtime, the client application must load its own local copies.

See Section 2.7, "Creating and Using a Custom Policy File" for information about creating security policy files.

2. Update the `build.xml` file that builds your client application.
3. Update your Java client application to load the client-side policy files
4. Rebuild your client application by running the relevant task. For example:

```
prompt> ant build-client
```

When you next run the client application, it will load local copies of the policy files that the Web service client runtime uses to enable security for the SOAP request message.

> **Note:** If you have a Web services operation that already have a security policy (for example, one that was set in the WSDL file that was stored when generating the client from the server policy), then when you use this procedure to programmatically set the client-side security policy, all previously-existing policies will be removed.

## 2.15.2 Updating clientgen to Generate Methods That Load Policy Files

For JAX-RPC, set the `generatePolicyMethods` attribute of the `clientgen` Ant task to `true` to specify that the Ant task should generate additional `getXXX()` methods in the implementation of the JAX-RPC `Service` interface for loading client-side copies of policy files when you get a port, as shown in the following example:

```
<clientgen
  wsdl="http://ariel:7001/policy/ClientPolicyService?WSDL"
  destDir="${clientclass-dir}"
  generatePolicyMethods="true"
  packageName="examples.webservices.client_policy.client"/>
```

See Section 2.15.3, "Updating a Client Application To Load Policy Files (JAX-RPC Only)" for a description of the additional methods that are generated and how to use them in a client application.

### JAX-WS Usage

For JAX-WS, you use the `weblogic.jws.jaxws.ClientPolicyFeature` class to override the effective policy defined for a service. `weblogic.jws.jaxws.ClientPolicyFeature` extends `javax.xml.ws.WebServiceFeature`.

## 2.15.3 Updating a Client Application To Load Policy Files (JAX-RPC Only)

When you set `generatePolicyMethods="true"` for `clientgen`, the Ant task generates additional methods in the implementation of the JAX-RPC `Service` interface that you can use to load policy files, where *XXX* refers to the name of the Web service.

You can use either an Array or Set of policy files to associate multiple files to a Web service. If you want to associate just a single policy file, create a single-member Array or Set.

- `getXXXPort(String operationName, java.util.Set<java.io.InputStream> inbound, java.util.Set<java.io.InputStream> outbound)`

Loads two different sets of client-side policy files from InputStreams and associates the first set to the SOAP request and the second set to the SOAP response. Applies to a specific operation, as specified by the first parameter.

- `getXXXPort(String operationName, java.io.InputStream[] inbound, java.io.InputStream[] outbound)`

  Loads two different arrays of client-side policy files from InputStreams and associates the first array to the SOAP request and the second array to the SOAP response. Applies to a specific operation, as specified by the first parameter.

- `getXXXPort(java.util.Set<java.io.InputStream> inbound, java.util.Set<java.io.InputStream> outbound)`

  Loads two different sets of client-side policy files from InputStreams and associates the first set to the SOAP request and the second set to the SOAP response. Applies to all operations of the Web service.

- `getXXXPort(java.io.InputStream[] inbound, java.io.InputStream[] outbound)`

  Loads two different arrays of client-side policy files from InputStreams and associates the first array to the SOAP request and the second array to the SOAP response. Applies to all operations of the Web service.

Use these methods, rather than the normal `getXXXPort()` method with no parameters, for getting a Web service port and specifying at the same time that invokes of all, or the specified, operation using that port have an associated policy file or files.

---

**Note:** The following methods from a previous release of WebLogic Server have been deprecated; if you want to associate a single client-side policy file, specify a single-member Array or Set and use the corresponding method described above.

---

- `getXXXPort(java.io.InputStream policyInputStream);`

  Loads a single client-side policy file from an InputStream and applies it to both the SOAP request (inbound) and response (outbound) messages.

- `getXXXPort(java.io.InputStream policyInputStream, boolean inbound, boolean outbound);`

  Loads a single client-side policy file from an `InputStream` and applies it to either the SOAP request or response messages, depending on the Boolean value of the second and third parameters.

Example 2–14 shows an example of using these policy methods in a simple client application; the code in bold is described after the example.

***Example 2–14 Loading Policies in a Client Application***

```
package examples.webservices.client_policy.client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
import java.io.FileInputStream;
import java.io.IOException;
/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the ClientPolicyService Web service.
 *
```

```
 * @author Copyright © 1996, 2008, Oracle and/or its affiliates.
 * All rights reserved.
 */
public class Main {
  public static void main(String[] args)
      throws ServiceException, RemoteException, IOException {
      FileInputStream [] inbound_policy_array = new FileInputStream[2];
      inbound_policy_array[0] = new FileInputStream(args[1]);
      inbound_policy_array[1] = new FileInputStream(args[2]);
      FileInputStream [] outbound_policy_array = new FileInputStream[2];
      outbound_policy_array[0] = new FileInputStream(args[1]);
      outbound_policy_array[1] = new FileInputStream(args[2]);
      ClientPolicyService service = new ClientPolicyService_Impl(args[0] +
"?WSDL");
      // standard way to get the Web service port
      ClientPolicyPortType normal_port = service.getClientPolicyPort();
      // specify an array of policy files for the request and response
      // of a particular operation
      ClientPolicyPortType array_of_policy_port =
service.getClientPolicyPort("sayHello",
inbound_policy_array, outbound_policy_array);
    try {
      String result = null;
      result = normal_port.sayHello("Hi there!");
      result = array_of_policy_port.sayHello("Hi there!");
      System.out.println( "Got result: " + result );
    } catch (RemoteException e) {
      throw e;
    }
  }
}
```

The second and third argument to the client application are the two policy files from which the application makes an array of `FileInputStreams` (`inbound_policy_array` and `outbound_policy_array`). The `normal_port` uses the standard parameterless method for getting a port; the `array_of_policy_port`, however, uses one of the policy methods to specify that an invoke of the `sayHello` operation using the port has multiple policy files (specified with an Array of `FileInputStream`) associated with both the inbound and outbound SOAP request and response:

```
ClientPolicyPortType array_of_policy_port =
 service.getClientPolicyPort("sayHello", inbound_policy_array, outbound_policy_
array);
```

## 2.16 Using WS-SecurityPolicy 1.2 Policy Files

WebLogic Server includes a number of WS-SecurityPolicy files you can use in most Web services applications. The policy files are located in *MW_HOME/WL_HOME*`/server/lib/weblogic.jar`. Within `weblogic.jar`, the policy files are located in `/weblogic/wsee/policy/runtime`.

There are two sets of these policies. In most of the cases, they perform identical functions, but the policy uses different namespace.

The first set has a prefix of "Wssp1.2-2007-". These security policy files conform to the OASIS WS-SecurityPolicy 1.2 specification and have the following namespace:

```
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
```

```
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
  >
```

The second set carries over from WebLogic Server version 10.0 and has the prefix "Wssp1.2-":

```
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512"
  >
```

Oracle recommends that you use the new policy namespace, as those are official namespaces from OASIS standards and they will perform better when interoperating with other vendors. The old policies having the prefix of "Wssp1.2-" are mainly for users who want to interoperate with existing applications that already use this version of the policies.

The following sections describe the available WS-SecurityPolicy 1.2 policy files:

- Section 2.16.1, "Transport Level Policies"
- Section 2.16.2, "Protection Assertion Policies"
- Section 2.16.3, "WS-Security 1.0 Username and X509 Token Policies"
- Section 2.16.4, "WS-Security 1.1 Username and X509 Token Policies"
- Section 2.16.5, "WS-SecureConversation Policies"
- Section 2.16.6, "SAML Token Profile Policies"

In addition, see Section 2.17, "Choosing a Policy" and Section 2.21.2, "Configuring Smart Policy Selection" for information about how to choose the best security policy approach for your Web services implementation and for information about WS-SecurityPolicy 1.2 elements that are not supported in this release of WebLogic Server.

### 2.16.1 Transport Level Policies

These policies require use of the `https` protocol to access WSDL and invoke Web services operations:

> **Note:** If you specify a transport-level security policy for your Web service, it must be at the class level.
>
> In addition, the transport-level security policy must apply to both the inbound and outbound directions. That is, you cannot have HTTPS for inbound and HTTP for outbound.

***Table 2–3 Transport Level Policies***

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Https.xml | One way SSL. |
| Wssp1.2-2007-Https-BasicAuth.xml | One way SSL with Basic Authentication. A 401 challenge occurs if the Authorization header is not present in the request. |

*Table 2–3 (Cont.) Transport Level Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Https-ClientCertReq.xml | Two way SSL. The recipient checks for the initiator's public certificate. Note that the client certificate can be used for authentication.<br>Set **Two Way Client Cert Behavior** to "Client Certs Requested But Not Enforced." See "Configure two-way SSL" in *Oracle WebLogic Server Administration Console Help* for information on how to do this. |
| Wssp1.2-2007-Https-UsernameToken-Digest.xml | One way SSL with digest Username Token. |
| Wssp1.2-2007-Https-UsernameToken-Plain.xml | One way SSL with plain text Username Token. |
| Wssp1.2-Https.xml | One way SSL. |
| Wssp1.2-Https-BasicAuth.xml | One way SSL with Basic Authentication. A 401 challenge occurs if the Authorization header is not present in the request. |
| Wssp1.2-Https-UsernameToken-Digest.xml | One way SSL with digest Username Token. |
| Wssp1.2-Https-UsernameToken-Plain.xml | One way SSL with plain text Username Token. |
| Wssp1.2-Https-ClientCertReq.xml | Two way SSL. The recipient checks for the initiator's public certificate. Note that the client certificate can be used for authentication. |

## 2.16.2 Protection Assertion Policies

Protection assertions are used to identify what is being protected and the level of protection provided. Protection assertion policies cannot be used alone; they should be used only in combination with X.509 Token Policies. For example, you might use `Wssp1.2-2007-Wss1.1-X509-Basic256.xml` together with `Wssp1.2-2007-SignBody.xml`. The following policy files provide for the protection of message parts by signing or encryption:

*Table 2–4 Protection Assertion Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-SignBody.xml | All message body parts are signed. |
| Wssp1.2-2007-EncryptBody.xml | All message body parts are encrypted. |
| Wssp1.2-2007-Sign-Wsa-Headers.xml | WS-Addressing headers are signed. |
| Wssp1.2-SignBody.xml | All message body parts are signed. |
| Wssp1.2-EncryptBody.xml | All message body parts are encrypted. |
| Wssp1.2-Sign-Wsa-Headers.xml | WS-Addressing headers are signed. |
| Wssp1.2-2007-SignAndEncryptWSATHeaders.xml | WS-AtomicTransaction headers are signed and encrypted. |

*Table 2–4 (Cont.) Protection Assertion Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Wsp1.5-SignAndEncryptWSATHeaders.xml | WS-AtomicTransaction headers are signed and encrypted. Web Services Policy 1.5 is used. |

## 2.16.3 WS-Security 1.0 Username and X509 Token Policies

The following policies support the Username Token or X.509 Token specifications of WS-Security 1.0:

*Table 2–5 WS-Security 1.0 Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Wss1.0-X509-Basic256.xml | Mutual Authentication with X.509 Certificates. The message is signed and encrypted on both request and response. The algorithm of Basic256 should be used for both sides. |
| Wssp1.2-2007-Wss1.0-UsernameToken-Digest-X509-Basic256.xml | Username token with digested password is sent in the request for authentication. The encryption method is Basic256. |
| Wssp1.2-2007-Wss1.0-UsernameToken-Plain-X509-Basic256.xml | Username token with plain text password is sent in the request for authentication, signed with the client's private key and encrypted with server's public key. The client also signs the request body and includes its public certificate, protected by the signature in the message. The server signs the response body with its private key and sends its public certificate in the message. Both request and response messages include signed time stamps. The encryption method is Basic256. |
| Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic256.xml | Username token with plain text password is sent in the request for authentication, signed with the client's private key and encrypted with server's public key. The client also signs the request body and includes its public certificate, protected by the signature in the message. The server signs the response body with its private key and sends its public certificate in the message. Both request and response messages include signed time stamps. The encryption method is Basic256. |
| Wssp1.2-Wss1.0-UsernameToken-Plain-X509-TripleDesRsa15.xml | Username token with plain text password is sent in the request for authentication, signed with the client's private key and encrypted with server's public key. The client also signs the request body and includes its public certificate, protected by the signature in the message. The server signs the response body with its private key and sends its public certificate in the message. Both request and response messages include signed time stamps. The encryption method is TripleDes. |
| Wssp1.2-Wss1.0-UsernameToken-Digest-X509-Basic256.xml | Username token with digested password is sent in the request for authentication. The encryption method is Basic256. |
| Wssp1.2-Wss1.0-UsernameToken-Digest-X509-TripleDesRsa15.xml | Username token with digested password is sent in the request for authentication. The encryption method is TripleDes. |
| Wssp1.2-Wss1.0-X509-Basic256.xml | Mutual Authentication with X.509 Certificates. The message is signed and encrypted on both request and response. The algorithm of Basic256 should be used for both sides. |
| Wssp1.2-Wss1.0-X509-TripleDesRsa15.xml | Mutual Authentication with X.509 Certificates and message is signed and encrypted on both request and response. The algorithm of TripleDes should be used for both sides |

*Table 2–5 (Cont.) WS-Security 1.0 Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-Wss1.0-X509-EncryptRequest-SignResponse.xml | This policy is used where only the server has X.509v3 certificates (and public-private key pairs). The request is encrypted and the response is signed. |

## 2.16.4 WS-Security 1.1 Username and X509 Token Policies

The following policies support the Username Token or X.509 Token specifications of WS-Security 1.1:

*Table 2–6 WS-Security 1.1 Username and X509 Token Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Wss1.1-X509-Basic256.xml | WSS 1.1 X509 with asymmetric binding. |
| Wssp1.2-2007-Wss1.1-UsernameToken-Digest-X509-Basic256.xml | WSS 1.1 X509 with asymmetric binding and authentication with digested Username Token. |
| Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256.xml | WSS 1.1 X509 with asymmetric binding and authentication with plain-text Username Token. |
| Wssp1.2-2007-Wss1.1-EncryptedKey-X509-SignedEndorsing.xml | WSS 1.1 X509 with symmetric binding and protected by signed endorsing supporting token. |
| Wssp1.2-2007-Wss1.1-UsernameToken-Digest-EncryptedKey.xml | WSS 1.1 X509 with symmetric binding and authentication with digested Username Token. |
| Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey.xml | WSS 1.1 X509 with symmetric binding and authentication with plain-text Username Token. |
| Wssp1.2-2007-Wss1.1-DK-X509-SignedEndorsing.xml | WSS 1.1 X509 with derived key symmetric binding and protected by signed endorsing supporting token. |
| Wssp1.2-2007-Wss1.1-UsernameToken-Digest-DK.xml | WSS 1.1 X509 with derived key symmetric binding and authentication with digested Username Token. |
| Wssp1.2-2007-Wss1.1-UsernameToken-Plain-DK.xml | WSS 1.1 X509 with derived key symmetric binding and authentication with plain-text Username Token. |
| Wssp1.2-Wss1.1-X509-Basic256.xml | This policy is similar to policy Wssp1.2-Wss1.0-X509-Basic256.xml except it uses additional WS-Security 1.1 features, including Signature Confirmation and Thumbprint key reference. |
| Wssp1.2-Wss1.1-EncryptedKey.xml | This is a symmetric binding policy that uses the WS-Security 1.1 Encrypted Key feature for both signature and encryption. It also uses WS-Security 1.1 features, including Signature Confirmation and Thumbprint key reference. |
| Wssp1.2-Wss1.1-UsernameToken-DK.xml | WSS 1.1 X509 with derived key symmetric binding and authentication with plain-text Username Token. |
| Wssp1.2-Wss1.1-EncryptedKey-X509-SignedEndorsing.xml | This policy has all of the features defined in policy Wssp1.2-Wss1.1-EncryptedKey.xml, and in addition it uses sender's key to endorse the message signature. The endorsing key is also signed with the message signature. |

*Table 2–6 (Cont.) WS-Security 1.1 Username and X509 Token Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-Wss1.1-DK.xml | This policy has all of features defined in policy Wssp1.2-Wss1.1-EncryptedKey.xml, except that instead of using an encrypted key, the request is signed using DerivedKeyToken1, then encrypted using a DerivedKeyToken2. Response is signed using DerivedKeyToken3, and encrypted using DerivedKeyToken4. |
| Wssp1.2-Wss1.1-DK-X509-Endorsing.xml | This policy has all features defined in policy Wssp1.2-Wss1.1-DK.xml, and in addition it uses the sender's key to endorse the message signature. |
| Wssp1.2-Wss1.1-X509-EncryptRequest-SignResponse.xml | This policy is similar to policy Wssp1.2-Wss1.0-X509-EncryptRequest-SignResponse.xml, except that it uses additional WSS 1.1 features, including Signature Confirmation and Thumbprint key reference. |
| Wssp1.2-Wss1.1-X509-SignRequest-EncryptResponse.xml | This policy is the reverse of policy Wssp1.2-Wss1.1-X509-EncryptRequest-SignResponse.xml: the request is signed and the response is encrypted. |
| Wssp1.2-wss11_x509_token_with_message_protection_owsm_policy.xml | This policy endorses with the sender's X509 certificate, and the message signature is protected. It requires the use of the Basic128 algorithm suite (AES128 for encryption) instead of the Basic256 algorithm suite (AES256). |

## 2.16.5 WS-SecureConversation Policies

The policies in Table 2–7 implement WS-SecureConversation 1.3 and WS-SecureConversation 2005/2.

If you specify a WS-SecureConversation policy for your Web service, it must be at the class level.

*Table 2–7 WS-SecureConversation Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Wssc1.3-Bootstrap-Https-BasicAuth.xml | One way SSL with Basic Authentication. Timestamp is included. The algorithm suite is Basic256. The signature is encrypted. |
| Wssp1.2-2007-Wssc1.3-Bootstrap-Https-ClientCertReq.xml | Two way SSL. The recipient checks for the initiator's public certificate. Note that the client certificate can be used for authentication. |
| Wssp1.2-2007-Wssc1.3-Bootstrap-Https-UNT.xml | SSL Username token authentication. |
| Wssp1.2-2007-Wssc1.3-Bootstrap-Https.xml | WS-SecureConversation handshake (RequestSecurityToken and RequestSecurityTokenResponseCollection messages) occurs in https transport. The application messages are signed and encrypted with DerivedKeys. The signature is also encrypted. |
| Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.0.xml | WS-SecureConversation handshake is protected by WS-Security 1.0. The application messages are signed and encrypted with DerivedKeys. The soap:Body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages are both signed and encrypted. The WS-Addressing headers are signed. Timestamp is included and signed. The signature is encrypted. The algorithm suite is Basic256. |

*Table 2–7 (Cont.) WS-SecureConversation Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.1.xml | WS-SecureConversation handshake is protected by WS-Security 1.1. The application messages are signed and encrypted with DerivedKeys. The soap:Body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages are both signed and encrypted. The WS-Addressing headers are signed. Signature and encryption use derived keys from an encrypted key. |
| Wssp1.2-Wssc1.3-Bootstrap-Https-BasicAuth.xml | One way SSL with Basic Authentication. Timestamp is included. The algorithm suite is Basic256. The signature is encrypted. |
| Wssp1.2-Wssc1.3-Bootstrap-Https-ClientCertReq.xml | Two way SSL. The recipient checks for the initiator's public certificate. Note that the client certificate can be used for authentication. |
| Wssp1.2-Wssc1.3-Bootstrap-Https.xml | WS-SecureConversation handshake (RequestSecurityToken and RequestSecurityTokenResponseCollection messages) occurs in https transport. The application messages are signed and encrypted with DerivedKeys. The signature is also encrypted. |
| Wssp1.2-Wssc1.3-Bootstrap-Wss1.0.xml | WS-SecureConversation handshake is protected by WS-Security 1.0. The application messages are signed and encrypted with DerivedKeys. The soap:Body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages are both signed and encrypted. The WS-Addressing headers are signed. Timestamp is included and signed. The signature is encrypted. The algorithm suite is Basic256. |
| Wssp1.2-Wssc1.3-Bootstrap-Wss1.1.xml | WS-SecureConversation handshake is protected by WS-Security 1.1. The application messages are signed and encrypted with DerivedKeys. The soap:Body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages are both signed and encrypted. The WS-Addressing headers are signed. Signature and encryption use derived keys from an encrypted key. |
| Wssp1.2-Wssc200502-Bootstrap-Https.xml | WS-SecureConversation handshake (RequestSecurityToken and RequestSecurityTokenResponse messages) occurs in https transport. The application messages are signed and encrypted with DerivedKeys. |
| Wssp1.2-Wssc200502-Bootstrap-Wss1.0.xml | WS-SecureConversation handshake is protected by WS-Security 1.0. The application messages are signed and encrypted with DerivedKeys. The soap:Body of the RequestSecurityToken and RequestSecurityTokenResponse messages are both signed and encrypted. The WS-Addressing headers are signed. Timestamp is included and signed. The algorithm suite is Basic128. |
| Wssp1.2-Wssc200502-Bootstrap-Wss1.1.xml | WS-SecureConversation handshake is protected by WS-Security 1.1. The application messages are signed and encrypted with DerivedKeys. The soap:Body of the RequestSecurityToken and RequestSecurityTokenResponse messages are both signed and encrypted. The WS-Addressing headers are signed. Signature and encryption use derived keys from an encrypted key. |

### 2.16.6 SAML Token Profile Policies

The policies shown in Table 2–1 implement WS-Security SAML Token Profile 1.0 and 1.1.

> **Note:** WebLogic Server Version 10.3 supported SAML Holder of Key for the inbound request only. As of WebLogic Server Version 10.3MP1 and later, both the request and response messages are protected.

***Table 2–8 WS-Security SAML Token Profile Policies***

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Saml1.1-SenderVouches-Wss1.0.xml | The message is signed and encrypted on both request and response with WSS1.0 asymmetric binding. SAML 1.1 token is sent in the request for authentication with Sender Vouches confirmation method, signed by the X509 token. |
| Wssp1.2-2007-Saml1.1-SenderVouches-Wss1.1.xml | The message is signed and encrypted on both request and response with WSS1.1 X509 symmetric binding. SAML 1.1 token is sent in the request for authentication with Sender Vouches confirmation method, signed by the X509 token. |
| Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml | The message is signed and encrypted on both request and response with WSS1.1 X509 symmetric binding. SAML 2.0 token is sent in the request for authentication with Sender Vouches confirmation method, signed by the X509 token. |
| Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1-Asymmetric.xml | The message is signed and encrypted on both request and response with WSS1.1 asymmetric binding. It uses additional WS-Security 1.1 features, including Signature Confirmation and Thumbprint key reference. SAML 2.0 token is sent in the request for authentication with Sender Vouches confirmation method, signed by the X509 token. |
| Wssp1.2-2007-Saml1.1-HolderOfKey-Wss1.0.xml | The message is signed and encrypted on both request and response with WSS1.0 asymmetric binding. SAML 1.1 token is sent in the request for authentication with Holder of Key confirmation method, in which the key inside the SAML Token is used for the signature. |
| Wssp1.2-2007-Saml1.1-HolderOfKey-Wss1.1-Asymmetric.xml | The message is signed and encrypted on both request and response with WSS1.1 asymmetric binding. It uses additional WS-Security 1.1 features, including Signature Confirmation and Thumbprint key reference. SAML 1.1 token is sent in the request for authentication with Holder of Key confirmation method, in which the key inside the SAML Token is used for the signature. |
| Wssp1.2-2007-Saml2.0-HolderOfKey-Wss1.1-Asymmetric.xml | The message is signed and encrypted on both request and response with WSS1.1 asymmetric binding. It uses additional WS-Security 1.1 features, including Signature Confirmation and Thumbprint key reference. SAML 2.0 token is sent in the request for authentication with Holder of Key confirmation method, in which the key inside the SAML Token is used for the signature. |

*Table 2–8 (Cont.) WS-Security SAML Token Profile Policies*

| Policy File | Description |
|---|---|
| Wssp1.2-2007-Saml2.0-Bearer-Https.xml | One-way SSL uses SAML 2.0 token with Bearer confirmation method for Authentication.<br>WebLogic Server supports the SAML 2.0 Bearer confirmation method at the transport level, using *Wssp1.2-2007-Saml2.0-Bearer-Https.xml*.<br>SAML 1.1 Bearer is not supported.<br>To interoperate with other products that do not support SAML 2.0, for the SAML-over-HTTPS scenario, the sender vouches confirmation method is recommended.<br>Use the *Wssp1.2-2007-Saml1.1-SenderVouches-Https.xml* policy for this purpose, instead of using SAML 1.1 Bearer.<br>If you specify a transport-level security policy for your Web service, it must be at the class level. In addition, the transport-level security policy must apply to both the inbound and outbound directions. That is, you cannot have HTTPS for inbound and HTTP for outbound. |
| Wssp1.2-wss11_saml_token_with_message_protection_owsm_policy.xml | This policy endorses with the sender's X509 certificate, and message signature is protected. It requires the use of the Basic128 algorithm suite (AES128 for encryption) instead of the Basic256 algorithm suite (AES256). |

## 2.17 Choosing a Policy

WebLogic Server's implementation of WS-SecurityPolicy 1.2 makes a wide variety of security policy alternatives available to you. When choosing a security policy for your Web service, you should consider your requirements in these areas:

- Performance
- Security
- Interoperability
- Credential availability (X.509 certificate, username token, clear or digest password)

Whenever possible, Oracle recommends that you:

- Use a policy packaged in WebLogic Server rather than creating a custom policy.
- Use a WS-SecurityPolicy 1.2 policy rather than a WebLogic Server 9.x style policy, unless you require features that are not yet supported by WS-SecurityPolicy 1.2 policies.
- Use transport-level policies (`Wssp1.2-2007-Https-*.xml`) only where message-level security is not required.
- Use WS-Security 1.0 policies if you require interoperability with that specification. Use one of the following, depending on your authentication requirements and credential availability:
  - Wssp1.2-2007-Wss1.0-UsernameToken-Plain-X509-Basic256.xml
  - Wssp1.2-2007-Wss1.0-UsernameToken-Digest-X509-Basic256.xml
  - Wssp1.2-2007-Wss1.0-X509-Basic256.xml
- Use WS-Security 1.1 policies if you have strong security requirements. Use one of the following:

  - Wssp1.2-2007-Wss1.1-EncryptedKey-X509-SignedEndorsing.xml
  - Wssp1.2-2007-Wss1.1-DK-X509-SignedEndorsing.xml
  - Wssp1.2-Wss1.1-EncryptedKey-X509-SignedEndorsing.xml
  - Wssp1.2-Wss1.1-DK-X509-Endorsing.xml
- Use a WS-SecureConversation policy where WS-ReliableMessaging plus security are required:
  - Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.0.xml
  - Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.1.xml
  - Wssp1.2-Wssc1.3-Bootstrap-Wss1.0.xml
  - Wssp1.2-Wssc1.3-Bootstrap-Wss1.1.xml
  - Wssp1.2-Wssc200502-Bootstrap-Wss1.0.xml
  - Wssp1.2-Wssc200502-Bootstrap-Wss1.1.xml

## 2.18 Unsupported WS-SecurityPolicy 1.2 Assertions

The WS-SecurityPolicy 1.2 assertions in Table 2–1 are not supported in this release of WebLogic Server.

> **Note:** New WS-SecurityPolicy 1.3 assertions are also not supported in this release.

***Table 2–9 Web Services SecurityPolicy 1.2 Unsupported Assertions***

| Specification | Assertion | Remarks |
|---|---|---|
| 5.1.1 | TokenInclusion | includeTokenPolicy=Once is not supported. |
| 5.4.1 | UsernameToken | Only <sp:UsernameToken11> and Password Derived Keys are not supported in this release. Other Username Tokens assertions are supported. |
| 5.4.2 | IssuedToken | WS-Trust Policy assertion is not supported in this release. |
| 5.4.3 | X509Token | Support all token types. |
| 5.4.4 | KerberosToken | Not supported in this release. |
| 5.4.5 | SpnegoContextToken | Not supported in this release. |
| 5.4.9 | RelToken | Not supported in this release. |
| 5.4.11 | KeyValueToken | Not supported in this release. |
| 6.5 | Token Protection | Token Protection in cases where includeTokenPolicy="Never", or in cases where the Token is not in the Message, is not supported in this release. |

*Table 2–9 (Cont.) Web Services SecurityPolicy 1.2 Unsupported Assertions*

| Specification | Assertion | Remarks |
|---|---|---|
| 7.1 | AlgorithmSuite | /sp:AlgorithmSuite/wsp:Policy/sp:XPathFilter20 assertion, /sp:AlgorithmSuite/wsp:Policy/sp:XPath10 assertion and /sp:AlgorithmSuite/wsp:Policy/sp:SoapNormalization10 are not supported in this release. |
| 8.1 | SupportingTokens | Not supported in this release:<br>../sp:SignedParts assertion,<br>../sp:SignedElements assertion<br>../sp:EncryptedParts assertion<br>../sp:EncryptedElements assertion |
| 8.2<br>8.3<br>8.4<br>8.5 | SignedSupportingTokens<br>EndorsingSupportingTokens<br>SignedEndorsingSupportingTokens<br>SignedEncrtptedSupportingTokens | Not supported in this release:<br>../sp:SignedParts assertion<br>../sp:SignedElements assertion<br>../sp:EncryptedParts assertion<br>../sp:EncryptedElements assertion<br>The runtime will not be able to endorse the supporting token in cases where the token is not in the Message (such as for includeTokenPolicy=Never/Once). |
| 8.6 | EncryptedSupportingTokens | UserName Token is the only EncryptionSupportingTokens supported in this release.<br>Other type of tokens are not supported. |
| 8.7 | EndorsingEncryptedSupportingTokens | Not supported in this release. |
| 8.8 | SignedEndorsingEncryptedSupportingTokens | Not supported in this release. |
| 9.1 | WSS10 Assertion | <sp:MustSupportRefExternalURI> and <sp:MustSupportRefEmbeddedToken> are not supported in this release. |
| 9.2 | WSS11 Assertion | <sp:MustSupportRefExternalURI> and <sp:MustSupportRefEmbeddedToken> are not supported in this release. |
| 10.1 | Trust13 Assertion | MustSupportClientChallenge, MustSupportServerChallenge are not supported in this release. This assertion is supported only in WS-SecureConversation policy. |

## 2.19 Using the Optional Policy Assertion

WebLogic Server supports the `Optional` WS-Policy assertion. Consider the use of `Optional` in the following example:

```
<sp:SignedEncryptedSupportingTokens>
    <wsp:Policy>
     <sp:UsernameToken
      sp:IncludeToken="…/IncludeToken/AlwaysToRecipient" wsp:Optional="true" >
      <wsp:Policy>
        <sp:WssUsernameToken10/>
```

```
      </wsp:Policy>
     </sp:UsernameToken>
    </wsp:Policy>
</sp:SignedEncryptedSupportingTokens>
```

In the example, specifying the Username Token for authorization is optional. The client can continue if it cannot generate the Username Token because the user is anonymous or when there is no security context.

During the Security Policy enforcement process, the message is not rejected if the missing element has the Policy assertion with the attribute of `wsp:Optional="true"`.

The following security policy assertions are now supported by the `Optional` policy assertion:

- Username Token
- SAML Token
- Signature parts or signature elements
- Encryption parts or encryption elements
- Derive Key Token

## 2.20 Configuring Element-Level Security

WebLogic Server supports the element-level assertions defined in WS-SecurityPolicy 1.2. These assertions allow you to apply a signature or encryption to selected elements within the SOAP request or response message, enabling you to target only the specific data in the message that requires security and thereby reduce the computational requirements.

In addition, the assertion `RequiredElements` allows you to ensure that the message contains a specific header element.

The following element-level assertions are available:

- EncryptedElements (http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html#_Toc161826516)
- ContentEncryptedElements (http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html#_Toc161826517)
- SignedElements (http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html#_Toc161826513)
- RequiredElements (http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html#_Toc161826518)

In order to specify an element-level assertion, you must identify the particular request element or response element to which it applies.

You use XPath expressions in policy files to identify these elements, via either XPath Version 1.0 (http://www.w3.org/TR/xpath) or XPath Filter Version 2.0 (http://www.w3.org/TR/xmldsig-filter2/) syntax. The examples in this section use the default syntax, XPath Version 1.0.

Because each of these assertions identifies one or more particular elements in Web service message, you must use custom security policy files for all element-level security assertions. These custom policy files are typically combined with predefined security policy files, with the predefined files defining the way that signing or encryption is performed, and the custom policy files identifying the particular elements that are to be signed or encrypted.

### 2.20.1 Define and Use a Custom Element-Level Policy File

The first step is to determine the XPath expression that identifies the target element. To do this, you need to understand the format of the SOAP messages used by your web service, either through direct inspection or via analysis of the service's WSDL and XML Schema.

How you determine the format of the SOAP message, and therefore the required XPath expression, is heavily dependent on the tools you have available and is outside the scope of this document. For example, you might do the following:

1. Run the Web service without element-level security.
2. Turn on SOAP tracing.
3. Inspect the SOAP message in the logs.
4. Produce the XPath expression from the SOAP message.

Or, you might have a software tool that allows you to produce a sample SOAP request for a given WSDL, and then use it to generate the XPath expression.

Consider the example of a Web service that has a "submitOrderRequest" operation that will receive a SOAP request of the form shown in Example 2–15.

The sections in bold will be later used to construct the custom element-level policy.

***Example 2–15 submitOrderRequest SOAP Request***

```
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header/>
  <env:Body>
    <ns1:submitOrderRequest
            xmlns:ns1="http://www.oracle.com/OrderService">
      <ns1:OrderRequest>
        <ns1:orderNumber>4815162342</ns1:orderNumber>
        <ns1:creditCard>
          <ns1:cctype>MasterCard</ns1:cctype>
          <ns1:expires>12-01-2020</ns1:expires>
          <ns1:ccn>1234-567890-4444</ns1:ccn>
        </ns1:creditCard>
      </ns1:OrderRequest>
    </ns1:submitOrderRequest>
  </env:Body>
</env:Envelope>
```

Assume that you require that the **<ns1:creditCard>** element and its child elements be encrypted. To do this, you use the information obtained from the bold sections of Example 2–15 to create a custom security policy file, perhaps called `EncryptCreditCard.xml`.

Consider the example shown in Example 2–16.

***Example 2–16 EncryptCreditCard.xml Custom Policy File***

```
<?xml version="1.0"?>
<wsp:Policy
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512">
  <sp:EncryptedElements xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <sp:XPath xmlns:myns="http://www.oracle.com/OrderService">
/soapenv:Envelope/soapenv:Body/myns:submitOrderRequest/myns:OrderRequest/myns:cred
itCard
    </sp:XPath>
  </sp:EncryptedElements>
</wsp:Policy>
```

As described in the WS-SecurityPolicy 1.2 Specification (http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html#_Toc161826516), the `/sp:EncryptedElements/sp:XPath` element contains a string specifying an XPath expression that identifies the nodes to be confidentiality protected. The XPath expression is evaluated against the S:Envelope element node of the message. Multiple instances of this element may appear within this assertion and should be treated as separate references.

Note the following:

- The root element must be <wsp:Policy> with the prefix (in this case `wsp`) mapping to the full WS-Policy namespace.
- The assertion (in this case `EncryptedElements`) must also be namespace-qualified with the full WS-SecurityPolicy 1.2 namespace, as indicated by the "sp" prefix.
- The `creditCard` element in the SOAP message is namespace-qualified (via the `ns1` prefix), and has parent elements: `OrderRequest`, `submitOrderRequest`, `Body`, and `Envelope`. Each of these elements is namespace-qualified.

  The XPath query (beginning with `/soapenv:Envelope…`) matches the location of the `creditCard` element:

  ```
  /soapenv:Envelope/soapenv:Body/myns:submitOrderRequest/myns:OrderRequest/myns:c
  reditCard
  ```
- The namespace prefixes in the SOAP message need not match the prefixes in the custom security policy file. It is important only that the full namespaces to which the prefixes map are the same in both the message and policy assertion.
- WebLogic Server handles the mapping of SOAP 1.1 and SOAP 1.2 namespaces, and WS-Addressing 2004/08 and WS-Addressing 1.0 namespaces.

### 2.20.1.1 Adding the Policy Annotation to JWS File

After you have created your custom policy, add a Policy annotation to your JWS file so that the `ElementEncryption` policy is used for `submitOrder` web service requests, as shown in Example 2–17.

***Example 2–17 Adding Policy Annotation for Custom Policy File***

```
@WebMethod
@Policies({
  @Policy(uri="policy:Wssp1.2-2007-Wss1.1-UsernameToken-Plan-X509-Basic256.xml"),
  @Policy(uri="../policies/EncryptCreditCard.xml",
         direction=Policy.Direction.inbound)})
```

```
public String submitOrderRequest (OrderRequest orderRequest) {
    return orderService.processOrder(orderRequest);
}
```

Because the `creditCard` element is present in the SOAP request, but not the response, the code fragment configures the `EncryptedElements` custom policy only in the "inbound" direction.

### 2.20.2 Implementation Notes

Keep the following considerations in mind when implementing element-level security:

- You can include multiple element-level assertions in a policy; all are executed.
- You can include multiple <sp:XPath> expressions in a single assertions; all are executed.
- The `EncryptedElements` assertion causes the identified element and all of its children to be encrypted.
- The `ContentEncryptedElements` assertion does not encrypt the identified element, but does encrypt all of its children.
- The `RequiredElements` assertion may be used to test for the presence of a top-level element in the SOAP header. If the element is not found, a SOAP Fault will be raised.

  `RequiredElements` assertions cannot be used to test for elements in the SOAP Body.

## 2.21 Smart Policy Selection

Multiple policy alternatives for any given Web service are supported, which provides the service with significant flexibility.

Consider that a Web service might support any of the following:

- Different versions of the standard. For example, the Web service might allow WSRM 1.0 and WSRM 1.1, WSS1.0 and WSS 1.1, WSSC 1.1 and WWSSC 1.2, SAML 1.1 or SAML 2.0.
- Different credentials for authentication. For example, the Web service might allow either username token, X509, or SAML token for authentication.
- Different security requirements for internal and external clients. For example, external authentication might require a SAML token, while internal employee authentication requires only a username token for authentication.

The Web services client can also handle multiple policy alternatives. The same client can interoperate with different services that have different policy or policy alternatives.

For example, the same client can talk to one service that requires SAML 1.1 Token Profile 1.0 for authentication, while another service requires SAML 2.0 Token Profile 1.1 for authentication.

### 2.21.1 Example of Security Policy With Policy Alternatives

Example 2–18 shows an example of a security policy that supports both WS-Security 1.0 and WS-Security 1.1.

**Note:** Within the <wsp:ExactlyOne> element, each policy alternative is encapsulated within a <wsp:All> element.

***Example 2–18 Policy Defining Multiple Alternatives***

```
<?xml version="1.0"?>
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512">
<wsp:ExactlyOne>
  <wsp:All>
    <sp:AsymmetricBinding>
      <wsp:Policy>
        <sp:InitiatorToken>
          <wsp:Policy>
             <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/Include
Token/AlwaysToRecipient">
               <wsp:Policy>
                 <sp:WssX509V3Token10/>
               </wsp:Policy>
             </sp:X509Token>
          </wsp:Policy>
        </sp:InitiatorToken>
        <sp:RecipientToken>
          <wsp:Policy>
             <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/Include
Token/Never">
               <wsp:Policy>
                 <sp:WssX509V3Token10/>
               </wsp:Policy>
             </sp:X509Token>
          </wsp:Policy>
        </sp:RecipientToken>
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:Basic256/>
          </wsp:Policy>
        </sp:AlgorithmSuite>
        <sp:Layout>
          <wsp:Policy>
            <sp:Lax/>
          </wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp/>
        <sp:ProtectTokens/>
        <sp:OnlySignEntireHeadersAndBody/>
      </wsp:Policy>
    </sp:AsymmetricBinding>
    <sp:SignedParts>
      <sp:Body/>
    </sp:SignedParts>
    <sp:Wss10>
      <wsp:Policy>
        <sp:MustSupportRefKeyIdentifier/>
        <sp:MustSupportRefIssuerSerial/>
      </wsp:Policy>
    </sp:Wss10>
```

```
  </wsp:All>
  <wsp:All>
    <sp:AsymmetricBinding>
      <wsp:Policy>
        <sp:InitiatorToken>
          <wsp:Policy>
            <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/Include
Token/AlwaysToRecipient">
              <wsp:Policy>
                <sp:RequireThumbprintReference/>
                <sp:WssX509V3Token11/>
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
        </sp:InitiatorToken>
        <sp:RecipientToken>
          <wsp:Policy>
            <sp:X509Token

sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/Include
Token/Never">
              <wsp:Policy>
                <sp:RequireThumbprintReference/>
                <sp:WssX509V3Token11/>
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
        </sp:RecipientToken>
        <sp:AlgorithmSuite>
          <wsp:Policy>
            <sp:Basic256/>
          </wsp:Policy>
        </sp:AlgorithmSuite>
        <sp:Layout>
          <wsp:Policy>
            <sp:Lax/>
          </wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp/>
        <sp:ProtectTokens/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:AsymmetricBinding>
      <sp:SignedParts>
        <sp:Body/>
      </sp:SignedParts>
      <sp:Wss11>
        <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier/>
          <sp:MustSupportRefIssuerSerial/>
          <sp:MustSupportRefThumbprint/>
          <sp:MustSupportRefEncryptedKey/>
          <sp:RequireSignatureConfirmation/>
        </wsp:Policy>
      </sp:Wss11>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

## 2.21.2 Configuring Smart Policy Selection

You can configure multiple policy alternatives for a single Web service by creating a custom policy, as shown in Example 2–18. You then configure the Web service client to make a policy selection preference.

In this release of WebLogic Server, you can configure the policy selection preferences for the Web service client by using the WebLogic Server Administration Console, and via stubs.

The following preferences are supported:

- Security
- Performance
- Compatibility

### 2.21.2.1 How the Policy Preference is Determined

The Web services runtime uses your policy selection preference to examine the policy alternatives and select the best choice.

If there are multiple policy choices, the system uses the configured preference list, the availability of the credential, and setting of the optional function to determine the best selection policy.

If multiple policy alternatives exist for a client, the following selection rules are used:

- If the preference is not set, the first policy alternative will be picked, except if the policy alternative is defined as `wsp:optional=true`.
- If the preference is set to security first, then the policy that has the most security features is selected.
- If the preference is set to compatibility/interop first, then the policy that has the lowest version is selected.
- If the preference is set to performance first, then the policy with the fewest security features is selected.

For the optional policy assertions, the following selection rules are used:

- If the default policy selection preference is set, then the optional attribute on any assertion is ignored.
- If the Compatibility or Performance preference is set, then any assertion with an optional attribute is ignored; therefore the assertion is ignored.
- If the security policy selection preference is set, optional assertions are included and alternative assertions are never generated.

### 2.21.2.2 Configuring Smart Policy Selection in the Console

Perform the following steps to configure smart policy selection in the Console:

1. If you do not already have a functional Web services security configuration, create a Web services security configuration as described in the *Oracle WebLogic Server Administration Console Help*.
2. Edit the Web services security configuration. On the General tab, set the Policy Selection Preference. The following values are supported:

- None (default)
- Security then Compatibility then Performance (SCP)
- Security then Performance then Compatibility (SPC)
- Compatibility then Security then Performance (CSP)
- Compatibility then Performance then Security (CPS)
- Performance then Compatibility then Security (PCS)
- Performance then Security then Compatibility (PSC

3. Save and activate your changes.

### 2.21.2.3 Understanding Body Encryption in Smart Policy

In smart policy selection scenarios, whether or not the Body will be encrypted (for example, `<sp:EncryptedParts> <sp:Body /></sp:EncryptedParts>`) depends on the following policy selection preference rules:

- Default -- The first policy alternative will be used for the determination. If the encrypted body assertion is in the first policy alternative, the body is encrypted. If the encrypted body assertion is not in the first policy alternative, the body is not encrypted.
- SCP, SPC -- encrypted
- PCS, PSC -- not encrypted
- CPS -- not encrypted
- CSP -- encrypted

Consider the following two examples. In Example 2–19, the encrypted body assertion is in the first policy alternative. Therefore, in the default preference case the body is encrypted. For policy selection preferences other than the default, the other preference rules apply.

***Example 2–19 Body Assertion in First Policy Alternative***

```
<?xml version="1.0"?>
<wsp:Policy
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
>
<wsp:ExactlyOne>
<sp:EncryptedParts>
<sp:Body/>
</sp:EncryptedParts>
<sp:EncryptedParts/>
</wsp:ExactlyOne>
</wsp:Policy>
```

By contrast, in Example 2–20, the encrypted body assertion is not in the first policy alternative. Therefore, in the default preference case the body is not encrypted. For policy selection preferences other than the default, the other preference rules apply.

***Example 2–20 Body Assertion Not in First Policy Alternative***

```
<?xml version="1.0"?>
<wsp:Policy
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
```

```
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
>
<wsp:ExactlyOne>
<sp:EncryptedParts/>
<sp:EncryptedParts>
<sp:Body/>
</sp:EncryptedParts>
</wsp:ExactlyOne>
</wsp:Policy>
```

#### 2.21.2.4 Smart Policy Selection for a Standalone Client

You can set the policy selection preference via the stub property.

The following example sets the stub property for security, compatibility, and performance preferences:

```
stub._setProperty(WLStub.POLICY_SELECTION_PREFERENCE,
WLStub.PREFERENCE_SECURITY_COMPATIBILITY_PERFORMANCE);
```

If the policy selection preference is not set, then the default preference (None) is used.

### 2.21.3 Multiple Transport Assertions

If there are multiple available transport-level assertions in your security policies, WebLogic Server uses the policy that requires https. If more than one policy alternative requires https, WebLogic Server randomly picks one of them. You should therefore avoid using multiple policy alternatives that contain mixed transport-level policy assertions.

## 2.22 Example of Adding Security to MTOM Web Service

> **Note:** The example shows adding security to a JAX-RPC Web service. In this release, MTOM with WS-Security is supported for both JAX-WS and JAX-RPC.

As described in *Optimizing Binary Data Transmission Using MTOM/XOP*, SOAP Message Transmission Optimization Mechanism/XML-binary Optimized Packaging (MTOM/XOP) defines a method for optimizing the transmission of XML data of type `xs:base64Binary` or `xs:hexBinary` in SOAP messages.

This section describes a combination of two examples that are already included with WebLogic Server:

- *WL_HOME*`\samples\server\examples\src\examples\webservices\wss1.1`
- *WL_HOME*`\samples\server\examples\src\examples\webservices\mtom`

These existing examples include functional code and extensive `instructions.html` files that describes their use and function, how to build them, and so forth. This section does not repeat that information, but instead concentrates on the changes made to these examples, and the reasons for the changes.

## 2.22.1 Files Used by This Example

The example uses the files shown in Table 2–1. The contents of the source files are shown in subsequent sections.

***Table 2–10 Files Used in MTOM/Security Example***

| File | Description |
|---|---|
| build.xml | Ant build file that contains targets for building and running the example. |
| configWss.py | WLST script that configures a Web service security configuration. This file is copied without change from *WL_HOME*\samples\server\examples\src\examples\webservices\wss1.1 |
| MtomClient.java | Standalone client application that invokes the MTOM Web service. This file uses the JAX-RPC Stubs generated by clientgen, based on the WSDL of the Web service. |
| SecurityMtomService.java | JWS file that implements the MTOM Web service. The JWS file uses the @Policy annotation to specify the WS-Policy files that are associated with the Web service. |
| clientkeyStore.jks | Client-side key store, used to create a client-side BinarySecurityToken credential provider.<br>This file is copied without change from *WL_HOME*\samples\server\examples\src\examples\webservices\wss1.1\certs |
| serverkeyStore.jks | Server-side key store, used to create a Server-side BinarySecurityToken credential provider.<br>This file is copied without change from *WL_HOME*\samples\server\examples\src\examples\webservices\wss1.1\certs |
| testServerCertTempCert.der | Server-side certificate, used to create a client-side BinarySecurityToken credential provider.<br>This file is copied without change from *WL_HOME*\samples\server\examples\src\examples\webservices\wss1.1\certs |

## 2.22.2 SecurityMtomService.java

The `SecurityMtomService.java` JWS file is the same as that in *WL_HOME*`\samples\server\examples\src\examples\webservices\mtom\MtomService.java`, with the additional Policy annotations shown in bold.

***Example 2–21 SecurityMtomService.java***

```
package examples.webservices.security_mtom;
import weblogic.jws.Binding;
import weblogic.jws.Policy;
import weblogic.jws.Policies;
import weblogic.jws.Context;
import weblogic.jws.WLDeployment;
import weblogic.wsee.jws.JwsContext;
import weblogic.wsee.mtom.api.MtomPolicyInfo;
import weblogic.wsee.mtom.api.MtomPolicyInfoFactory;
import weblogic.wsee.policy.framework.PolicyException;

import javax.jws.WebService;
import javax.jws.WebMethod;
```

```
import java.rmi.RemoteException;

/**
 * Sample to MTOM with JAX-RPC
 *
 * @author Copyright © 1996, 2008, Oracle and/or its affiliates.
 * All rights reserved.
 */
@WebService
@Binding(Binding.Type.SOAP12)
//enable WSS + MTOM for this web service by adding the following canned policy
files
@Policies({
    @Policy(uri = "policy:Mtom.xml"),
    @Policy(uri = "policy:Wssp1.2-2007-SignBody.xml"),
    @Policy(uri = "policy:Wssp1.2-2007-EncryptBody.xml"),
@Policy(uri = "policy:Wssp1.2-Wss1.1-EncryptedKey.xml")
})
public class SecurityMtomService {

  public SecurityMtomService() {

  }

  /**
   * Input is sent as XOP'ed binary octet stream
   *
   * @param bytes input bytes
   * @return A simple String
   */
  @WebMethod
  public String echoBinaryAsString(byte[] bytes) {
    return new String(bytes);
  }

  /**
   * Output is sent as as XOP'ed binary octet stream
   *
   * @param s a simple String
   * @return byte[]
   */
  @WebMethod
  public byte[] echoStringAsBinary(String s) {
    return s.getBytes();
  }

  /**
   * input byte[] is sent as as XOP'ed binary octet stream
   *
   * @param array input byte[] array
   * @return String[]
   */
  @WebMethod
  public String[] echoBinaryArrayAsStringArray(byte[] array) {
    String[] strings = new String[1];
    strings[0] = new String(array);
    return strings;
  }
}
```

You can specify the @Policy annotation at both the class- and method- level. In this example, the annotation is used at the class-level to specify the predefined WS-Policy files, which means all public operations of the Web service are associated with the specified WS-Policy files.

You use the @Policies annotation to group together multiple @Policy annotations. You can specify this annotation at both the class- and method-level. In this example, the annotation is used at the class-level to group the four @Policy annotations that specify the predefined WS-Policy files:

- The predefined WS-Policy file `Mtom.xml` enables MTOM encoding.
- As described in Section 2.16.2, "Protection Assertion Policies", the `Wssp1.2-2007-SignBody.xml` policy file specifies that the body and WebLogic system headers of both the request and response SOAP message be digitally signed.
- The `Wssp1.2-2007-EncryptBody.xml` policy file specifies that the body of both the request and response SOAP messages be encrypted.
- The `Wssp1.2-Wss1.1-EncryptedKey.xml` symmetric binding policy uses the WS-Security 1.1 Encrypted Key feature. The client application invoking the Web service must use the encrypted key to encrypt and sign, and the server must send Signature Confirmation.

### 2.22.3 MtomClient.java

`MtomClient.java` is a standalone client application that invokes the `SecurityMtomService` Web service. It uses the JAX-RPC stubs generated by clientgen, based on the WSDL of the Web service. The MtomClient code is shown in Example 2–22.

***Example 2–22 MtomClient.java***

```
package examples.webservices.security_mtom.client;

import java.rmi.RemoteException;

import java.security.cert.X509Certificate;
import java.util.ArrayList;
import java.util.List;
import javax.xml.rpc.Stub;

import weblogic.security.SSL.TrustManager;

// Import classes to create the Binary and Username tokens
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;

// Import classes for creating the client-side credential provider
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.wsee.security.util.CertUtils;

/**
 * @author Copyright © 1996, 2008, Oracle and/or its affiliates.
 * All rights reserved.
 */
public class MtomClient {
  private static final String FOO = "FOO";
```

```
  private static SecurityMtomService port;

  public MtomClient(String args[]) throws Exception {
    //client keystore file
    String clientKeyStore = args[0];
    String clientKeyStorePass = args[1];
    String clientKeyAlias = args[2];
    String clientKeyPass = args[3];

    //server certificate
    String serverCertFile = args[4];
    String wsdl = args[5];

    SecurityMtomServiceService service = new SecurityMtomServiceService_
Impl(wsdl);
    port = service.getSecurityMtomServiceSoapPort();

X509Certificate serverCert = (X509Certificate)
CertUtils.getCertificate(serverCertFile);

    //create emtpy list of credential providers
    List credProviders = new ArrayList();

    //Create client-side BinarySecurityToken credential provider that uses
    // X.509 for identity, based on certificate and keys parameters
    CredentialProvider cp = new ClientBSTCredentialProvider(clientKeyStore,
 clientKeyStorePass, clientKeyAlias, clientKeyPass, "JKS", serverCert);
    credProviders.add(cp);

    Stub stub = (Stub) port;

    // Set stub property to point to list of credential providers
    stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);

    // setup the TrustManager.
    stub._setProperty(WSSecurityContext.TRUST_MANAGER,
        new TrustManager() {
          public boolean certificateCallback(X509Certificate[] chain, int
validateErr) {
            //Typically in a real-life application, Java code that actually
            //verifies the certificate goes here; for sake of simplicity, this
            //example assumes the certificate is valid and simply returns true.

            return true;
          }
        });
  }

  public static void main(String[] args) throws Exception {
    MtomClient client = new MtomClient(args);
    client.invokeEchoBinaryAsString();
    client.invokeEchoStringAsBinary();
    client.invokeEchoBinaryArrayAsStringArray();
  }

  public void invokeEchoBinaryArrayAsStringArray() throws RemoteException {
    System.out.println("sending a String '" + FOO + "' as a byte array.");
    String result =
port.echoBinaryArrayAsStringArray(FOO.getBytes()).getJavaLangstring()[0];
    System.out.println("echoing '" + result + "' as a String array.");
```

```
  }

  public void invokeEchoStringAsBinary() throws RemoteException {
    System.out.println("sending a String '" + FOO + "'");
    String result = new String(port.echoStringAsBinary(FOO));
    System.out.println("echoing '" + result + "' as a byte array.");
  }

  public void invokeEchoBinaryAsString() throws RemoteException {
    System.out.println("sending a String '" + FOO + "' as a byte array.");
    String result = port.echoBinaryAsString(FOO.getBytes());
    System.out.println("echoing '" + result + "' as a String.");
  }
}
```

The client application takes six arguments:

- Client keystore
- Client keystore password
- Client key alias
- Client key password
- The server certificate file
- WSDL of the deployed Web service

The client application uses the following WebLogic Web services security APIs to create the needed client-side credential providers, as specified by the WS-Policy files that are associated with the Web service:

- weblogic.wsee.security.bst.ClientBSTCredentialProvider to create a binary security token credential provider, using the certificate and private key.
- weblogic.xml.crypto.wss.WSSecurityContext to specify the list of credential providers to the JAX-RPC stub.
- weblogic.xml.crypto.wss.provider.CredentialProvider, which is the main credential provider class.

When you write this client application, you need to consult the WS-Policy files associated with a Web service to determine the types and number of credential providers that must be set in the JAX-RPC stub. Typically, if the WS-Policy file specifies that SOAP messages must be signed or encrypted, using X.509 for identity, then you must create a ClientBSTCredentialProvider. (If it specifies that the user provides a username token for identity, then the application must create a ClientUNTCredentialProvider.)

The example creates a client BST credential provider for the indicated keystore, certificate alias, and server certificate. The certificate passed for the parameter `serverCert` is used to encrypt the message body contents and to verify the received signature. Any KeyInfo received as part of the in-bound signature (for example, certificate thumbprint) must correctly identify the same server certificate.

The Web services client runtime also consults this WSDL so it can correctly create the security headers in the SOAP request when an operation is invoked.

Finally, the client application must use the weblogic.security.SSL.TrustManager WebLogic security API to verify that the certificate used to encrypt the SOAP request is valid. The client runtime gets this certificate (`serverCert` in the example) from the deployed WSDL of the Web service, which in real-life situations is not automatically

trusted, so the client application must ensure that it is okay before it uses it to encrypt the SOAP request.

> **Note:** The client-side certificate and private key used in this example have been created for simple testing purposes, and therefore are always trusted by WebLogic Server. For this reason, there is no additional server-side security configuration needed to run this example. In real life, however, the client application would use a certificate from a real certificate authority, such as Verisign. In this case, administrators would need to use the WebLogic Administration Console to add this certificate to the list that is trusted by WebLogic Server.

## 2.22.4 configWss.py Script File

The SecurityMtomService Web service does not explicitly invoke any WebLogic Server API to handle the requirements imposed by any associated policy files, nor does this Web service have to understand which, if any, security providers, tokens, or other such mechanisms are involved.

The script file `configWss.py` uses WLST to create and configure the default Web service security configuration, `default_wss`, for the active security realm. (The default Web service security configuration is used by *all* Web services in the domain unless they have been explicitly programmed to use a different configuration.) Further, this script makes sure that x509 tokens are supported, creates the needed security providers, and so forth.

Example 2–23 shows the `configWss.py` file. The `build.xml` file provides the command input. Sections of particular interest are shown in bold.

> **Note:** Long lines in this script have been formatted for readability.

***Example 2–23 configWss.py***

```
userName = sys.argv[1]
passWord = sys.argv[2]
url="t3://"+sys.argv[3]+":"+sys.argv[4]

print "Connect to the running adminSever"

connect(userName, passWord, url)

edit()
startEdit()

#Enable assert x509 in SecurityConfiguration
rlm = cmo.getSecurityConfiguration().getDefaultRealm()
ia = rlm.lookupAuthenticationProvider("DefaultIdentityAsserter")
activeTypesValue = list(ia.getActiveTypes())
existed = "X.509" in activeTypesValue
if existed == 1:
  print 'assert x509 is aleady enabled'
else:
  activeTypesValue.append("X.509")
ia.setActiveTypes(array(activeTypesValue,java.lang.String))
ia.setDefaultUserNameMapperAttributeType('CN');
ia.setUseDefaultUserNameMapper(Boolean('true'));
```

```
#Create default WebServcieSecurity
securityName='default_wss'
defaultWss=cmo.lookupWebserviceSecurity(securityName)
if defaultWss == None:
  print 'creating new webservice security bean for: ' + securityName
  defaultWss = cmo.createWebserviceSecurity(securityName)
else:
  print 'found exsiting bean for: ' + securityName


#Create credential provider for DK
cpName='default_dk_cp'
wtm=defaultWss.lookupWebserviceCredentialProvider(cpName)
if wtm == None:
        wtm = defaultWss.createWebserviceCredentialProvider(cpName)
       wtm.setClassName(‘weblogic.wsee.security.wssc.v200502.dk.
            DKCredentialProvider‘)
        wtm.setTokenType('dk')
        cpm = wtm.createConfigurationProperty('Label')
        cpm.setValue('WS-SecureConversationWS-SecureConversation')
        cpm = wtm.createConfigurationProperty('Length')
        cpm.setValue('16')
else:
  print 'found exsiting bean for: DK ' + cpName

#Create credential provider for x.509
cpName='default_x509_cp'
wtm=defaultWss.lookupWebserviceCredentialProvider(cpName)
if wtm == None:
        wtm = defaultWss.createWebserviceCredentialProvider(cpName)
        wtm.setClassName('weblogic.wsee.security.bst.
            ServerBSTCredentialProvider')
        wtm.setTokenType('x509')
else:
  print 'found exsiting bean for: x.509 ' + cpName

#Custom keystore for xml encryption
cpName='ConfidentialityKeyStore'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty(cpName)
keyStoreName=sys.argv[5]
cpm.setValue(keyStoreName)

cpName='ConfidentialityKeyStorePassword'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty(cpName)
cpm.setEncryptValueRequired(Boolean('true'))
KeyStorePasswd=sys.argv[6]
cpm.setEncryptedValue(KeyStorePasswd)

cpName='ConfidentialityKeyAlias'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty(cpName)
keyAlias=sys.argv[7]
cpm.setValue(keyAlias)
```

```
cpName='ConfidentialityKeyPassword'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty('ConfidentialityKeyPassword')
cpm.setEncryptValueRequired(Boolean('true'))
keyPass=sys.argv[8]
cpm.setEncryptedValue(keyPass)

#Custom keystore for xml digital signature
cpName='IntegrityKeyStore'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty(cpName)
keyStoreName=sys.argv[5]
cpm.setValue(keyStoreName)

cpName='IntegrityKeyStorePassword'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty(cpName)
cpm.setEncryptValueRequired(Boolean('true'))
KeyStorePasswd=sys.argv[6]
cpm.setEncryptedValue(KeyStorePasswd)

cpName='IntegrityKeyAlias'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty(cpName)
keyAlias=sys.argv[7]
cpm.setValue(keyAlias)

cpName='IntegrityKeyPassword'
cpm=wtm.lookupConfigurationProperty(cpName)
if cpm == None:
        cpm = wtm.createConfigurationProperty(cpName)
cpm.setEncryptValueRequired(Boolean('true'))
keyPass=sys.argv[8]
cpm.setEncryptedValue(keyPass)

#Create token handler for x509 token
#cpName='default_x509_handler'
th=defaultWss.lookupWebserviceTokenHandler(cpName)
if th == None:
        th = defaultWss.createWebserviceTokenHandler(cpName)
        th.setClassName('weblogic.xml.crypto.wss.BinarySecurityTokenHandler')
        th.setTokenType('x509')
        cpm = th.createConfigurationProperty('UseX509ForIdentity')
        cpm.setValue('true')

save()
activate(block="true")
disconnect()
exit()
```

## 2.22.5 Build.xml File

The build.xml file has the targets shown in Table 2–1.

***Table 2–11 build.xml targets***

| Target | Description |
|---|---|
| client | Target that builds the Security MTOM Web service client. |
| config.server.security | Target that configures the Web service security. |
| deploy | Target that deploys the Web service. |
| server | Target that builds the Security MTOM Web service. |
| clean | Deletes temporary directories. |
| build | Depends on server, client, and clean. |
| run | Target that runs the Security MTOM Web service client. |
| all | Default target. Depends on build, deploy. |

The complete `build.xml` file is shown in Example 2–24.

***Example 2–24 build.xml File***

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="webservices.security_mtom" default="all" basedir=".">

  <!-- set global properties for this build -->
  <property file="../../../examples.properties"/>

  <property name="client.dir"
value="${client.classes.dir}/webservicesSecurityMtom_Client" />
  <property name="package.dir" value="examples/webservices/security_mtom"/>
  <property name="package" value="examples.webservices.security_mtom"/>
  <property name="ws.file" value="SecurityMtomService" />
  <property name="ear.dir"
value="${examples.build.dir}/webservicesSecurityMtomEar" />
  <property name="cert.dir" value="${basedir}/certs" />
  <property name="certs.dir" value="${basedir}/certs" />

  <!--client keystore-->
  <property name="client-keystore-name" value="clientKeyStore.jks"/>
  <property name="client-keystore-pass" value="keystorepw"/>
  <property name="client-cert" value="ClientCert"/>
  <property name="client-key" value="ClientKey"/>
  <property name="client-key-pass" value="ClientKeyPass"/>
  <property name="client-cert-alias" value="testClientCert"/>

  <!--server keystore-->
  <property name="server-keystore-name" value="serverKeyStore.jks"/>
  <property name="server-keystore-pass" value="keystorepw"/>
  <property name="server-cert" value="ServerCert"/>
  <property name="server-key" value="ServerKey"/>
  <property name="server-key-pass" value="ServerKeyPass"/>
  <property name="server-cert-alias" value="testServerCert"/>

  <path id="client.class.path">
    <pathelement path="${client.dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>

  <!-- Web Service WLS Ant task definitions -->
```

```
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
      classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

  <target name="all" depends="build, deploy"/>

  <target name="build" depends="clean,server,client"/>

  <target name="clean">
    <delete dir="${ear.dir}"/>
    <delete dir="${client.dir}"/>
  </target>

  <!-- Target that builds the MTOM Web Service -->
  <target name="server" description="Target that builds the MTOM Web Service">
    <jwsc
      srcdir="${examples.src.dir}/${package.dir}"
      sourcepath="${examples.src.dir}"
      destdir="${ear.dir}"
      classpath="${java.class.path}"
      fork="true"
      keepGenerated="true"
      deprecation="${deprecation}"
      debug="${debug}">
        <jws file="SecurityMtomService.java" explode="true"/>
    </jwsc>
  </target>

  <!-- Target that builds the MTOM Web Service client -->
  <target name="client" description="Target that builds the source Web Service">
    <mkdir dir="${client.dir}/${package.dir}/client/"/>
    <clientgen
      wsdl="${ear.dir}/${ws.file}/WEB-INF/${ws.file}Service.wsdl"
      destDir="${client.dir}"
      classpath="${java.class.path}"
      packageName="${package}.client"/>
    <copy file="MtomClient.java" todir="${client.dir}/${package.dir}/client/"/>
    <javac
      srcdir="${client.dir}" destdir="${client.dir}"
      classpath="${java.class.path}"
      includes="${package.dir}/client/**/*.java"/>
  </target>

  <!-- Target that deploys the MTOM Web Service -->
  <target name="deploy" description="Target that deploys the reliable destination
Web Service">
    <wldeploy
      action="deploy"
      source="${ear.dir}"
      user="${wls.username}"
      password="${wls.password}"
      verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}"
      failonerror="${failondeploy}"/>
  </target>

  <!-- Target that runs the MTOM Web Service client -->
  <target name="run" >
```

```
    <java fork="true"
      classname="examples.webservices.security_mtom.client.MtomClient"
      failonerror="true" >
  <jvmarg line="-Dweblogic.wsee.verbose=*"/>
      <classpath refid="client.class.path"/>
      <arg line="
      ${basedir}/certs/${client-keystore-name}
      ${client-keystore-pass}
      ${client-cert-alias}
      ${client-key-pass}
      ${basedir}/certs/testServerCertTempCert.der

http://${wls.hostname}:${wls.port}/SecurityMtomService/SecurityMtomService?WSDL"
/>
    </java>
  </target>

    <!-- Target the configure the web service security -->
  <target name="config.server.security" description="Target the configure the web
service security">
    <copy todir="${examples.domain.dir}" overwrite="true">
      <fileset dir="${certs.dir}" includes="${server-keystore-name}"/>
    </copy>

    <java classname="weblogic.WLST" fork="true" failonerror="true">
      <arg line="configWss.py ${wls.username} ${wls.password} ${wls.hostname}
${wls.port}
      ${server-keystore-name} ${server-keystore-pass} ${server-cert-alias}
${server-key-pass}" />
    </java>

  </target>

</project>
```

## 2.22.6 Building and Running the Example

Follow these steps to build and run the example:

1. Start the Examples server.
2. Set up your environment, as described in the *MW_HOME*\WL_HOME\samples\server\examples\src\examples\examples.html instructions file.

   *MW_HOME**WL_HOME*\samples\domains\wl_server>setExamplesEnv.cmd
3. Change to the *MW_HOME**WL_HOME*\samples\server\examples\src\examples\webservices directory and create a new subdirectory called security_mtom.
4. Cut and paste the contents of the build.xml, configWss.py, MtomClient.java, and SecurityMtomService.java sections to files with the same names in the *MW_HOME**WL_HOME*\samples\server\examples\src\examples\webservices\security_mtom directory.
5. Copy all of the files (clientKeyStore.jks, serverKeyStore.jks, and testServerCertTempCert.der) from

*MW_HOME**WL_HOME*\samples\server\examples\src\examples\webservices\wss1.1\certs

to a new `certs` subdirectory

*MW_HOME**WL_HOME*\samples\server\examples\src\examples\webservices\security_mtom\certs

6. Change to the *MW_HOME**WL_HOME*\samples\server\examples\src\examples\webservices\security_mtom directory.
7. Execute the following command:

```
prompt> ant config.server.security
```

8. Restart Weblogic Server.
9. Build, deploy and run the example:

```
prompt> ant build deploy run
```

## 2.22.7 Deployed WSDL for SecurityMtomService

The deployed WSDL for the SecurityMtomService Web service is available at the following URL:

```
http://host:port/SecurityMtomService/SecurityMtomService?WSDL
```

The complete WSDL is shown in Example 2–25.

***Example 2–25 Deployed WSDL for SecurityMtomService***

```
<?xml version="1.0" encoding="UTF-8" ?>
  <s1:definitions name="SecurityMtomServiceServiceDefinitions"
targetNamespace="http://examples/webservices/security_mtom" xmlns=""
xmlns:s0="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utili
ty-1.0.xsd"
xmlns:s1="http://schemas.xmlsoap.org/wsdl/"
xmlns:s2="http://examples/webservices/security_mtom"
xmlns:s3="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:UsingPolicy s1:Required="true" />
  <wsp:Policy s0:Id="Mtom.xml">
  <wsoma:OptimizedMimeSerialization
xmlns:wsoma="http://schemas.xmlsoap.org/ws/2004/09/policy/optimizedmimeserializati
on" />
  </wsp:Policy>
  <wsp:Policy s0:Id="Wssp1.2-Wss1.1-EncryptedKey.xml">
  <sp:SymmetricBinding
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512">
  <wsp:Policy>
  <sp:ProtectionToken>
  <wsp:Policy>
  <sp:X509Token
sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/Include
Token/Never">
  <wsp:Policy>
  <sp:RequireThumbprintReference />
  <sp:WssX509V3Token11 />
  </wsp:Policy>
```

```
  </sp:X509Token>
  </wsp:Policy>
  </sp:ProtectionToken>
  <sp:AlgorithmSuite>
  <wsp:Policy>
  <sp:Basic256 />
  </wsp:Policy>
  </sp:AlgorithmSuite>
  <sp:Layout>
  <wsp:Policy>
  <sp:Lax />
  </wsp:Policy>
  </sp:Layout>
  <sp:IncludeTimestamp />
  <sp:OnlySignEntireHeadersAndBody />
  </wsp:Policy>
  </sp:SymmetricBinding>
  <sp:Wss11 xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512">
  <wsp:Policy>
  <sp:MustSupportRefKeyIdentifier />
  <sp:MustSupportRefIssuerSerial />
  <sp:MustSupportRefThumbprint />
  <sp:MustSupportRefEncryptedKey />
  <sp:RequireSignatureConfirmation />
  </wsp:Policy>
  </sp:Wss11>
  </wsp:Policy>
  <wsp:Policy s0:Id="Wssp1.2-2007-EncryptBody.xml">
  <sp:EncryptedParts
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <sp:Body />
  </sp:EncryptedParts>
  </wsp:Policy>
  <wsp:Policy s0:Id="Wssp1.2-2007-SignBody.xml">
  <sp:SignedParts
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <sp:Body />
  </sp:SignedParts>
  </wsp:Policy>
  <s1:types>
  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
targetNamespace="java:examples.webservices.security_mtom"
xmlns:s0="http://schemas.xmlsoap.org/wsdl/"
xmlns:s1="http://examples/webservices/security_mtom"
xmlns:s2="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="ArrayOfJavaLangstring_literal">
  <xs:sequence>
  <xs:element maxOccurs="unbounded" minOccurs="0" name="JavaLangstring"
nillable="true" type="xs:string" />
  </xs:sequence>
  </xs:complexType>
  <xs:element name="ArrayOfJavaLangstring_literal"
type="java:ArrayOfJavaLangstring_literal"
xmlns:java="java:examples.webservices.security_mtom" />
  <xs:element name="base64Binary_literal" type="xs:base64Binary" />
  </xs:schema>
  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
targetNamespace="http://examples/webservices/security_mtom"
```

```
xmlns:s0="http://schemas.xmlsoap.org/wsdl/"
xmlns:s1="http://examples/webservices/security_mtom"
xmlns:s2="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:import namespace="java:examples.webservices.security_mtom" />
  <xs:element name="echoBinaryAsString">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="bytes" type="xs:base64Binary" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="echoBinaryAsStringResponse">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="return" type="xs:string" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="echoBinaryArrayAsStringArray">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="array" type="xs:base64Binary" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="echoBinaryArrayAsStringArrayResponse">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="return" type="java:ArrayOfJavaLangstring_literal"
xmlns:java="java:examples.webservices.security_mtom" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="echoStringAsBinary">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="s" type="xs:string" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="echoStringAsBinaryResponse">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="return" type="xs:base64Binary" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  </xs:schema>
  </s1:types>
  <s1:message name="echoBinaryAsString">
  <s1:part element="s2:echoBinaryAsString" name="parameters" />
  </s1:message>
  <s1:message name="echoBinaryAsStringResponse">
  <s1:part element="s2:echoBinaryAsStringResponse" name="parameters" />
  </s1:message>
  <s1:message name="echoBinaryArrayAsStringArray">
  <s1:part element="s2:echoBinaryArrayAsStringArray" name="parameters" />
  </s1:message>
```

```
  <s1:message name="echoBinaryArrayAsStringArrayResponse">
  <s1:part element="s2:echoBinaryArrayAsStringArrayResponse" name="parameters" />
  </s1:message>
  <s1:message name="echoStringAsBinary">
  <s1:part element="s2:echoStringAsBinary" name="parameters" />
  </s1:message>
  <s1:message name="echoStringAsBinaryResponse">
  <s1:part element="s2:echoStringAsBinaryResponse" name="parameters" />
  </s1:message>
  <s1:portType name="SecurityMtomService"
wsp:PolicyURIs="#Wssp1.2-2007-SignBody.xml #Wssp1.2-2007-EncryptBody.xml
#Wssp1.2-Wss1.1-EncryptedKey.xml">
  <s1:operation name="echoBinaryAsString" parameterOrder="parameters">
  <s1:input message="s2:echoBinaryAsString" />
  <s1:output message="s2:echoBinaryAsStringResponse" />
  </s1:operation>
  <s1:operation name="echoBinaryArrayAsStringArray" parameterOrder="parameters">
  <s1:input message="s2:echoBinaryArrayAsStringArray" />
  <s1:output message="s2:echoBinaryArrayAsStringArrayResponse" />
  </s1:operation>
  <s1:operation name="echoStringAsBinary" parameterOrder="parameters">
  <s1:input message="s2:echoStringAsBinary" />
  <s1:output message="s2:echoStringAsBinaryResponse" />
  </s1:operation>
  </s1:portType>
  <s1:binding name="SecurityMtomServiceServiceSoapBinding"
type="s2:SecurityMtomService">
  <s3:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
  <wsp:Policy>
  <wsp:PolicyReference URI="#Mtom.xml" />
  </wsp:Policy>
  <s1:operation name="echoBinaryAsString">
  <s3:operation style="document" />
  <s1:input>
  <s3:body parts="parameters" use="literal" />
  </s1:input>
  <s1:output>
  <s3:body parts="parameters" use="literal" />
  </s1:output>
  </s1:operation>
  <s1:operation name="echoBinaryArrayAsStringArray">
  <s3:operation style="document" />
  <s1:input>
  <s3:body parts="parameters" use="literal" />
  </s1:input>
  <s1:output>
  <s3:body parts="parameters" use="literal" />
  </s1:output>
  </s1:operation>
  <s1:operation name="echoStringAsBinary">
  <s3:operation style="document" />
  <s1:input>
  <s3:body parts="parameters" use="literal" />
  </s1:input>
  <s1:output>
  <s3:body parts="parameters" use="literal" />
  </s1:output>
  </s1:operation>
  </s1:binding>
  <s1:service name="SecurityMtomServiceService">
```

```
  <s1:port binding="s2:SecurityMtomServiceServiceSoapBinding"
name="SecurityMtomServiceSoapPort">
  <s3:address
location="http://localhost:7001/SecurityMtomService/SecurityMtomService" />
  </s1:port>
  </s1:service>
  </s1:definitions>
```

## 2.23 Example of Adding Security to Reliable Messaging Web Service

This section describes an update to an example that is already included with WebLogic Server:

- *WL_HOME*\samples\server\examples\src\examples\webservices\wsrm_security

This section shows how to update the example to use the most recent version of the policy file. Oracle recommends that you use the new policy namespace, as shown in the revised example, as those are official namespaces from OASIS standards and they will perform better when interoperating with other vendors.

### 2.23.1 Overview of Secure and Reliable SOAP Messaging

Reliable SOAP messaging is a framework whereby an application running in one WebLogic Server instance can reliably invoke a Web service running on another WebLogic Server instance. Reliable is defined as the ability to guarantee message delivery between the two Web services.

WebLogic Web services conform to the *WS-ReliableMessaging 1.1* specification, which describes how two Web services running on different WebLogic Server application servers can communicate reliably in the presence of failures in software components, systems, or networks. In particular, the specification describes an interoperable protocol in which a message sent from a source endpoint (client Web service) to a destination endpoint (Web service whose operations can be invoked reliably) is guaranteed either to be delivered, according to one or more delivery assurances, or to raise an error. The WS-ReliableMessaging specification defines an interoperable way to provide security by composing WS-ReliableMessaging with WS-SecureConversation and associating a reliable sequence with a secure session. At sequence creation time, the sending side needs to present a Security Token Reference to point to a Security Context Token that will be used to identify the owner of the sequence. All subsequent sequence messages and protocol messages in both directions will need to demonstrate proof-of-possession of the referenced key.

WebLogic reliable SOAP messaging works only between two Web services. This means that you can invoke a WebLogic Web service reliably only from another Web service, and not from a standalone client application. This example shows how to create both types of Web services (source and destination). The `WsrmSecurityClient.java` class is a standalone Java application that then invokes the source Web service.

### 2.23.2 Overview of the Example

The existing example shows how to provide security functionality on top of reliability for Web services messaging by creating two WebLogic Web services:

- Web service whose operations can be invoked using reliable and secure SOAP messaging (destination endpoint). The destination `ReliableEchoService` Web

service has two operations that can be invoked reliably and in a secure way: `echo` and `echoOneway`.

- Client Web service that invokes an operation of the first Web service in a reliable and secure way (source endpoint). The source `ReliableEchoClientService` Web service has one operation for invoking the `echo` and `echoOneway` operations of the `ReliableEchoService` Web service reliably and in a secure way within one conversation: `echo`.

The existing example includes functional code and an extensive `instructions.html` file that describes its use and function, how to build it, and so forth This section does not repeat that information, but instead concentrates on the changes made to the example, and the reasons for the changes.

#### 2.23.2.1 How the Example Sets Up WebLogic Security

The `configWSS.py` WLST script sets up security for the WebLogic Server instance that hosts the source and destination Web service. The security requirements are dictated by the WS-SecurityPolicy files associated with the destination Web service.

The `Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.0.xml` policy imposes the following requirements:

- WS-SecureConversation handshake is protected by WS-Security 1.0.
- The application messages are signed and encrypted with DerivedKeys.
- The soap:Body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages (part of the WS-SecureConversation handshake) are both signed and encrypted.
- The WS-Addressing headers are signed.
- Timestamp is included and signed.
- The signature is encrypted.
- The algorithm suite is Basic256.

In response, the `configWSS.py` WLST script performs the following functions:

- Enables X.509 tokens for the default IdentityAsserter in the default security realm.
- Creates the default Web service security configuration.
- Configures a credential provider for the Security Context Token.
- Configures a credential provider for Derived Key.
- Configures a BinarySecurityTokenHandler token handler for X.509 tokens.
- Configures a ServerBSTCredentialProvider credential provider for X.509 tokens.
- Configures keystores for confidentiality and integrity.
- Configures the PKI credential mapper. This maps the initiator and target resource to a key pair or public certificate

In addition, the `configWSSRuntime.py` WLST script also performs the following function:

- Sets up the PKI credential mapper (configured by `configWSS.py`) to invoke the destination Web service.

### 2.23.3 Files Used by This Example

The example uses the files shown in Table 2–1. The contents of revised source files are shown in subsequent sections.

***Table 2–12 Files Used in WSRM/Security Example***

| File | Description |
|---|---|
| build.xml | Ant build file that contains targets for building and running the example. |
| ReliableEchoClientServiceImpl.java | JWS file that implements the source Web service that reliably invokes the echoOneWay and echo operation of the ReliableEchoService Web service in a secure way. This JWS file uses the @ServiceClient annotation to specify the Web service it invokes reliably. |
| ReliableEchoServiceImpl.java | JWS file that implements the reliable destination Web service. This JWS file uses the @Policy annotation to specify a WS-Policy file that contains reliable SOAP messaging assertions. |
| ws_rm_configuration.py | WLST script that configures a SAF Agent, FileStore, JMS Server, and JMS queue, which are required for reliable SOAP messaging. Execute this script for the WebLogic Server instance that hosts the reliable destination Web service. The out-of-the-box Examples server has already been configured for the source Web service that invokes an operation reliably. |
| configWss.py | WLST script that configures a credential provider for Security Context Token, a credential provider for Derived Key, a credential provider for x.509, KeyStores for Confidentiality and Integrity, and PKI Cred Mapper that are required for secure SOAP messaging. Execute this script for the WebLogic Server instance that hosts the source and destination Web service. Remember to restart the Weblogic server after executing this script |
| configWss_Service.py | WLST script that configures a credential provider for Security Context Token, a credential provider for Derived Key, a credential provider for x.509, KeyStores for Confidentiality and Integrity that are required by the server host the destination Web service for secure SOAP messaging. Execute this script for the WebLogic Server instance that hosts the destination Web service when the source and destination Web service are hosted in two servers. Remember to restart the Weblogic server after executing this script. |
| configWssRuntime.py | WLST script that configures a KeyPair Credential for invoking the destination Web service. |
| certs/testServerCertTempCert.der | Server-side certificate, used create client-side BinarySecurityToken credential provider. |
| certs/clientKeyStore.jks | Client-side key store, used to create client-side BinarySecurityToken credential provider. |
| certs/serverKeyStore.jks | Server-side key store, used to create Server-side BinarySecurityToken credential provider. |
| WsrmSecurityClient.java | Standalone Java client application that invokes the source WebLogic Web service, that in turn invokes an operation of the ReliableEchoService Web service in a reliable and secure way. |

## 2.23.4 Revised ReliableEchoServiceImpl.java

The `ReliableEchoServiceImpl.java` JWS file is the same as that in *WL_HOME*`\samples\server\examples\src\examples\webservices\wsrm_security\ReliableEchoServiceImpl.java`, with the revised Policy annotation shown in bold.

***Example 2–26 ReliableEchoServiceImpl.java***

```
@WebService(name = "ReliableEchoPort",
    serviceName = "ReliableEchoService")
@WLHttpTransport(contextPath = "WsrmSecurity", serviceUri = "ReliableEchoService")
@Policies({
  @Policy(uri="policy:Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.0.xml"),
  @Policy(uri="policy:Reliability1.1_SequenceSTR")}
)
```

You can specify the @Policy annotation at both the class- and method- level. In this example, the annotation is used at the class-level to specify the predefined WS-Policy files, which means all public operations of the Web service are associated with the specified WS-Policy files.

## 2.23.5 Revised configWss.py

The ReliableEchoServiceImpl Web service does not explicitly invoke any WebLogic Server API to handle the requirements imposed by any associated policy files, nor does this Web service have to understand which, if any, security providers, tokens, or other such mechanisms are involved.

The script file `configWss.py` uses WLST to create and configure the default Web service security configuration, `default_wss`, for the active security realm. (The default Web service security configuration is used by *all* Web services in the domain unless they have been explicitly programmed to use a different configuration.) Further, this script makes sure that x509 tokens are supported, creates the needed security providers, and so forth.

The `configWss.py` file is the same as that in *WL_HOME*`\samples\server\examples\src\examples\webservices\wsrm_security\configWss.py`, with the changes shown in bold. The `build.xml` file provides the command input.

> **Note:** Long lines in this script have been formatted for readability.

***Example 2–27 configWss.py***

```
:
#Create credential provider for SCT
cpName='default_sct_cp'
wtm=defaultWss.lookupWebserviceCredentialProvider(cpName)
if wtm == None:
  print 'creating new webservice credential provider : ' + cpName
  wtm = defaultWss.createWebserviceCredentialProvider(cpName)
 wtm.setClassName('weblogic.wsee.security.wssc.v13.sct.
    ServerSCCredentialProvider')
  wtm.setTokenType('sct')
  cpm = wtm.createConfigurationProperty('TokenLifeTime')
  cpm.setValue('43200000')
else:
```

```
  print 'found exsiting bean for: ' + cpName

#Create credential provider for DK
cpName='default_dk_cp'
wtm=defaultWss.lookupWebserviceCredentialProvider(cpName)
if wtm == None:
        wtm = defaultWss.createWebserviceCredentialProvider(cpName)
      wtm.setClassName('weblogic.wsee.security.wssc.v13.
           dk.DKCredentialProvider')
        wtm.setTokenType('dk')
        cpm = wtm.createConfigurationProperty('Label')
        cpm.setValue('WS-SecureConversationWS-SecureConversation')
        cpm = wtm.createConfigurationProperty('Length')
        cpm.setValue('16')
else:
  print 'found exsiting bean for: DK ' + cpName
:
```

## 2.23.6 Revised configWss_Service.py

The `configWss_Service.py` script is similar to `configWss.py`, but it is used only when the source and destination Web service are hosted in two servers.

The `configWss_Service.py` file is the same as that in *`WL_HOME`*`\samples\server\examples\src\examples\webservices\wsrm_security\configWss_Service.py`, with the changes shown in bold. The `build.xml` file provides the command input.

> **Note:** Long lines in this script have been formatted for readability.

***Example 2–28 configWss_Service.py***

```
:
 #Create credential provider for SCT
cpName='default_sct_cp'
wtm=defaultWss.lookupWebserviceCredentialProvider(cpName)
if wtm == None:
  print 'creating new webservice credential provider : ' + cpName
  wtm = defaultWss.createWebserviceCredentialProvider(cpName)
 wtm.setClassName('weblogic.wsee.security.wssc.
       v13.sct.ServerSCCredentialProvider')
  wtm.setTokenType('sct')
  cpm = wtm.createConfigurationProperty('TokenLifeTime')
  cpm.setValue('43200000')
else:
  print 'found exsiting bean for: ' + cpName

#Create credential provider for DK
cpName='default_dk_cp'
wtm=defaultWss.lookupWebserviceCredentialProvider(cpName)
if wtm == None:
        wtm = defaultWss.createWebserviceCredentialProvider(cpName)
      wtm.setClassName('weblogic.wsee.security.wssc.v13.dk.
           DKCredentialProvider')
        wtm.setTokenType('dk')
        cpm = wtm.createConfigurationProperty('Label')
        cpm.setValue('WS-SecureConversationWS-SecureConversation')
        cpm = wtm.createConfigurationProperty('Length')
```

```
        cpm.setValue('16')
else:
  print 'found existing bean for: DK ' + cpName
:
```

### 2.23.7 Building and Running the Example

After you have changed the example to use the new policy namespace, follow the steps in the *WL_HOME*\samples\server\examples\src\examples\webservices\wsrm_security\instructions.html file to build and run the example.

There are no changes needed to these steps.

## 2.24 Securing Web Services Atomic Transactions

When using Web services atomic transactions, as described in "Using Web Services Atomic Transactions" in *Programming Advanced Features of JAX-WS Web Services for Oracle WebLogic Server*, it is recommended that you secure the application message headers that contain the coordination context and `IssuedTokens` using one of the following predefined policies:

- Wssp1.2-2007-SignAndEncryptWSATHeaders.xml—Specifies that the WS-AtomicTransaction headers are signed and encrypted.
- Wssp1.2-2007-Wsp1.5-SignAndEncryptWSATHeaders.xml—Specifies that the WS-AtomicTransaction headers are signed and encrypted. Web Services Policy 1.5 is used.

> **Note:** Because header encryption is available as part of the WS-Security 1.1 standard, it is highly recommended that you use only WS-Security 1.1 binding policies in conjunction with the policies listed above to secure the application request messages. WS-Security 1.1 binding policies contain <sp:Wss11> assertion in the policy and -Wss1.1 in the predefined policy name. If WS-Security 1.0 policies are used, WebLogic Server encrypts the header into WS-Security 1.0 non-standard format.

You can attach policies using one of the following methods:

- At design time, using the `@Policy` and `@Policies` annotations, as described in Section 2.6, "Example of Adding Security to a JAX-WS Web Service".
- At deployment time, using the WebLogic Server Administration Console, as described in Section 2.10, "Associating Policy Files at Runtime Using the Administration Console".

The following example shows how to secure a Web services atomic transaction programmatically, using the `@Policy` and `@Policies` annotations. Relevant code is shown in **bold**.

```
package jaxws.interop.rsp;
...
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.TransactionalFlowType;
```

```
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.jws.Policy;
import weblogic.jws.Policies;
...
@WebService(
     portName = "FlightServiceBindings_Basic",
     serviceName = "FlightService",
     targetNamespace = "http://wsinterop.org/samples",
     wsdlLocation = "/wsdls/FlightService.wsdl",
     endpointInterface = "jaxws.interop.rsp.IFlightService"
)
@BindingType("http://schemas.xmlsoap.org/wsdl/soap/http")
@javax.xml.ws.soap.Addressing
public class FlightServiceImpl implements IFlightService {
...
    @Transactional(value = Transactional.TransactionFlowType.SUPPORTS,
                   version = Transactional.Version.WSAT12)
    @Policies({
       @Policy(uri="policy:Wssp1.2-2007-EncryptBody.xml"
       @Policy(uri="policy:Wssp1.2-2007-SignAndEncryptWSATHeaders.xml"
       @Policy(uri="policy:Wssp1.2-2007-SignBody.xml"
       @Policy(uri="policy:Wssp1.2-2007-Wss1.1-X509-Basic256.xml"
    })
    public FlightReservationResponse reserveFlight(FlightReservationRequest request) {
        //replace with your impl here
        FlightReserverationEnitity entity = new FlightReserverationEnitity();
        entity.setAirlineID(request.getAirlineID());
        entity.setFlightNumber(request.getFlightNumber());
        entity.setFlightType(request.getFlightType());
        boolean successful = saveRequest(entity);
        FlightReservationResponse response = new FlightReservationResponse();
        if (!successful) {
           response.setConfirmationNumber("OF" +  CONF_NUMBER++ + "-" + request.getAirlineID() +
                   String.valueOf(entity.getId()));
        } else if (request.getFlightNumber() == null ||
                   request.getFlightNumber().trim().endsWith("LAS"))   {
           successful = false;
           response.setConfirmationNumber("OF" + "- No flight available for " +
                   request.getAirlineID());
        } else {
           response.setConfirmationNumber("OF" +  CONF_NUMBER++ + "-" + request.getAirlineID() +
                  String.valueOf(entity.getId()));
        }
        response.setSuccess(successful);
        return response;
    }
```

## 2.25 Proprietary Web Services Security Policy Files (JAX-RPC Only)

Previous releases of WebLogic Server, released before the formulation of the WS-SecurityPolicy specification, used security policy files written under the WS-Policy specification, using a proprietary schema for security policy.

**Note:** The security policy files written under the Web services security policy schema are deprecated in this release.

WS-SecurityPolicy 1.2 policy files and proprietary Web services security policy schema files are not mutually compatible; you cannot define both types of policy file in the same Web service. If you want to use WS-Security 1.1 features, you must use the WS-SecurityPolicy 1.2 policy file format.

This section describes the set of predefined Web services security policy schema files included in WebLogic Server. These policy files are all abstract; see Section 2.25.1, "Abstract and Concrete Policy Files" for details.

The policy assertions used in these security policy files to configure message-level security for a WebLogic Web service are based on the assertions described in the December 18, 2002 version of the *Web Services Security Policy Language* (WS-SecurityPolicy) specification. This means that although the exact syntax and usage of the assertions in WebLogic Server are different, they are similar in meaning to those described in the specification. The assertions are *not* based on later updates of the specification.

The predefined Web services security policy files are:

- Section 2.25.2, "Auth.xml" specifies that the client must authenticate itself. Can be used on its own, or together with `Sign.xml` and `Encrypt.xml`.
- Section 2.25.3, "Sign.xml" specifies that the SOAP messages are digitally signed. Can be used on its own, or together with `Auth.xml` and `Encrypt.xml`.
- Section 2.25.4, "Encrypt.xml" specifies that the SOAP messages are encrypted. Can be used on its own, or together with `Auth.xml` and `Sign.xml`.
- Section 2.25.5, "Wssc-dk.xml" specifies that the client and service share a security context when multiple messages are exchanged and that derived keys are used for encryption and digital signatures, as described by the WS-SecureConversation specification.

  **Note:** This predefined policy file is meant to be used on its own and not together with Auth.xml, Sign.xml, Encrypt.xml, or Wssc-sct.xml. Also, Oracle recommends that you use this policy file, rather than Wssc-sct.xml ( Section 2.25.6, "Wssc-sct.xml"), if you want the client and service to share a security context, due to its higher level of security.

- Section 2.25.6, "Wssc-sct.xml" specifies that the client and service share a security context when multiple messages are exchanged, as described by the WS-SecureConversation specification.

**Note:** This predefined policy file is meant to be used on its own and not together with Auth.xml, Sign.xml, Encrypt.xml, or Wssc-dk.xml. Also, Oracle provides this policy file to support the various use cases of the WS-SecureConversation specification; however, Oracle recommends that you use the Wssc-dk.xml (Section 2.25.5, "Wssc-dk.xml") policy file, rather than Wssc-sct.xml (Section 2.25.6, "Wssc-sct.xml"), if you want the client and service to share a security context, due to its higher level of security.

### 2.25.1 Abstract and Concrete Policy Files

The WebLogic Web services runtime environment recognizes two slightly different types of security policy files: *abstract* and *concrete*.

**Abstract policy files** do not explicitly specify the security tokens that are used for authentication, encryption, and digital signatures, but rather, the Web services runtime environment determines the security tokens when the Web service is deployed. Specifically, this means the `<Identity>` and `<Integrity>` elements (or assertions) of the policy files do not contain a `<SupportedTokens><SecurityToken>` child element, and the `<Confidentiality>` element policy file does not contain a `<KeyInfo><SecurityToken>` child element.

If your Web service is associated with only the predefined policy files, then client authentication requires username tokens. Web services support only one type of token for encryption and digital signatures (`X.509`), which means that in the case of the `<Integrity>` and `<Confidentiality>` elements, concrete and abstract policy files end up being essentially the same.

If your Web service is associated with an abstract policy file and it is published as an attachment to the WSDL (which is the default behavior), the static WSDL file packaged in the Web service archive file (JAR or WAR) will be slightly different than the dynamic WSDL of the deployed Web service. This is because the static WSDL, being abstract, does not include specific `<SecurityToken>` elements, but the dynamic WSDL *does* include these elements because the Web services runtime has automatically filled them in when it deployed the service. For this reason, in the code that creates the JAX-RPC stub in your client application, ensure that you specify the dynamic WSDL or you will get a runtime error when you try to invoke an operation: `HelloService service = new HelloService(Dynamic_WSDL);`

You can specify either the static or dynamic WSDL to the `clientgen` Ant task in this case. See "Browsing to the WSDL of the Web Service" in *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server* for information on viewing the dynamic WSDL of a deployed Web service.

**Concrete policy files** explicitly specify the details of the security tokens at the time the Web service is programmed. Programmers create concrete security policy files when they know, at the time they are programming the service, the details of the type of authentication (such as using `x509` or `SAML` tokens); whether multiple private key and certificate pairs from the keystore are going to be used for encryption and digital signatures; and so on.

### 2.25.2 Auth.xml

The WebLogic Server `Auth.xml` file, shown below, specifies that the client application invoking the Web service must authenticate itself with one of the tokens (username or X.509) that support authentication.

Because the predefined Web services security policy schema files are abstract, there is no specific username or X.509 token assertions in the `Auth.xml` file at development-time. Depending on how you have configured security for WebLogic Server, either a username token, an X.509 token, or both will appear in the actual runtime-version of the `Auth.xml` policy file associated with your Web service. Additionally, if the runtime-version of the policy file includes an X.509 token and it is applied to a client invoke, then the entire body of the SOAP message is signed.

If you want to specify that *only* X.509, and never username tokens, be used for identity, or want to specify that, when using X.509 for identity, only certain parts of the SOAP message be signed, then you must create a custom security policy file.

***Example 2–29 Auth.xml***

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
  >
  <wssp:Identity/>
</wsp:Policy>
```

## 2.25.3 Sign.xml

The WebLogic Server `Sign.xml` file specifies that the body and WebLogic-specific system headers of the SOAP message be digitally signed. It also specifies that the SOAP message include a Timestamp, which is digitally signed, and that the token used for signing is also digitally signed. The token used for signing is included in the SOAP message.

The following headers are signed when using the `Sign.xml` security policy file:

- `SequenceAcknowledgement`
- `AckRequested`
- `Sequence`
- `Action`
- `FaultTo`
- `From`
- `MessageID`
- `RelatesTo`
- `ReplyTo`
- `To`
- `SetCookie`
- `Timestamp`

The WebLogic Server `Sign.xml` file is shown below:

***Example 2–30 Sign.xml***

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
```

```
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-util
ity-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
  >
  <wssp:Integrity>
    <wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <wssp:CanonicalizationAlgorithm
          URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1" />
      <wssp:MessageParts
          Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
          wls:SystemHeaders()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1" />
      <wssp:MessageParts
          Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
          wls:SecurityHeader(wsu:Timestamp)
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1" />
      <wssp:MessageParts
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
          wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>
  </wssp:Integrity>
  <wssp:MessageAge/>
</wsp:Policy>
```

## 2.25.4 Encrypt.xml

The WebLogic Server `Encrypt.xml` file specifies that the entire body of the SOAP message be encrypted. By default, the encryption token is *not* included in the SOAP message.

***Example 2–31 Encrypt.xml***

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
  >
  <wssp:Confidentiality>
    <wssp:KeyWrappingAlgorithm URI="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
    <wssp:Target>
      <wssp:EncryptionAlgorithm
         URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <wssp:MessageParts
         Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
         wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:KeyInfo/>
  </wssp:Confidentiality>
</wsp:Policy>
```

### 2.25.5 Wssc-dk.xml

Specifies that the client and Web service share a security context, as described by the WS-SecureConversation specification, and that a derived key token is used. This ensures the highest form of security.

This policy file provides the following configuration:

- A derived key token is used to sign all system SOAP headers, the timestamp security SOAP header, and the SOAP body.
- A derived key token is used to encrypt the body of the SOAP message. This token is different from the one used for signing.
- Each SOAP message uses its own pair of derived keys.
- For both digital signatures and encryption, the key length is 16 (as opposed to the default 32)
- The lifetime of the security context is 12 hours.

If you need to change the default security context and derived key behavior, you will have to create a custom security policy file, described in later sections.

> **Note:** If you specify this predefined security policy file, you should not also specify any other predefined security policy file.

***Example 2–32 Wssc-dk.xml***

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-util
ity-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
  >
  <wssp:Integrity SupportTrust10="true">
    <wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
    <wssp:CanonicalizationAlgorithm
URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
          wls:SystemHeaders()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
          wls:SecurityHeader(wsu:Timestamp)
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
       wsp:Body()
      </wssp:MessageParts>
```

```
    </wssp:Target>
    <wssp:SupportedTokens>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"
        DerivedFromTokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
        <wssp:Claims>
          <wssp:Label>WS-SecureConversationWS-SecureConversation</wssp:Label>
          <wssp:Length>16</wssp:Length>
        </wssp:Claims>
      </wssp:SecurityToken>
    </wssp:SupportedTokens>
  </wssp:Integrity>
  <wssp:Confidentiality SupportTrust10="true">
    <wssp:Target>
      <wssp:EncryptionAlgorithm
URI="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
      <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
wsp:Body()</wssp:MessageParts>
    </wssp:Target>
    <wssp:KeyInfo>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"
        DerivedFromTokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
        <wssp:Claims>
          <wssp:Label>WS-SecureConversationWS-SecureConversation</wssp:Label>
          <wssp:Length>16</wssp:Length>
        </wssp:Claims>
      </wssp:SecurityToken>
    </wssp:KeyInfo>
  </wssp:Confidentiality>
  <wssp:MessageAge/>
</wsp:Policy>
```

### 2.25.6 Wssc-sct.xml

Specifies that the client and Web service share a security context, as described by the WS-SecureConversation specification. In this case, security context tokens are used to encrypt and sign the SOAP messages, which differs from Wssc-dk.xml (Section 2.25.5, "Wssc-dk.xml") in which derived key tokens are used. The `Wssc-sct.xml` policy file is provided to support all the use cases of the specification; for utmost security, however, Oracle recommends you always use Wssc-dk.xml (Section 2.25.5, "Wssc-dk.xml") when specifying shared security contexts due to its higher level of security.

This security policy file provides the following configuration:

- A security context token is used to sign all system SOAP headers, the timestamp security SOAP header, and the SOAP body.
- A security context token is used to encrypt the body of the SOAP message.
- The lifetime of the security context is 12 hours.

If you need to change the default security context and derived key behavior, you will have to create a custom security policy file, described in later sections.

---

**Note:** If you specify this predefined security policy file, you should not also specify any other predefined security policy file.

---

***Example 2–33 Wssc-sct.xml***

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-util
ity-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
  >
  <wssp:Integrity SupportTrust10="true">
    <wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
    <wssp:CanonicalizationAlgorithm
URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
         wls:SystemHeaders()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
         wls:SecurityHeader(wsu:Timestamp)
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
      wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:SupportedTokens>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
      </wssp:SecurityToken>
    </wssp:SupportedTokens>
  </wssp:Integrity>
  <wssp:Confidentiality SupportTrust10="true">
    <wssp:Target>
      <wssp:EncryptionAlgorithm
URI="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
      <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
wsp:Body()</wssp:MessageParts>
    </wssp:Target>
    <wssp:KeyInfo>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
      </wssp:SecurityToken>
    </wssp:KeyInfo>
  </wssp:Confidentiality>
  <wssp:MessageAge />
</wsp:Policy>
```

# 3 Configuring Transport-Level Security

Transport-level security refers to securing the connection between a client application and a Web service with Secure Sockets Layer (SSL) or HTTP Basic authentication, either alone or in combination.

SSL provides secure connections by allowing two applications connecting over a network to authenticate the other's identity and by encrypting the data exchanged between the applications. Authentication allows a server, and optionally a client, to verify the identity of the application on the other end of a network connection. A client certificate (two-way SSL) can be used to authenticate the user.

See "Secure Sockets Layer (SSL)" in *Understanding Security for Oracle WebLogic Server* for general information about SSL and the implementations included in WebLogic Server.

The following sections describe how to configure transport-level security for your Web service:


- Section 3.1, "Configuring Transport-Level Security Through Policy"
- Section 3.2, "Example of Using JWS Annotations in Your JWS File"
- Section 3.3, "Example of Configuring Transport Security for JAX-WS"
- Section 3.4, "New Two-Way Persistent SSL Client API for JAX-WS"
- Section 3.5, "Configuring Transport-Level Security Via UserDataConstraint: Main Steps (JAX-RPC Only)"
- Section 3.6, "Configuring Two-Way SSL for a Client Application"
- Section 3.7, "Using a Custom SSL Adapter with Reliable Messaging"


## 3.1 Configuring Transport-Level Security Through Policy

You can specify a policy that requires SSL, HTTP BASIC authentication, or both.

If the policy requires SSL, make sure you configure SSL for the core WebLogic Server security subsystem. You can configure one-way SSL where WebLogic Server is required to present a certificate to the client application, or two-way SSL where both the client applications and WebLogic server present certificates to each other.

To configure two-way or one-way SSL for the core WebLogic Server security subsystem, see "Configuring SSL" in *Securing Oracle WebLogic Server*.

For example, the Oracle-supplied `Wssp1.2-2007-Saml2.0-Bearer-Https.xml` policy file includes the following assertion indicating that the policy is expecting a client certificate via SSL, as shown in Example 3–1.

***Example 3–1 Specifying SSL in a Policy***

```
<sp:TransportToken>
<wsp:Policy>
<sp:HttpsToken/>
</wsp:Policy>
</sp:TransportToken>
```

If two-way SSL is required, also use the `RequireClientCertificate` assertion, as shown in Example 3–2.

***Example 3–2 Two-Way SSL in a Policy***

```
<sp:TransportToken>
<wsp:Policy>
<sp:HttpsToken >
<wsp:Policy>
<sp:RequireClientCertificate/>
</wsp:Policy>
</sp:HttpsToken>
</wsp:Policy>
</sp:TransportToken>
```

The `Wssp1.2-2007-Https-BasicAuth.xml` policy file requires both a client certificate via SSL and HTTP BASIC Authentication, as shown in Example 3–3.

***Example 3–3 SSL and HTTP Basic Authentication in a Policy***

```
<sp:TransportToken>
<wsp:Policy>
<sp:HttpsToken>
<wsp:Policy>
<sp:HttpBasicAuthentication/>
</wsp:Policy>
</sp:HttpsToken>
</wsp:Policy>
</sp:TransportToken>
```

### 3.1.1 Configuring Transport-Level Security Through Policy: Main Steps

To configure transport-level Web services security via one or more policy files:

1. Configure SSL for the core WebLogic Server security subsystem.

   You can configure one-way SSL where WebLogic Server is required to present a certificate to the client application, or two-way SSL where both the client applications and WebLogic server present certificates to each other.

   To configure two-way or one-way SSL for the core WebLogic Server security subsystem, see "Configuring SSL" in *Securing Oracle WebLogic Server*.

2. Use `@Policy` or `@Policies` JWS annotations in your JWS file, or associate policy files only at runtime using the Administration Console, or specify some policy files using the annotations and then associate additional ones at runtime.

> **Note:** If you specify a transport-level security policy for your Web service, it must be at the class level.
>
> In addition, the transport-level security policy must apply to both the inbound and outbound directions. That is, you cannot have HTTPS for inbound and HTTP for outbound.

3. If you added `@Policy` or `@Policies` JWS annotations in your JWS file, compile and redeploy your Web service as part of the normal iterative development process.
4. When you run the client application that invokes the Web service, specify certain properties to indicate the SSL implementation that your application should use. In particular:
   - To specify the Certicom SSL implementation, use the following properties

     ```
     -Djava.protocol.handler.pkgs=weblogic.net
     -Dweblogic.security.SSL.trustedCAKeyStore=trustStore
     ```

     where *trustStore* specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server's certificate). To disable host name verification, also specify the following property:

     ```
     -Dweblogic.security.SSL.ignoreHostnameVerification=true
     ```

   - To specify Sun's SSL implementation, use the following properties:

     ```
     -Djavax.net.ssl.trustStore=trustStore
     ```

     where *trustStore* specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server's certificate). To disable host name verification, also specify the following property:

     ```
     -Dweblogic.wsee.client.ssl.stricthostchecking=false
     ```

     See Section 3.6, "Configuring Two-Way SSL for a Client Application" for details about two-way SSL.

## 3.2 Example of Using JWS Annotations in Your JWS File

If you specify a transport-level security policy for your Web service, it must be at the class level, as shown in the following example.

**Transport-level policy at the class level**

```
@Policy(uri="policy:Wssp1.2-2007-Saml2.0-Bearer-Https.xml")
public class EchoService {
   ....
```

## 3.3 Example of Configuring Transport Security for JAX-WS

This section describes a simple example for configuring JAX-WS with Transport Security from a standalone client for one-way SSL.

See the following documentation for additional prerequisite information:

- "Configuring SSL" in *Securing Oracle WebLogic Server*

- "Configure SSL" in the *Oracle WebLogic Server Administration Console Help*
- "Configure KeyStores" in the *Oracle WebLogic Server Administration Console Help*

### 3.3.1 One-Way SSL (HTTPS and HTTP Basic Authentication Example)

The Web service Java source is shown in Example 3–4:

> **Note:** If you specify a transport-level security policy for your Web service, it must be at the class level.
>
> In addition, the transport-level security policy must apply to both the inbound and outbound directions. That is, you cannot have HTTPS for inbound and HTTP for outbound.

***Example 3–4 Web Service One-Way SSL Example***

```
package httpbasicauth
import javax.jws.WebMethod;
import javax.jws.WebService;

import weblogic.jws.Policy;

@WebService(name="HttpsBasicAuth", portName="HttpsBasicAuthSoapPort"
 targetNamespace="https://httpsbasicauth")

// Security Policy for Https and Http Basic Authentication
@Policy(uri = "policy:Wssp1.2-2007-Https-BasicAuth.xml)


public class HttpsBasicAuth {


 public HttpsBasicAuth() {}

 WebMethod()
 public String echoString(String input) {

   return("[HttpsBasicAuth.echoString]: " + input);

 }

}
```

The standalone Java Web service client code that uses "weblogic.net" as the Java protocol handler is shown in Example 3–5:

***Example 3–5 Web Service Client One-Way SSL Example With weblogic.net***

```
package httpbasicauth.client

import java.net.URL;
import java.security.cert.X509Certificate;
import java.util.Map;

import javax.xml.namespace.QName;

import javax.xml.ws.BindingProvider;
```

```
import httpsbasicauth.client.HttpsBasicAuthService;
import httpsbasicauth.client.HttpsBasicAuth;


public class HttpsBasicAuthClient

  private final static String ENDPOINT = .....;
  private final static String TARGET_NAMESPACE = "https://httpsbasicauth
  private final static String USERNAME = ....;
  private final static String PASSWORD = ....;
  private final static String TRUST_STORE_LOCATION = .....;
  private final static String TARGET_NAMESPACE = ....;

  private HttpsBasicAuthService service;
  private HttpsBasicAuth stub;


   public HttpsBasicAuthClient() {

    try {
      // This ignores the host name verifcation for the Public Certificate used by
the Server

System.setProperty("weblogic.security.SSL.ignoreHostnameVerification","true");

      System.setProperty("java.protocol.handler.pkgs", "weblogic.net");
      System.setProperty("weblogic.security.TrustKeyStore","CustomTrust");
      System.setProperty("weblogic.security.CustomTrustKeyStoreFileName", "TRUST_
STORE_LOCATION");
      System.setProperty("weblogic.security.CustomTrustKeyStorePassPhrase","TRUST_
STORE_PASSWORD");
      System.setProperty("weblogic.security.CustomTrustKeyStoreType","JKS");

      URL url = new URL(endpoint+"?WSDL");
      QName serviceName = new QName(TARGET_NAMESPACE, "HttpsBasicAuthService");

      service = new HttpsBasicAuthService();

      stub = service.getHttpsBasicAuthSoapPort();

      BindingProvider bp = (BindingProvider) stub;

      Map<String,Object> context = bp.getRequestContext();

      context.put(BindingProvider.USERNAME_PROPERTY, USERNAME)
      context.put(BindingProvider.PASSWORD_PROPERTY, PASSWORD);
      context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, ENDPOINT);

    } catch (Exception e) {
      System.out.println("Error in creating the stub : " + e.getMessage());
      if (verbose) e.printStackTrace();
    }
  }

  public void invokeEchoString() throws Exception {

    String output = stub.echoString(ENDPOINT);

    System.out.println("[HttpsBasicAuthClient.invokeGEchoString]: " + output);
```

```
  }

  public static void main(String[] argv) throws Exception {

    HttpsBasicAuthClient client = new HttpsBasicAuthClient();

    System.setProperty("weblogic.wsee.verbose","*");

    System.out.println("----------------------");
    System.out.println("            Invoking echoString            ");
    client.invokeEchoString();

  }

}
```

The standalone Java Web service client code that uses the default Java protocol handler is shown in Example 3–6:

***Example 3–6 Web Service Client One-Way SSL Example With java.net***

```
package httpbasicauth.client

import java.net.URL;
import java.security.cert.X509Certificate;
import java.util.Map;

import javax.xml.namespace.QName;

import javax.xml.ws.BindingProvider;

import httpsbasicauth.client.HttpsBasicAuthService;
import httpsbasicauth.client.HttpsBasicAuth;


public class HttpsBasicAuthClient

  private final static String ENDPOINT = .....;
  private final static String TARGET_NAMESPACE = "https://httpsbasicauth
  private final static String USERNAME = ....;
  private final static String PASSWORD = ....;
  private final static String TRUST_STORE_LOCATION = .....;
  private final static String TARGET_NAMESPACE = ....;

  private HttpsBasicAuthService service;
  private HttpsBasicAuth stub;

  public HttpsBasicAuthClient() {

     try {

       System.setProperty("java.protocol.handler.pkgs", "java.net");
       System.setProperty("javax.net.ssl.trustStore", TRUST_STORE_LOCATION);
       System.setProperty("javax.net.ssl.trustStorePassword", TRUST_STORE_
PASSWORD);


       URL url = new URL(ENDPOINT+"?WSDL");
       QName serviceName = new QName(TARGET_NAMESPACE, "HttpsBasicAuthService");
```

```
        service = new HttpsBasicAuthService();

        stub = service.getHttpsBasicAuthSoapPort();

        BindingProvider bp = (BindingProvider) stub;

        Map<String,Object> context = bp.getRequestContext();

        context.put(BindingProvider.USERNAME_PROPERTY, USERNAME)
        context.put(BindingProvider.PASSWORD_PROPERTY, PASSWORD);
        context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, ENDPOINT);

      } catch (Exception e) {
       System.out.println("Error in creating the stub : " + e.getMessage());
       if (verbose) e.printStackTrace();
     }

  }


  public void invokeEchoString() throws Exception {

    String output = stub.echoString(ENDPOINT);

    System.out.println("[HttpsBasicAuthClient.invokeGEchoString]: " + output);

  }

  public static void main(String[] argv) throws Exception {

    HttpsBasicAuthClient client = new HttpsBasicAuthClient();

    System.setProperty("weblogic.wsee.verbose","*");

    System.out.println("----------------------");
    System.out.println("            Invoking echoString            ");
    client.invokeEchoString();

  }

}
```

The related portion of the ant build file is shown in Example 3–7:

***Example 3–7 Ant Build File***

```
<property name="output.dir" value="../../build/httpsbasicauth" />
<property name="service.dir" value="${output.dir}/httpsbasicauthApp" />
<property name="output.dir.client" value="${output.dir}/client" />
<property name="clientclasses.dir" value="${output.dir}/client" />
<property name="service.name" value="HttpsBasicAuth" />
<property name="wsdl.name" value="HttpsBasicAuthService" />
<property name="packageName" value="httpsbasicauth.client" />

<path id="client.class.path">
  <pathelement path="${java.class.path}" />
  <pathelement path="${clientclasses.dir}" />
</path>

<taskdef name="clientgen"
          classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
```

```
<taskdef name="jwsc" classname="weblogic.wsee.tools.anttasks.JwscTask"/>


 <target name="jwsc">

    <jwsc srcdir="." destdir="${output.dir.server}" sourcepath="../" debug="true"
keepGenerated="true">

      <module name="HttpsBasicAuth" contextPath="httpsbasicauth">

        <jws file="HttpsBasicAuth.java" type="JAXWS" generateWsdl="true">
           <WLHttpTransport contextPath="httpsbasicauth"
serviceUri="httpsbasicauth"/>
        </jws>

    </jwsc>

 </target>

  <target name="client">

    <clientgen
wsdl="jar:file:${service.dir}/${service.name}.war!/WEB-INF/${wsdl.name}.wsdl"
               type="JAXWS"
               destDir="${clientclasses.dir}"
               packageName="${packageName}">

    </clientgen>

     <javac srcdir="${clientclasses.dir}"
            destdir="${clientclasses.dir}"
            includes="**/*.java"
            classpathref="client.class.path" />

    <javac srcdir="./"
           destdir="${clientclasses.dir}"
           includes="HttpsBasicAuthClient.java"
           classpathref="client.class.path" />

  </target>

  <target name="run">

    <java classname="httpsbasicauth.client.HttpsBasicAuthClient"
          classpathref="client.class.path"
          fork="true" />
  </target>
```

## 3.4 New Two-Way Persistent SSL Client API for JAX-WS

JAX-RPC clients can use the SSLAdapter mechanism described in Section 3.7, "Using a Custom SSL Adapter with Reliable Messaging" to persist the state of a request over an SSL connection. In doing so, they persist the instance of the custom SSLAdapter used to establish the connection.

This release of Oracle WebLogic Server includes a new two-way SSL client API for JAX-WS that you can use to construct an *SSLSocketFactory* from system properties or from a new *weblogic.wsee.jaxws.sslclient.PersistentSSLInfo class*. The API can persist

SSL info for Reliable Messaging, callbacks, and so forth, and supports the following well-known system properties:

- *weblogic.wsee.client.ssl.relaxedtrustmanager*
- *weblogic.security.SSL.ignoreHostnameVerification*

The following new classes are available. See the Javadoc for complete descriptions.

- *weblogic.wsee.jaxws.sslclient.SSLClientUtil*. This class has the following methods:
  - *public static SSLSocketFactory getSSLSocketFactory(KeyManager[] kms, TrustManager[] tms);*
  - *public static SSLSocketFactory getSSLSocketFactory(PersistentSSLInfo sslInfo);*
  - *public static SSLSocketFactory getSSLSocketFactoryFromSysProperties();*
- *weblogic.wsee.jaxws.sslclient.PersistentSSLInfo*, a Javabean for setting SSL info.
- *weblogic.wsee.jaxws.JAXWSProperties*, includes a *CLIENT_PERSISTENT_SSL_INFO* property.

### 3.4.1 Example of Getting SSLSocketFactory From System Properties

Example 3–8 shows an example of getting the SSLSocketFactory from system properties and using them in the request context.

> **Note:** The *clientKeyStore* and *clientKeyStorePasswd* have this restriction: the SSL package of J2SE requires that the password of the client's private key must be the same as the password of the client's keystore. For this reason, the client keystore can include only one private key and X.509 certificate pair.

***Example 3–8 Getting SSLSocketFactory From System Properties***

```
String clientKeyStore = ...;
      String clientKeyStorePasswd = ...;
      String trustKeystore = ...;
      String trustKeystorePasswd = ...;

      System.setProperty("javax.net.ssl.keyStore", clientKeyStore);
      System.setProperty("javax.net.ssl.keyStorePassword", clientKeyStorePasswd);
      System.setProperty("javax.net.ssl.trustStore", trustKeystore);
      System.setProperty("javax.net.ssl.trustStorePasswd", trustKeystorePasswd);

      ((BindingProvider) port).getRequestContext().put(
          JAXWSProperties.SSL_SOCKET_FACTORY,
          SSLClientUtil.getSSLSocketFactoryFromSysProperties());
```

Example 3–9 shows an example of getting SSLSocketFactory from persistent info (PersistentSSLInfo), as well as directly setting a *SSLSocketFactory* if persistence is not needed.

***Example 3–9 Getting SSLSocketFactory from PersistentSSLInfo***

```
String clientKeyStore = ...;
      String clientKeyStorePasswd = ...;
      String clientKeyAlias = ...;
      String clientKeyPass = ...;
      String trustKeystore = ...;
```

```
        String trustKeystorePasswd = ...;

        PersistentSSLInfo sslInfo = new PersistentSSLInfo();
        sslInfo.setKeystore(clientKeyStore);
        sslInfo.setKeystorePassword(clientKeyStorePasswd);
        sslInfo.setKeyAlias(clientKeyAlias);
        sslInfo.setKeyPassword(clientKeyPass);
        sslInfo.setTrustKeystore(trustKeystore);

        //user can print out the sslInfo for debug
        System.out.print(sslInfo.toString());

//Put sslInfo into requestContext for persistence, it might be required by JAX-WS
advance features, such as, RM, Callback
        ((BindingProvider) port).getRequestContext().put(
          JAXWSProperties.CLIENT_PERSISTENT_SSL_INFO, sslInfo);

        //Alternatively, you can directly set a SSLSocketFactory if persistence is
not necessary. Note: The following line should be omitted if sslInfo is set with
above line.
        ((BindingProvider) port).getRequestContext().put(
          JAXWSProperties.SSL_SOCKET_FACTORY,
          SSLClientUtil.getSSLSocketFactory(sslInfo));
```

*sslInfo* can set a key alias (*clientKeyAlias*) that points to a key in keystore (as an SSL client-side key) in the event that the client keystore has multiple keys.

## 3.5 Configuring Transport-Level Security Via UserDataConstraint: Main Steps (JAX-RPC Only)

The `UserDataConstraint` annotation requires that the Web service be invoked using the HTTPS transport.

To configure transport-level Web services security via the `UserDataConstraint` annotation in your JWS file:

1. Configure SSL for the core WebLogic Server security subsystem.

   You can configure one-way SSL where WebLogic Server is required to present a certificate to the client application, or two-way SSL where both the client applications and WebLogic server present certificates to each other.

   To configure two-way or one-way SSL for the core WebLogic Server security subsystem, see "Configuring SSL" in *Securing Oracle WebLogic Server*.

2. In the JWS file that implements your Web service, add the `@weblogic.jws.security.UserDataConstraint` annotation to require that the Web service be invoked using the HTTPS transport.

   For details, see "weblogic.jws.security.UserDataConstraint" in the *WebLogic Web Services Reference for Oracle WebLogic Server*.

3. Recompile and redeploy your Web service as part of the normal iterative development process.

   See "Developing WebLogic Web Services" in *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*

4. Update the `build.xml` file that invokes the `clientgen` Ant task to use a static WSDL to generate the JAX-RPC stubs of the Web service, rather than the dynamic deployed WSDL of the service.

The reason `clientgen` cannot generate the stubs from the dynamic WSDL in this case is that when you specify the `@UserDataConstraint` annotation, all client applications are required to specify a truststore, including `clientgen`. However, there is currently no way for `clientgen` to specify a truststore, thus the Ant task must generate its client components from a static WSDL that describes the Web service in the same way as the dynamic WSDL.

5. When you run the client application that invokes the Web service, specify certain properties to indicate the SSL implementation that your application should use. In particular:

   - To specify the Certicom SSL implementation, use the following properties

     ```
     -Djava.protocol.handler.pkgs=weblogic.net
     -Dweblogic.security.SSL.trustedCAKeyStore=trustStore
     ```

     where *trustStore* specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server's certificate).To disable host name verification, also specify the following property:

     ```
     -Dweblogic.security.SSL.ignoreHostnameVerification=true
     ```

   - To specify Sun's SSL implementation, use the following properties:

     ```
     -Djavax.net.ssl.trustStore=trustStore
     ```

     where *trustStore* specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server's certificate). To disable host name verification, also specify the following property:

     ```
     -Dweblogic.wsee.client.ssl.stricthostchecking=false
     ```

     See Section 3.6, "Configuring Two-Way SSL for a Client Application" for details about two-way SSL.

## 3.6 Configuring Two-Way SSL for a Client Application

> **Note:** Web services using asynchronous or reliable messaging will automatically use the server's SSL certificate when establishing a new connection (back from the receiving service to the sending service) for the purposes of sending asynchronous responses, acknowledgments, and so forth.

If you configured two-way SSL for WebLogic Server, the client application must present a certificate to WebLogic Server, in addition to WebLogic Server presenting a certificate to the client application as required by one-way SSL. You must also follow these requirements:

- Create a client-side keystore that contains the client's private key and X.509 certificate pair.

  The SSL package of J2SE requires that the password of the client's private key must be the same as the password of the client's keystore. For this reason, the client keystore can include only *one* private key and X.509 certificate pair.

- Configure the core WebLogic Server's security subsystem, mapping the client's X.509 certificate in the client keystore to a user. See "Configuring a User Name Mapper" in *Securing Oracle WebLogic Server*.
- Create a *truststore* which contains the certificates that the client trusts; the client application uses this truststore to validate the certificate it receives from WebLogic Server. Because of the J2SE password requirement described in the preceding bullet item, this truststore must be different from the keystore that contains the key pair that the client presents to the server.

  You can use the Cert Gen utility or Sun Microsystem's keytool (`http://java.sun.com/javase/6/docs/tooldocs/solaris/keytool.html`) utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

  See "Obtaining Private Keys, Digital Certificates, and Trusted Certificate Authorities" in *Securing Oracle WebLogic Server*.
- When you run the client application that invokes the Web service, specify the following properties:
  - -Djavax.net.ssl.trustStore=trustStore
  - -Djavax.net.ssl.trustStorePassword=trustStorePassword

    where *trustStore* specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server's certificate) and *trustStorePassword* specifies the truststore's password.

    The preceding properties are in addition to the standard properties you must set to specify the client-side keystore:
  - -Djavax.net.ssl.keyStore=keyStore
  - -Djavax.net.ssl.keyStorePassword=keyStorePassword

## 3.7 Using a Custom SSL Adapter with Reliable Messaging

> **Note:** All objects placed into Stub and MessageContext properties must be serializable and externalizable, and must have their implementations available on the server system CLASSPATH. This section describes the specific case of a custom SSLAdapter implementation.

You can use a custom *SSLAdapter* implementation to provide client certificates and other services needed to establish SSL connections between client and server when using reliable messaging or buffering. The reliable messaging and buffering subsystems persist the state of a request over an SSL connection. In doing so, they persist the instance of the custom SSLAdapter used to establish the connection.

When the request is restored from persistence, the persistence facility must have access to the custom SSLAdapter class in order to properly restore the custom SSLAdapter object saved with the request. To allow for this, you must provide your custom SSLAdapter class via the server's system CLASSPATH (and not within an application deployed to the server).

The custom SSLAdapter must extend *SSLAdapter*, and is installed and enabled via the following procedure:

1. Create an instance of `weblogic.wsee.connection.transport.https.HttpsTransportInfo`.
2. Set the custom SSL adapter on that transport info by calling `HttpsTransportInfo.setSSLAdapter(SSLAdapter adapter)`.
3. Set the transport info on the web services stub instance `(stub of type javax.xml.rpc.Stub)` by calling

   ```
   stub._setProperty(weblogic.wsee.connection.soap.SoapClientConnection.TRANSPORT_
   INFO_PROPERTY,ti);
   ```

   Where `stub` is the Web services stub, and `ti` is the `HttpsTransportInfo` you configured.

If you do not follow this procedure and provide the custom SSLAdapter class on the system CLASSPATH, a `ClassNotFoundException` exception is generated:

```
java.io.IOException: java.lang.ClassNotFoundException:
 examples.webservices.client.ServiceBase$TestSSLAdapter
```

# 4

# Configuring Access Control Security (JAX-RPC Only)

The following sections describe how to configure security for your Web service:


- Section 4.1, "Configuring Access Control Security: Main Steps"
- Section 4.2, "Updating the JWS File With the Security-Related Annotations"
- Section 4.3, "Updating the JWS File With the @RunAs Annotation"
- Section 4.4, "Setting the Username and Password When Creating the Service Object"


## 4.1 Configuring Access Control Security: Main Steps

Access control security refers to configuring the Web service to control the users who are allowed to access it, and then coding your client application to authenticate itself, using HTTP/S or username tokens, to the Web service when the client invokes one of its operations.

You specify access control security for your Web service by using one or more of the following annotations in your JWS file:

- `weblogic.jws.security.RolesAllowed`
- `weblogic.jws.security.SecurityRole`
- `weblogic.jws.security.RolesReferenced`
- `weblogic.jws.security.SecurityRoleRef`
- `weblogic.jws.security.RunAs`

> **Note:** The @weblogic.security.jws.SecurityRoles and @weblogic.security.jws.SecurityIdentity JWS annotations were deprecated as of WebLogic Server 9.1.

The following procedure describes the high-level steps to use these annotations to enable access control security; later sections in the chapter describe the steps in more detail.

> **Note:** It is assumed in the following procedure that you have already created a JWS file that implements a WebLogic Web service and you want to update it with access control security.
>
> It is also assumed that you use Ant build scripts to iteratively develop your Web service and that you have a working build.xml file that you can update with new information.
>
> Finally, it is assumed that you have a client application that invokes the non-secured Web service. If these assumptions are not true, see *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*.

1. Update your JWS file, adding the `@weblogic.jws.security.RolesAllowed`, `@weblogic.jws.security.SecurityRole`, `@weblogic.jws.security.RolesReferenced`, or `@weblogic.jws.security.SecurityRoleRef` annotations as needed at the appropriate level (class or operation).

   See Section 4.2, "Updating the JWS File With the Security-Related Annotations".

2. Optionally specify that WebLogic Server internally run the Web service using a specific role, rather than the role assigned to the user who actually invokes the Web service, by adding the `@weblogic.jws.security.RunAs` JWS annotation.

   See Section 4.3, "Updating the JWS File With the @RunAs Annotation".

3. Optionally specify that your Web service can be, or is required to be, invoked using HTTPS by adding the `@weblogic.jws.security.UserDataConstraint` JWS annotation.

   See Section 3.5, "Configuring Transport-Level Security Via UserDataConstraint: Main Steps (JAX-RPC Only)" for details. This section also discusses how to update your client application to use SSL.

4. Recompile and redeploy your Web service as part of the normal iterative development process.

   See "Developing WebLogic Web Services" in *Getting Started With WebLogic Web Services Using JAX-RPC*.

5. Using the Administration Console, create valid WebLogic Server users, if they do not already exist. If the mapping of users to roles is external, also use the Administration Console to create the roles specified by the `@SecurityRole` annotation and map the users to the roles.

   > **Note:** The mapping of users to roles is defined externally if you do not specify the mapToPrincipals attribute of the @SecurityRole annotation in your JWS file to list all users who can invoke the Web service.

   See "Users, Groups, and Security Roles" in *Securing WebLogic Resources Using Roles and Policies*.

6. Update your client application to use the `HttpTransportInfo` WebLogic API to specify the appropriate user and password when creating the `Service` object.

   See Section 4.4, "Setting the Username and Password When Creating the Service Object".

7. Update the `clientgen` Ant task in your `build.xml` file to specify the username and password of a valid WebLogic user (in the case where your Web service uses the `@RolesAllowed` annotation) and the trust store that contains the list of trusted certificates, including WebLogic Server's (in the case you specify `@UserDataConstraint`).

   You do this by adding the standard Ant `<sysproperty>` nested element to the `clientgen` Ant task, and set the `key` attribute to the required Java property, as shown in the following example.

   **Note:** The example hard-codes the username and password; prompting for both provides more security. You need the username and password for @RolesAllowed, and trustStore if SSL must be used.

```
<clientgen
    wsdl="http://example.com/myapp/myservice.wsdl"
    destDir="/output/clientclasses"
    packageName="myapp.myservice.client"
    serviceName="StockQuoteService" >
    <sysproperty key="javax.net.ssl.trustStore"
                 value="/keystores/DemoTrust.jks"/>
    <sysproperty key="weblogic.wsee.client.ssl.stricthostchecking"
                 value="false"/>
    <sysproperty key="javax.xml.rpc.security.auth.username"
                 value="juliet"/>
    <sysproperty key="javax.xml.rpc.security.auth.password"
                 value="secret"/>
</clientgen>
```

8. Regenerate client-side components and recompile client Java code as usual.

## 4.2 Updating the JWS File With the Security-Related Annotations

Use the WebLogic-specific `@weblogic.jws.security.RolesAllowed` annotation in your JWS file to specify an array of `@weblogic.jws.security.SecurityRoles` annotations that list the roles that are allowed to invoke the Web service. You can specify these two annotations at either the class- or method-level. When set at the class-level, the roles apply to all public operations. You can add additional roles to a particular operation by specifying the annotation at the method level.

The `@SecurityRole` annotation has the following two attributes:

- `role` is the name of the role that is allowed to invoke the Web service.
- `mapToPrincipals` is the list of users that map to the role. If you specify one or more users with this attribute, you do not have to externally create the mapping between users and roles, typically using the Administration Console. However, the mapping specified with this attribute applies only within the context of the Web service.

The `@RolesAllowed` annotation does not have any attributes.

You can also use the `@weblogic.jws.security.RolesReferenced` annotation to specify an array of `@weblogic.jws.security.SecurityRoleRef` annotations that list references to existing roles. For example, if the role `manager` is already allowed to invoke the Web service, you can specify that the `mgr` role be linked to the `manager` role and any user mapped to `mgr` is also able to invoke the Web service. You can specify these two annotations only at the class-level.

The `@SecurityRoleRef` annotation has the following two attributes:

- `role` is the name of the role reference.
- `link` is the name of the already-specified role that is allowed to invoke the Web service. The value of this attribute corresponds to the value of the `role` attribute of a `@SecurityRole annotation` specified in the same JWS file.

The `@RolesReferenced` annotation does not have any attributes.

The following example shows how to use the annotations described in this section in a JWS file, with the relevant sections shown in bold:

```
package examples.webservices.security_roles;
import javax.jws.WebMethod;
import javax.jws.WebService;
// WebLogic JWS annotations
import weblogic.jws.WLHttpTransport;
import weblogic.jws.security.RolesAllowed;
import weblogic.jws.security.RolesReferenced;
import weblogic.jws.security.SecurityRole;
import weblogic.jws.security.SecurityRoleRef;
@WebService(name="SecurityRolesPortType",
            serviceName="SecurityRolesService",
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="security",
                 serviceUri="SecurityRolesService",
                  portName="SecurityRolesPort")
@RolesAllowed (  {
    @SecurityRole (role="manager",
                   mapToPrincipals={ "juliet","amanda" }),
    @SecurityRole (role="vp")
} )
@RolesReferenced (
    @SecurityRoleRef (role="mgr", link="manager")
)
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 */
public class SecurityRolesImpl {
  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

The example shows how to specify that only the `manager`, `vp`, and `mgr` roles are allowed to invoke the Web service. The `mgr` role is actually a reference to the `manager` role. The users `juliet` and `amanda` are mapped to the `manager` role within the context of the Web service. Because no users are mapped to the `vp` role, it is assumed that the mapping occurs externally, typically using the Administration Console to update the WebLogic Server security realm.

See "JWS Annotation Reference" in *WebLogic Web Services Reference* for reference information on these annotations.

## 4.3 Updating the JWS File With the @RunAs Annotation

Use the WebLogic-specific `@weblogic.jws.security.RunAs` annotation in your JWS file to specify that the Web service is always run as a particular role. This means that regardless of the user who initially invokes the Web service (and the role to which the user is mapped), the service is internally executed as the specified role.

You can set the `@RunAs` annotation only at the class-level. The annotation has the following attributes:

- `role` is the role which the Web service should run as.
- `mapToPrincipal` is the principal user that maps to the role.

The following example shows how to use the `@RunAs` annotation in a JWS file, with the relevant sections shown in bold:

```
package examples.webservices.security_roles;
import javax.jws.WebMethod;
import javax.jws.WebService;
// WebLogic JWS annotations
import weblogic.jws.WLHttpTransport;
import weblogic.jws.security.RunAs;
@WebService(name="SecurityRunAsPortType",
            serviceName="SecurityRunAsService",
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="security_runas",
                 serviceUri="SecurityRunAsService",
                 portName="SecurityRunAsPort")
@RunAs (role="manager", mapToPrincipal="juliet")
/**
 * This JWS file forms the basis of simple WebLogic
 * Web Service with a single operation: sayHello
 *
 */
public class SecurityRunAsImpl {
  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

## 4.4 Setting the Username and Password When Creating the Service Object

When you use the `@RolesAllowed` JWS annotation to secure a Web service, only the specified roles are allowed to invoke the Web service operations. This means that you must specify the username and password of a user that maps to the role when creating the `Service` object in your client application that invokes the protected Web service.

WebLogic Server provides the `HttpTransportInfo` class for setting the username and password and passing it to the `Service` constructor. The following example is based on the standard way to invoke a Web service from a standalone Java client (as described in *Invoking Web Services* in *Getting Started With WebLogic Web Services Using JAX-RPC*) but also shows how to use the `HttpTransportInfo` class to set the username and password. The sections in bold are discussed after the example.

```
package examples.webservices.sec_wsdl.client;
import weblogic.wsee.connection.transport.http.HttpTransportInfo;
import java.rmi.RemoteException;
```

```
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the SecWsdlService Web service.
 *
 * @author Copyright © 1996, 2008, Oracle and/or its affiliates.
 * All rights reserved.
 */
public class Main {
  public static void main(String[] args)
      throws ServiceException, RemoteException{
    HttpTransportInfo info = new HttpTransportInfo();
    info.setUsername("juliet".getBytes());
    info.setPassword("secret".getBytes());
    SecWsdlService service = new SecWsdlService_Impl(args[0] + "?WSDL", info);
    SecWsdlPortType port = service.getSecWsdlPort();
    try {
      String result = null;
      result = port.sayHello("Hi there!");
      System.out.println( "Got result: " + result );
    } catch (RemoteException e) {
      throw e;
    }
  }
}
```

The main points to note in the preceding example are as follows:

- Import the `HttpTransportInfo` class into your client application:

  ```
  import weblogic.wsee.connection.transport.http.HttpTransportInfo;
  ```

- Use the `setXXX()` methods of the `HttpTransportInfo` class to set the username and password:

  ```
  HttpTransportInfo info = new HttpTransportInfo();
  info.setUsername("juliet".getBytes());
  info.setPassword("secret".getBytes());
  ```

  In the example, it is assumed that the user `juliet` with password `secret` is a valid WebLogic Server user and has been mapped to the role specified in the `@RolesAllowed` JWS annotation of the Web service.

  If you are accessing a Web service using a proxy, the Java code would be similar to:

  ```
  HttpTransportInfo info = new HttpTransportInfo();
  Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
  Integer.parseInt(proxyPort)));
  info.setProxy(p);
  info.setProxyUsername(user.getBytes());
  info.setProxyPassword(pass.getBytes());
  ```

- Pass the `info` object that contains the username and password to the `Service` constructor as the second argument, in addition to the standard WSDL first argument:

  ```
  SecWsdlService service = new SecWsdlService_Impl(args[0] + "?WSDL", info);
  ```

See "Invoking Web Services" in *Getting Started With WebLogic Web Services Using JAX-RPC* for general information about invoking a non-secured Web service.

# A

# Using Oracle Web Services Manager Security Policies

This appendix describes how to use Oracle Web Services Manager (WSM) WS-Security policies with WebLogic Server. This Appendix includes the following topics:


- "Overview" on page A-1
- "What Oracle WSM Security Policies Are Available?" on page A-4
- "What Oracle WSM WS-Security Policies Are Not Available?" on page A-10
- "Where are the Oracle WSM Policies Documented?" on page A-10
- "Adding Oracle WSM WS-Security Policies to Clients" on page A-14
- "Configuring Permission-Based Authorization Policies" on page A-15
- "Configuring the Credential Store Using WLST" on page A-16
- "Adding Oracle WSM WS-Security Policies to a Web Service" on page A-11
- "Policy Configuration Overrides for the Web Service Client" on page A-19
- "Creating Custom Assertions" on page A-20
- "Monitoring and Testing the Web Service" on page A-24


## A.1 Overview

Oracle Fusion Middleware 11*g* Release 1 (10.3.3) products install a portability layer on top of WebLogic Server that integrates Oracle Web Services Manager WS-Security policies into the WebLogic Server environment. This portability layer provides Oracle WSM WS-Security policies that you can use to protect WebLogic Server JAX-WS Web services and Web service clients.

You can use the Oracle WSM policies as an alternative to the WebLogic Server WS-Security policies for enforcing security for Web services. You can also create custom Oracle WSM WS-Security policies and use them with WebLogic Web services.

### A.1.1 When Should You Use Oracle WS-Security Policies?

You might want to use Oracle WSM WS-Security policies to protect JAX-WS Web services if you already use SOA, ADF, or Web Center applications elsewhere in your environment and you want to have a consistent security environment.

You should secure a WebLogic Server JAX-WS Web service with Oracle WSM WS-Security policies to have consistent and interoperable Web service security when

these Web services are used in conjunction with Oracle Fusion Middleware applications.

That is, you should secure WebLogic Server JAX-WS Web services with Oracle WSM WS-Security policies for use with applications that interact with Oracle Fusion Middleware applications, not with standalone WebLogic Server web service applications.

Consider the following scenarios:

- If you develop WebLogic Server JAX-WS Web services or clients that interact with SOA Composite Services, ADF Components, or WebCenter Services, then you should use the Oracle WSM WS-Security policies.
- If you develop only WebLogic Server native Java JAX-WS Web services, then you should use WebLogic Server's WS-Security policies.

Table A–1 lists policy selection guidelines for using the Oracle WSM policies.

In the table, the *@Policy* annotation applies to WebLogic Web policies, and the *@SecurityPolicy* annotation applies to Oracle WSM policies.

***Table A–1 Policy Selection Guidelines***

| @Policy | @Security Policy | Feature to be Implemented | Which Policies to Use |
|---|---|---|---|
| Yes | No | WSS1.0 with multiple must support key reference methods | Wssp1.2-2007-Wss1.0-UsernameToken-Plain-X509-Basic256.xml<br>Wssp1.2-2007-Saml1.1-SenderVouches-Wss1.0.xml |
| Yes | No | Username Token digest authentication | Wssp1.2-2007-Https-UsernameToken-Digest.xml<br>Wssp1.2-2007-Wss1.0-UsernameToken-Digest-X509-Basic256.xml<br>Wssp1.2-2007-Wss1.1-UsernameToken-Digest-X509-Basic256.xml |
| No | Yes | Kerberos Authentication | oracle/wss11_kerberos_token_client_policy<br>oracle/wss11_kerberos_token_service_policy<br>oracle/wss11_kerberos_token_with_message_protection_client_policy<br>oracle/wss11_kerberos_token_with_message_protection_service_policy |
| Yes | No | WSS 1.1 Derived Key | Wssp1.2-2007-Wss1.1-DK-X509-SignedEndorsing.xml<br>Wssp1.2-2007-Wss1.1-UsernameToken-Plain-DK.xml |

*Table A–1 (Cont.) Policy Selection Guidelines*

| @Policy | @Security Policy | Feature to be Implemented | Which Policies to Use |
|---|---|---|---|
| Yes | No | All Secure Conversations | Wssp1.2-2007-Wssc1.3-Bootstrap-Https.xml<br>Wssp1.2-2007-Wssc1.3-Bootstrap-Https-BasicAuth.xml<br>Wssp1.2-2007-Wssc1.3-Bootstrap-Https-ClientCertReq.xml<br>Wssp1.2-2007-Wssc1.3-Bootstrap-Https-UNT.xml<br>Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.0.xml<br>Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.1.xml |
| Yes | No | All SAML 2.0 scenarios | Wssp1.2-2007-Saml2.0-Bearer-Https.xml<br>Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml<br>Wssp1.2-2007-Saml2.0-HolderOfKey-Wss1.1-Asymmetric.xml<br>Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1-Asymmetric.xm |
| No | Yes | SAML 1.1 Bearer Confirmation for HTTPS | oracle/wss_saml_token_bearer_over_ssl_client_policy<br>oracle/wss_saml_token_bearer_over_ssl_service_policy |
| Yes | No | Encrypt before signing | Policy assertion <sp:EncryptBeforeSigning/> in both WSS10 or WSS11, Symmetric Binding or Asymmetric Binding, such as the following:<br>`<wsp:Policy xmlns:wsp="..." ><br><sp:SymmetricBinding><br><wsp:Policy><br>.. .<br><sp:EncryptBeforeSigning/><br>. . .<br></wsp:Policy><br></sp:SymmetricBinding><br>. . .<br></wsp:Policy>` |

*Table A–1 (Cont.) Policy Selection Guidelines*

| @Policy | @Security Policy | Feature to be Implemented | Which Policies to Use |
|---|---|---|---|
| Yes | No | Multiple policy alternatives | Policy assertion such as the following:<br>`<wsp:Policy xmlns:wsp="..." >`<br>`<wsp:ExactlyOne>`<br>`<wsp:All>`<br>`... ALternative 1`<br>`...`<br>`</wsp:All>`<br>`<wsp:All>`<br>`... ALternative 2`<br>`...`<br>`</wsp:All>`<br>`</wsp:ExactlyOne>`<br>`</wsp:Policy>` |

For non-security features, such as WS-RM and MTOM, use WebLogic Web services.

For specific policy instances, you can attach an Oracle WSM policy to the Web service client or service, and an Oracle WebLogic Server policy to the WebLogic Java EE Web service or client, and they will interoperate. The specific interoperability scenarios are described in "Interoperability with Oracle WebLogic Server 11g Web Service Security Environments" in *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

For these interoperability scenarios, you can use either Oracle WSM or WebLogic policies, depending on the following considerations:

- If additional non-standard policy assertions in the Oracle WSM policy are needed for configuration, then use the *@SecurityPolicy* annotation.

  Examples of these non-standard assertions might be as follows:

  ```
  <oralgp:Logging
  xmlns:oralgp="http://schemas.oracle.com/ws/2006/01/loggingpolicy"  . . .
     orawsp:category="security/logging">
          . . .
  </oralgp:Logging>
  ```

  or

  ```
  <orawsp:Config xmlns:orawsp="http://schemas.oracle.com/ws/2006/01/policy" . .
  .>
      <orawsp:PropertySet  . . .>
            . . .
      </orawsp:PropertySet>
  </orawsp:Config>
  ```

- If the application will be used to interoperate with existing WebLogic Web services or Microsoft Windows Communication Foundation (WCF)/.NET 3.5 Framework services, and the previously-mentioned non-standard policy assertions are not required, then use the *@Policy* annotation with the WebLogic policies.

## A.2 What Oracle WSM Security Policies Are Available?

The available Oracle WSM policies are shown in Table A–2.

***Table A–2 Available Oracle WSM WS-Security Policies***

| Service Policy Name | Client Policy Name | Description |
|---|---|---|
| oracle/binding_authorization_denyall_policy | None | This policy is a special case of simple role based authorization policy based upon the authenticated subject. This policy denies all users with any roles. This policy should follow an authentication policy where the subject is established. This policy can be attached to any SOAP-based endpoint. |
| oracle/binding_authorization_permitall_policy | None | This policy is a special case of simple role based authorization policy based upon the authenticated subject. This policy permits all users with any roles. This policy should follow an authentication policy where the subject is established. This policy can be attached to any SOAP-based endpoint. |
| oracle/binding_permission_authorization_policy | None | This policy is a special case of simple Permission based authorization policy based upon the authenticated subject. This policy checks if the subject has permission to invoke any operation on a Web service. This policy should follow an authentication policy where the subject is established. This policy can be attached to any SOAP-based endpoint. |
| oracle/wss10_message_protection_service_policy | oracle/wss10_message_protection_client_policy | This policy enforces message integrity and confidentiality for inbound SOAP requests in accordance with the WS-Security v1.0 standard. The messages are protected using WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanism for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore is configured through the security configuration. This policy does not authenticate or authorize the requestor. |
| oracle/wss10_saml_token_service_policy | oracle/wss10_saml_token_client_policy | This policy is not secure and is provided for demonstration purposes only. Although the SAML issuer name is present, the SAML token is not endorsed. Therefore, it is possible to spoof the message.<br>This policy authenticates users using credentials provided in SAML tokens in the WS-Security SOAP header. The credentials in the SAML token are authenticated against a SAML login module. This policy can be applied to any SOAP-based endpoint. |
| oracle/wss10_saml_token_with_message_integrity_service_policy | oracle/wss10_saml_token_with_message_integrity_client_policy | This policy enforces message-level integrity protection and SAML-based authentication for inbound SOAP requests in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, SHA-1 hashing algorithm for message integrity. The keystore is configured through the security configuration. It extracts the SAML token from the WS-Security binary security token, and uses those credentials to validate users against the configured identity store. |

*Table A–2 (Cont.) Available Oracle WSM WS-Security Policies*

| Service Policy Name | Client Policy Name | Description |
|---|---|---|
| oracle/wss10_saml_token_with_message_protection_service_policy | oracle/wss10_saml_token_with_message_protection_client_policy | This policy enforces message-level protection and SAML-based authentication for inbound SOAP requests in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. This policy uses subject Key Identifier (ski) reference mechanism for encryption key in the request and for both signature and encryption keys in the response. The keystore is configured through the security configuration. It extracts the SAML token from the WS-Security binary security token, and uses those credentials to validate users against the configured identity store. |
| oracle/wss10_username_id_propagation_with_msg_protection_service_policy | oracle/wss10_username_id_propagation_with_msg_protection_client_policy | This policy enforces message level protection (i.e., integrity and confidentiality) and identity propagation for inbound SOAP requests using mechanisms described in WS-Security 1.0. Message protection is provided using WS-Security 1.0's Basic128 suite of asymmetric key technologies. Specifically RSA key mechanisms for confidentiality, SHA-1 hashing algorithm for integrity and AES-128 bit encryption. The keystore is configured through the security configuration. Identity is set using username provided via the UsernameToken WS-Security SOAP header. The subject is established against the currently configured identity store. This policy can be applied to any SOAP based endpoint. |
| oracle/wss10_username_token_with_message_protection_service_policy | oracle/wss10_username_token_with_message_protection_client_policy | This policy enforces message-level protection (message integrity and confidentiality) and authentication for inbound SOAP requests in accordance with the WS-Security v1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanism for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore is configured through the security configuration. Authentication is enforced using credentials in the WS-Security UsernameToken SOAP header. Both plain text and digest mechanisms are supported. Credentials are authenticated against the configured credential store. This policy can be attached to any SOAP-based endpoint. |
| oracle/wss10_x509_token_with_message_protection_service_policy | oracle/wss10_x509_token_with_message_protection_client_policy | This policy enforces message-level protection and certificate-based authentication for inbound SOAP requests in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore is configured through the security configuration. The user is authenticated by extracting the certificate from the WS-Security binary security token and validating those credentials against the configured identity store. |

*Table A–2 (Cont.) Available Oracle WSM WS-Security Policies*

| Service Policy Name | Client Policy Name | Description |
|---|---|---|
| oracle/wss11_ kerberos_token_ service_ policy | oracle/wss11_ kerberos_token_ client_ policy | This policy is enforced in accordance with the WS-Security Kerberos Token Profile v1.1 standard. It extracts the Kerberos token from the SOAP header and authenticates the user. The container must have the Kerberos security infrastructure configured. This policy can be attached to any SOAP endpoint. This policy is compatible with MIT and Active Directory KDCs. |
| oracle/wss11_ kerberos_token_ with_message_ protection_ service_policy | oracle/wss11_ kerberos_token_ with_message_ protection_ client_policy | This policy is enforced in accordance with the WS-Security Kerberos Token Profile v1.1 standard. It extracts the Kerberos token from the SOAP header and authenticates the user, and it enforces message integrity and confidentiality using Kerberos keys. The container must have the Kerberos security infrastructure configured. This policy can be attached to any SOAP endpoint. This policy is compatible with MIT KDC only. |
| oracle/wss11_ message_ protection_ service_policy | oracle/wss11_ message_ protection_ client_policy | This policy enforces message integrity and confidentiality for inbound SOAP requests in accordance with the WS-Security 1.1 standard. The messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity and AES-128 bit encryption. The keystore is configured through the security configuration. |
| oracle/wss11_ saml_token_ with_message_ protection_ service_policy | oracle/wss11_ saml_token_ with_message_ protection_ client_policy | This policy enforces message-level protection (that is, message integrity and message confidentiality) and SAML-based authentication for inbound SOAP requests in accordance with the WS-Security 1.1 standard. Messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore is configured through the security configuration. It extracts the SAML token from the WS-Security binary security token, and uses those credentials to validate users against the configured identity store. This policy can be attached to any SOAP-based endpoint. |
| oracle/wss11_ username_ token_with_ message_ protection_ service_policy | oracle/wss11_ username_ token_with_ message_ protection_ client_policy | This policy enforces message-level protection (that is, message integrity and message confidentiality) and authentication for inbound SOAP requests in accordance with the WS-Security 1.1 standard. Messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore is configured through the security configuration. Credentials are provided through the UsernameToken WS-Security SOAP header. Both plain text and digest mechanisms are supported. The credentials are authenticated against the configured identity store. This policy can be attached to any SOAP-based endpoint. |

***Table A–2 (Cont.) Available Oracle WSM WS-Security Policies***

| Service Policy Name | Client Policy Name | Description |
|---|---|---|
| oracle/wss11_ x509_token_ with_message_ protection_ service_policy | oracle/wss11_ x509_token_ with_message_ protection_ client_policy | This policy enforces message-level protection and certificate-based authentication for inbound SOAP requests in accordance with the WS-Security 1.1 standard. Messages are protected using WS-Security's Basic 128 suite of symmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore is configured through the security configuration. The certificate is extracted from the WS-Security binary security token header, and the credentials in the certificate are validated against the configured identity store. |
| oracle/wss_ oam_token_ service_policy | oracle/wss_ oam_token_ client_policy | This policy uses the credentials in the WS-Security header's binary security token to authenticate users against the Oracle Access Manager identity store. This policy can be attached to any SOAP endpoint. |
| oracle/wss_ saml_token_ bearer_over_ ssl_service_ policy | oracle/wss_ saml_token_ bearer_over_ ssl_client_policy | This policy authenticates users using credentials provided in SAML tokens with confirmation method 'Bearer' in the WS-Security SOAP header. The credentials in the SAML token are authenticated against a SAML login module. The policy verifies that the transport protocol provides SSL message protection. This policy can be applied to any SOAP-based endpoint. |
| oracle/wss_ saml_token_ over_ssl_ service_policy | oracle/wss_ saml_token_ over_ssl_ client_policy | This policy authenticates users using credentials provided in SAML tokens in the WS-Security SOAP header. The credentials in the SAML token are authenticated against a SAML login module. The policy verifies that the transport protocol provides SSL message protection. This policy can be applied to any SOAP-based endpoint. |
| oracle/wss_ username_ token_over_ssl_ service_policy | oracle/wss_ username_ token_over_ssl_ client_policy | This policy uses the credentials in the UsernameToken WS-Security SOAP header to authenticate users against the configured identity store. Both plain text and digest mechanisms are supported. The policy verifies that the transport protocol provides SSL message protection. This policy can be attached to any SOAP-based endpoint. |
| oracle/wss_ username_ token_service_ policy | oracle/wss_ username_ token_client_ policy | This policy is not secure and is provided for demonstration purposes only. The password is sent in clear text.<br>This policy uses the credentials in the UsernameToken WS-Security SOAP header to authenticate users against the configured identity store. Both plain text and digest mechanisms are supported. The policy verifies that the transport protocol provides SSL message protection. This policy can be attached to any SOAP-based endpoint. |

*Table A–2 (Cont.) Available Oracle WSM WS-Security Policies*

| Service Policy Name | Client Policy Name | Description |
|---|---|---|
| oracle/wss10_ saml_hok_ token_with_ message_ protection_ service_policy | oracle/wss10_ saml_hok_ token_with_ message_ protection_ client_policy | This policy enforces message-level protection and SAM holder of key based authentication for inbound SOAP requests in accordance with the WS-Security 1.0 standard. It uses WS-Security's Basic 128 suite of asymmetric key technologies, specifically RSA key mechanisms for message confidentiality, SHA-1 hashing algorithm for message integrity, and AES-128 bit encryption. The keystore is configured through the security configuration. It extracts the SAML token from the WS-Security binary security token, and uses those credentials to validate users against the configured identity store. |

## A.2.1 Is There Compatibility Between WebLogic Policies and Oracle WSM Policies?

A subset of WebLogic Web service policies interoperate with Oracle WSM policies.

That is, for specific policy instances, you can attach an Oracle WSM policy to the Web service client or service, and an Oracle WebLogic Server policy to the WebLogic Java EE Web service or client, and they will interoperate.

The specific interoperability scenarios are described in *Interoperability with Oracle WebLogic Server 11g Web Service Security Environments* in *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

This release of WebLogic Server includes the policies shown in Table A–3 for interoperability with Oracle WSM.

*Table A–3 Interoperability WS-Security Policies*

| Policy Name | Description |
|---|---|
| Wssp1.2-2007-Saml1.1-HolderOfKey-Wss1.0-Basic128.xml | This policy provides similar security features to that of Wssp1.2-2007-Saml1.1-HolderOfKey-Wss1.0.xml, including SAML token for authentication with holder of key confirmation method, in which the key inside the SAML Token is used for the signature. It requires using Basic128 algorithm suite (AES128 for encryption) instead of Basic256 algorithm suite (AES256). |
| Wssp1.2-wss11_saml_token_with_message_protection_owsm_policy.xml | This policy provides similar security features to that of Wssp1.2-2007-Saml1.1-SenderVouches-Wss1.1.xml, including a SAML token for authentication with the sender vouches confirmation method, signed and encrypted on both request and response with WSS1.1 X509 symmetric binding.<br>It endorses with the sender's X509 certificate, and message signature is protected. It requires the use of the Basic128 algorithm suite (AES128 for encryption) instead of the Basic256 algorithm suite (AES256). |
| Wssp1.2-wss10_saml_token_with_message_protection_owsm_policy.xml | This policy provides similar security features to that of Wssp1.2-2007-Saml1.1-SenderVouches-Wss1.0.xml, including SAML token for authentication with sender vouches confirmation method, signed with the client's private key. It requires using Basic128 algorithm suite (AES128 for encryption) instead of Basic256 algorithm suite (AES256). It also uses the direct key reference that includes public certificates. |
| Wssp1.2-2007-Saml1.1-SenderVouches-Https.xml | Two-way SSL that uses SAML 1.1 token with sender vouches confirmation method for authentication. It requires client certificates, and the recipient checks for the initiator's public certificate. |

*Table A–3 (Cont.) Interoperability WS-Security Policies*

| Policy Name | Description |
|---|---|
| Wssp1.2-wss10_x509_token_with_message_protection_owsm_policy.xml | This policy provides similar security features to that of Wssp1.2-2007-Wss1.0-X509-Basic256.xml for mutual authentication with X.509 Certificates. It requires using Basic128 algorithm suite (AES128 for encryption) instead of Basic256 algorithm suite (AES256). It also usesthe direct key reference that includes public certificates. |
| Wssp1.2-2007-Wss1.1-EncryptedKey-Basic128.xml | This policy provides similar security features to that of Wssp1.2-Wss1.1-EncryptedKey.xml. The policy requires the message to be encrypted and signed without X509 certificate from the client side. It is used for anonymous authentication. |
| Wssp1.2-wss11_x509_token_with_message_protection_owsm_policy.xml | This policy provides similar security features to that of Wssp1.2-Wss1.1-EncryptedKey-X509-SignedEndorsing.xml. It endorses with the sender's X509 certificate, and the message signature is protected. It requires the use of the Basic128 algorithm suite (AES128 for encryption) instead of the Basic256 algorithm suite (AES256). |
| Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml | This policy provides similar security features to that of Wssp1.2-Wss1.1-UsernameToken-Plain-X509-Basic256.xml , which has WSS 1.1 X509 with asymmetric binding and authentication with plain-text Username Token. It requires using Basic128 algorithm suite (AES128 for encryption) instead of Basic256 algorithm suite (AES256). |
| Wssp1.2-wss10_username_token_with_message_protection_owsm_policy.xml | This policy provides similar security features to that of Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic256.xml, including encrypted plain text password for authentication, signed with the client's private key. It requires using Basic128 algorithm suite (AES128 for encryption) instead of Basic256 algorithm suite (AES256). It also uses the direct key reference that includes public certificates. |

## A.3 What Oracle WSM WS-Security Policies Are Not Available?

If you are already familiar with the wide range of Oracle WSM WS-Security policies included in Oracle Fusion Middleware 11*g* Release 1 (10.3.3) products, note that the following Oracle WSM WS-Security policies are not currently supported with WebLogic Server JAX-WS:

- Log policies
- WS-Addressing Policies
- MTOM Policies
- Reliable Message Policies
- HTTP-based authentication:
  - oracle/wss_http_token_client_policy
  - oracle/wss_http_token_service_policy
  - oracle/wss_http_token_over_ssl_client_policy
  - oracle/wss_http_token_over_ssl_service_policy

## A.4 Where are the Oracle WSM Policies Documented?

The Oracle WSM policies are documented in the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

The description for each policy is repeated in Table A–2 for your convenience, but see the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services* for complete information regarding these policies.

## A.5 Adding Oracle WSM WS-Security Policies to a Web Service

The Oracle WSM WS-Security policy attachment model is similar to that of the WebLogic Web service policies:

- Through policy annotations at design time
- Via the Administration Console at runtime
- Via Fusion Middleware Control. See *Oracle Fusion Middleware Security and Administrator's Guide for Web Services* for information about how to do this.

You can attach only one type of security policy to a Web service, either WebLogic Server security policies or Oracle WSM policies. You cannot attach both WebLogic Server policies and Oracle WSM policies to the same Web service, through either the annotation mechanism, the Administration Console, Fusion Middleware Control, or a combination of the three.

You can attach an Oracle WSM WS-Security policy only to a JAX-WS Web service; you cannot attach this type of policy to a JAX-RPC Web service.

### A.5.1 SecurityPolicy and SecurityPolicies Annotations

The Oracle WSM policies use unique *@SecurityPolicy* (single policy) and *@SecurityPolicies* (multiple policies) annotations that have similar syntax and semantics to the existing *@Policies* and *@Policy* annotations used by the WebLogic Server WS-Security policies, with the following exceptions:

- @SecurityPolicy and @SecurityPolicies can be applied only at the class level.
- "Direction" is not used in the @SecurityPolicy annotation.

For example:

***Example A–1 Attaching a Policy Using SecurityPolicy Annotation***

```
@SecurityPolicies({
@SecurityPolicy(uri=
"policy:oracle/wss10_username_token_with_message_protection_server_policy"),
@SecurityPolicy(uri=
"policy:oracle/authorization_policy")})
```

To add Oracle OWSM WS-Security policies via JWS annotations at design time:

1. Update your JWS file, adding `@SecurityPolicy` and `@SecurityPolicies` JWS annotations to specify the predefines Oracle OWSM WS-Security policy files that are attached to the entire Web service.
2. Recompile and redeploy your Web service as part of the normal iterative development process.

   See "Developing WebLogic Web Services" in *Getting Started With WebLogic Web Services Using JAX-WS* .
3. Using the Administration Console, create users for authentication in your security realm.

   See *Securing WebLogic Resources Using Roles and Policies*.

4. Update your client application by adding the Java code to invoke the message-secured Web service, as described in "Adding Oracle WSM WS-Security Policies to Clients" on page A-14.
5. Recompile your client application.

   See *Getting Started With WebLogic Web Services Using JAX-WS*.

## A.5.2 Configuring Oracle WSM Security Policies in Administration Console

Attaching one of the Oracle WSM policies to a deployed Web service at runtime in the Administration Console is similar to attaching WebLogic Server policies, as described in Chapter 2, "Configuring Message-Level Security".

You can choose to not use *@SecurityPolicy* or *@SecurityPolicies* annotations at all in your JWS file and associate policy files only at runtime using the Administration Console. Or, you can specify some policy files using the annotations and then associate additional ones at runtime.

If you associate a policy file using the JWS annotations, you can then delete this association at runtime using the Administration Console.

At runtime, the Administration Console allows you to associate as many policy files as you want with a Web service and its operations, even if the policy assertions in the files contradict each other or contradict the assertions in policy files associated with the JWS annotations. It is up to you to ensure that multiple associated policy files work together. If any contradictions do exist, WebLogic Server returns a runtime error when a client application invokes the Web service operation.

There is no policy validation. Only the following specific combinations are valid:

- Only one Management policy can be attached to a policy subject.
- Only one Security policy with subtype authentication can be attached to a subject.
- Only one Security policy with subtype message protection can be attached to a subject.
- Only one security policy with subtype authorization can be attached to a subject.

> **Note:** There may be either one or two security policies attached to a policy subject. A security policy can contain an assertion that belongs to the authentication or message protection subtype categories, or an assertion that belongs to both subtype categories. The second security policy contains an assertion that belongs to the authorization subtype.

- If an authentication policy and an authorization policy are both attached to a policy subject, the authentication policy must precede the authorization policy.

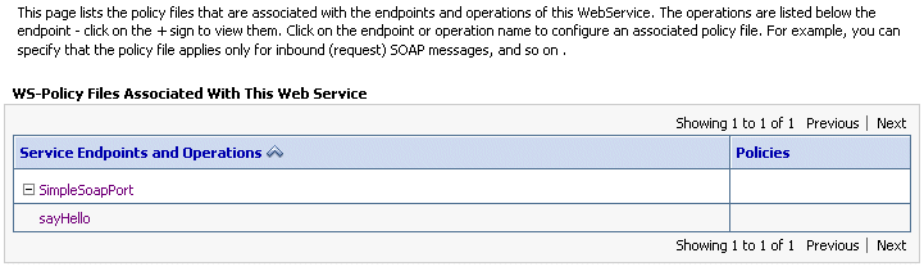
Perform the following steps to attach an Oracle WSM WS-Security policy via the Administration Console:

1. Using the Administration Console, create the default Web service security configuration, which must be named `default_wss`. The default Web service security configuration is used by *all* Web services in the domain unless they have been explicitly programmed to use a different configuration.

   See "Create a Web Service Security Configuration" in the *WebLogic Server Administration Console Online Help*.

2. From the Summary of Deployments page, select the application for which you want to secure a Web service.
3. Click the plus sign (+) to expand the application. Select the Web service you want to secure.
4. Select the **Configuration** page.
5. Select the **WS-Policy** page.
6. Select the Web service endpoint, as shown in Figure A–1. You can attach Oracle WSM WS-Security policies only at the class/port level.

***Figure A–1 Service Endpoints for the Web Service***




7. Select OWSM, as shown in Figure A–2.

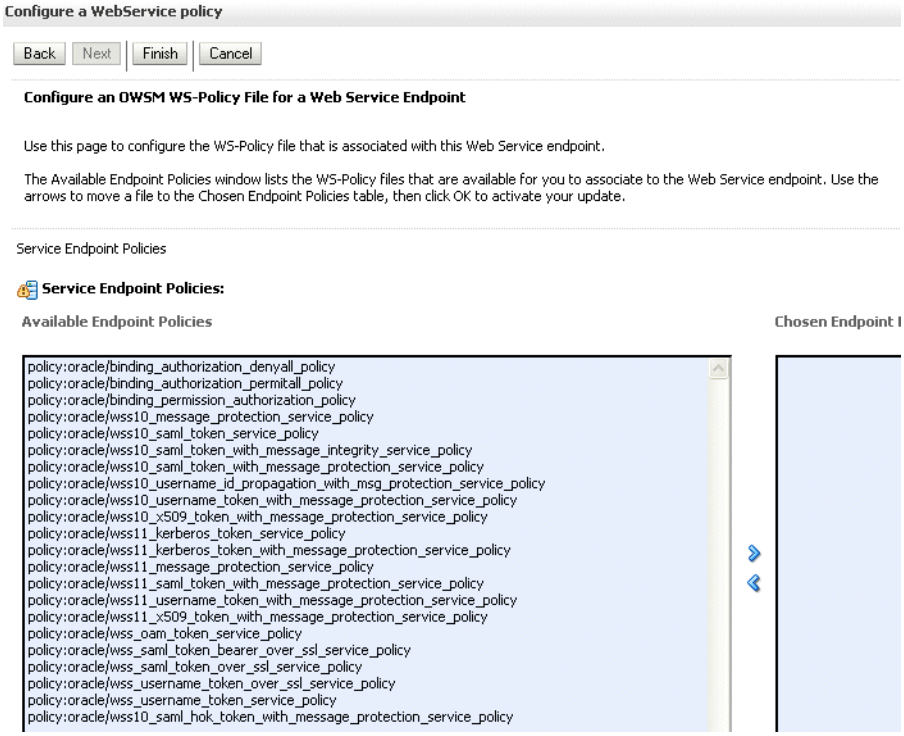
***Figure A–2 Selecting the Oracle WSM WS-Security Policy Type***


8. If you had instead mistakenly selected a particular Web service operation, note that you are not presented with the policy choice screen, as shown in Figure A–3. Click **Cancel** to start over.

***Figure A–3 WebLogic Server Policy Page***


9. Select the Oracle WSM WS-Security policies that you want to attach to this Web service, and use the control to move them into the Chosen Endpoint Policies box, as shown in Figure A–4. Click Finish when done.

***Figure A–4 Selecting From the Available Oracle WSM WS-Security Policies***




10. Save the deployment plan.
11. If the change is not automatically activated as indicated in the WebLogic Server change message, restart the deployed application to reflect the new deployment plan.

## A.6 Adding Oracle WSM WS-Security Policies to Clients

You use *weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature* class to attach a single policy, and *weblogic.wsee.jws.jaxws.owsm.SecurityPoliciesFeature* to attach multiple policies. The available client policies are listed in Table A–2.

Consider the following examples:

**Note:** The `oracle/wss_username_token_client` policy shown in this example is not secure and is provided for demonstration purposes only. The password is sent in clear text.

***Example A–2 Using SecurityPolicyFeature***

```
JAXWSService jaxWsService = new JAXWSService ();
weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature securityFeature = new
weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature {
new weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature("policy:oracle/wss_
username_token_client_policy") };

JAXWSServicePort  port =  jaxWsService.getJaxWsServicePort(securityFeature);
```

***Example A–3 Using SecurityPoliciesFeature***

```
weblogic.wsee.jws.jaxws.owsm.SecurityPoliciesFeature
securityFeature = new weblogic.wsee.jws.jaxws.owsm.SecurityPoliciesFeature
{
new weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature("policy:oracle/wss_
username_token_client_policy") ,
new weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature("policy:oracle/log_
policy")};
```

### A.6.1 Associating a Policy File with a Client Application: Main Steps

The following procedure describes the high-level steps to associate an Oracle WSM WS-Security policy file with the client application that invokes a Web service operation.

It is assumed that you have created the client application that invokes a deployed Web service, and that you want to update it by associating a client-side policy file. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `clientgen` Ant task.

See "Invoking Web Services" in *Getting Started With WebLogic Web Services Using JAX-WS*.

1. Select the client-side security policy files you want to attach from Table A–2.
2. Update the `build.xml` file that builds your client application.
3. Update your Java client application.
4. Rebuild your client application by running the relevant task. For example:

   ```
   prompt> ant build-client
   ```

When you next run the client application, it will load the policy files that the Web service client runtime uses to enable security for the SOAP request message.

## A.7 Configuring Permission-Based Authorization Policies

The permission-based policy *oracle/binding_permission_authorization_policy* provides a permission-based authorization policy based on the authenticated subject.

The policies ensure that the subject has permission to perform the operation. To do this, the Oracle WSM Authorization Policy executor leverages Oracle Platform Security Services (OPSS) to check if the authentiated subject has been granted

*oracle.wsm.security.WSFunctionPermission* (or whatever permission class is specified in *Permission Check Class*) using the *Resource Pattern* and *Action Pattern* as parameters.

*Resource Pattern* and *Action Pattern* are used to identify if the authorization assertion is to be enforced for this particular request. Access is allowed if the authenticated subject has been granted *WSFunctionPermission*.

You can grant the *WSFunctionPermission* permission to a user, a group, or an application role. If you grant *WSFunctionPermission* to a user or group it will apply to all applications that are deployed in the domain.

To do this, edit the *system-jazn-data.xml* file under *<domain-home>/config/fmwconfig* to grant the *WSFunctionPermission* permission to the user, group, or application that will attempt to authenticate to the Web service, as shown in Example A–4.

In the example, the user who has the ApplicationRole must be a valid WebLogic Server user.

***Example A–4 Editing the system-jazn-data.xml File to Grant Permission***

```
:
<jazn-policy>
   <grant>
       <grantee>
            <display-name>myPolicy</display-name>
               <principals>
                  <principal>

<class>oracle.security.jps.service.policystore.ApplicationRole</class>
                     <name>testapp</name>
                  </principal>
               </principals>
       </grantee>
       <permissions>
         <permission>
            <class>oracle.wsm.security.WSFunctionPermission</class>
            <name>*</name>
            <actions>echo1</actions>
         </permission>
       </permissions>
   </grant>
</jazn-policy>
:
```

## A.8 Configuring the Credential Store Using WLST

In order to sign and encrypt SOAP messages you must first create and configure the Web Services Manager Keystore for a WebLogic domain. This keystore is used to store public and private keys for SOAP messages within the WebLogic Domain.

> **Note:** The Web services manager runtime does **not** use the WebLogic Server keystore that is used for SSL.

The signature and encryption keys are used to sign, verify, encrypt, and decrypt the SOAP messages.

The keystore configuration is domain wide: all Web services and Web service clients in the domain use this keystore.

To set up the keystore used by Web Services Manager, follow these steps:

1. Create the Keystore, as described in "How to Create and Use a Java Keystore" on page A-17
2. Use WLST to configure the credential store.

## A.8.1 How to Create and Use a Java Keystore

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.

### A.8.1.1 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/javase/6/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" -keystore test.jks

   Allow keytool to prompt you for the password. DSA key is not supported. Make sure you pass the parameter " -keyalg RSA " in the command.

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. Allow keytool to prompt you for the password.

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

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. Allow keytool to prompt you for the password.

   keytool -certreq -alias test -sigalg "RSAwithSHA1" -file certreq_file -storetype jks -keystore test.jks

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

## A.8.2 Manage the Credential Store Framework

Oracle WSM uses the Credential Store Framework (CSF) to manage the credentials in a secure form. The CSF configuration is maintained in the *jps-config.xml* file under *<domain-home>/config/fmwconfig*.

CSF provides a way to store, retrieve, and delete credentials for a Web service and other applications. For example, the *oracle/wss_username_token_client_policy* policy includes the *csf-key* property, with a default value of *basic.credentials*. This credential is stored in the CSF.

A password credential can store a username and password. A generic credential can store any credential object.

Each credential is stored in the store using the alias and the key pair. The alias, called the *map name*, is a logical name that defines a group of various keys and one credential associated with that key. That is, the map name and the key name are combined to make a primary key in the store.

Typically the map name is the name of the application or component to make it easy to identify. By convention, Web services should use *oracle.wsm.security*.

Consider the keystore example shown in Example A–5.

***Example A–5 Example Keystore from jps-config.xml***

```
serviceInstance name="keystore" provider="keystore.provider"
location="./default-keystore.jks">
            <description>Default JPS Keystore Service</description>
            <property name="keystore.type" value="JKS"/>
            <property name="keystore.csf.map" value="oracle.wsm.security"/>
            <property name="keystore.pass.csf.key" value="keystore-csf-key"/>
            <property name="keystore.sig.csf.key" value="sign-csf-key"/>
            <property name="keystore.enc.csf.key" value="enc-csf-key"/>
        </serviceInstance>
```

The keystore provider location must identify the location of the keystore you created in "How to Create Private Keys and Load Trusted Certificates" on page A-17.

The *keystore.csf.map* property points to the CSF map that contains the CSF aliases. In this case *keystore.csf.map* is defined as the recommended name *oracle.wsm.security*, but it can be any value.

The *keystore.pass.csf.key* property points to the CSF alias that is mapped to the username and password of the keystore. Only the password is used; username is redundant in the case of the keystore.

The *keystore.sig.csf.key* property points to the CSF alias that is mapped to the username and password of the private key that is used for signing.

The *keystore.enc.csf.key* property points to the CSF alias that is mapped to the username and password of the private key that is used for decryption.

### A.8.2.1 How to Update Your Credential Store Using WLST

Perform the following steps to update the credential store:

1. Start WLST.
2. Connect to the running server.

```
wls:/offline> connect()
Please enter your username [weblogic] :weblogic
Please enter your password [...] :
Please enter your server URL [t3://localhost:7001] :t3://localhost:7101
Connecting to t3://localhost:7101 with userid weblogic ...
Successfully connected to Admin Server 'DefaultServer' that belongs to domain
'DefaultDomain'.

Warning: An insecure protocol was used to connect to the
server. To ensure on-the-wire security, the SSL port or
Admin port should be used instead.

wls:/DefaultDomain/serverConfig>
```

3. Create an entry in the credential store.

```
wls:/DefaultDomain/serverConfig> createCred(map="oracle.wsm.security",
key="basic.credentials", user="john", desc="test-key")
```

   This adds the key *basic.credentials* to the map *oracle.wsm.security* with the username as *john*. This entry will be stored inside <your domain>/config/fmwconfig/cwallet.sso.

4. Optionally, list the value of the key.

```
wls:/DefaultDomain/serverConfig> listCred(map="oracle.wsm.security",
key="basic.credentials")
```

## A.9 Policy Configuration Overrides for the Web Service Client

You have the option to override the default settings of an Oracle WSM WS-Security policy programmatically at design time, as described in the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

> **Note:** You cannot override a server-side policy, either through annotations, through Fusion Middleware Control, or through the Administration Console.

For example, you might set the *csf key*, specify the recipient key alias, and so forth. To do this, you use the standard JAX-WS RequestContext, as shown in Example A–6.

***Example A–6 Web Service Client Programmatic Overrides***

```
import oracle.wsm.security.util.SecurityConstants.ClientConstants;
...
public class MyClientJaxWs {
```

```
    public void doWork(String[] args) {
        try {
            RequestContext context = …
   // the user code does this for passing the Policy Names that are used for the
call

JAXWSService jaxWsService = new JAXWSService ();
weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature []
securityFeature = new weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature [] {
new weblogic.wsee.jws.jaxws.owsm.SecurityPolicyFeature("policy:oracle/wss_
username_token_client_policy") };

JAXWSServicePort  port =  jaxWsService.getJaxWsServicePort(securityFeature);

       context.put(BindingProvider.USERNAME_PROPERTY, getCurrentUsername() );
       context.put(BindingProvider.PASSWORD_PROPERTY, getCurrentPassword() );
       context.put(ClientConstants.WSSEC_KEYSTORE_LOCATION, "c:/mykeystore.jks");
       context.put(ClientConstants.WSSEC_KEYSTORE_PASSWORD, "keystorepassword" );
       context.put(ClientConstants.WSSEC_KEYSTORE_TYPE, "JKS" );
       context.put(ClientConstants.WSSEC_SIG_KEY_ALIAS, "your signature alias" );
       context.put(ClientConstants.WSSEC_SIG_KEY_PASSWORD, "your signature
password" );
       context.put(ClientConstants.WSSEC_ENC_KEY_ALIAS, "your encryption alias" );
       context.put(ClientConstants.WSSEC_ENC_KEY_PASSWORD, "your encryption
password" );
       System.out.println(proxy.myOperation("MyInput"));
…
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## A.10 Creating Custom Assertions

This section describes how to create custom assertions. It includes the following sections:

- Overview of Custom Assertion Creation
- Step 1: Create the Custom Policy File
- Step 2: Add the Custom Policy to the Policy Store
- Step 3: Create the Custom Assertion Class
- Step 4: Create the Custom Assertion Class JAR File
- Step 5: Update Your CLASSPATH
- Step 6: Develop and Deploy a JAX-WS Web Service
- Step 7: Attach the Custom Policy to the JAX-WS Web Service

### A.10.1 Overview of Custom Assertion Creation

If the predefined assertion templates, defined in "Predefined Assertion Templates" in *Security and Administrator's Guide for Web Services*, do not fit your needs, you can create your own custom assertions.

To create a custom assertion, you need to create the following files:

- Custom assertion class—Implements the Java class and its parsing and enforcement logic.
- Custom policy file—Enables you to define the bindings for and configure the custom assertion.

You package custom assertion class and policy file as JAR files and make the JAR files available in the CLASSPATH for your domain. Then, you import the custom policy file and attach it to your Web service or client, as required.

The following sections describe each step in the process.

## A.10.2 Step 1: Create the Custom Policy File

Create the custom policy file to define the bindings for and configure the custom assertion. "Schema Reference for Custom Assertions" in *Security and Administrator's Guide for Web Services* describes the schema that you can use to construct your custom policy file and custom assertion.

The following example defines the oracle/ip_assertion_policy custom policy file. The assertion defines a comma-separated list of IP addresses that are valid for a request.

The executor class is specified as follows:
`<orawsp:Implementation>sampleassertion.IpAssertionExecutor</orawsp:Implementation>`

***Example A–7 Example Custom Policy File***

```
<?xml version = '1.0' encoding = 'UTF-8'?>

<wsp:Policy xmlns="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:orasp="http://schemas.oracle.com/ws/2006/01/securitypolicy"
   orawsp:status="enabled"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" orawsp:category="security"
   orawsp:attachTo="binding.server" wsu:Id="ip_assertion_policy"
   xmlns:orawsp="http://schemas.oracle.com/ws/2006/01/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   wsp:Name="oracle/ip_assertion_policy">
      <orasp:ipAssertion orawsp:Silent="true" orawsp:Enforced="true"
         orawsp:name="WSSecurity IpAssertion Validator" orawsp:category="security/authentication">
            <orawsp:bindings>
                  <orawsp:Implementation>sampleassertion.IpAssertionExecutor</orawsp:Implementation>
                      <orawsp:Config orawsp:name="ipassertion" orawsp:configType="declarative">
                          <orawsp:PropertySet orawsp:name="valid_ips">
                               <orawsp:Property orawsp:name="valid_ips" orawsp:type="string"
                                orawsp:contentType="constant">
                                    <orawsp:Value>127.0.0.1,192.168.1.1</orawsp:Value>
                               </orawsp:Property>
                          </orawsp:PropertySet>
                      </orawsp:Config>
             </orawsp:bindings>
      </orasp:ipAssertion>
</wsp:Policy>
```

## A.10.3 Step 2: Add the Custom Policy to the Policy Store

Once you create your custom policy file (in Step 1), you need to add it to the policy store. You can import the policy using Fusion Middleware Control, as described in "Importing Web Service Policies" in *Security and Administrator's Guide for Web Services*.

## A.10.4 Step 3: Create the Custom Assertion Class

Create the custom assertion class to execute and validate the logic of your policy assertion. The custom assertion class must extend `oracle.wsm.policyengine.impl.AssertionExecutor`.

When building the custom assertion class, ensure that the following JAR files are in your CLASSPATH: wsm-policy-core.jar and wsm-agent-core.jar.

The following example shows a custom assertion executor that can be used to validate the IP address of the request. If the IP address of the request is invalid, a `FAULT_FAILED_CHECK` exception is thrown.

For more information about the APIs that are available to you for developing your own custom assertion class, see the *Java API Reference for Oracle Web Services Manager*.

***Example A–8 Example Custom Assertion Class***

```
package sampleassertion;

import oracle.wsm.common.sdk.IContext;
import oracle.wsm.common.sdk.IMessageContext;
import oracle.wsm.common.sdk.IResult;
import oracle.wsm.common.sdk.Result;
import oracle.wsm.common.sdk.WSMException;
import oracle.wsm.policy.model.IAssertionBindings;
import oracle.wsm.policy.model.IConfig;
import oracle.wsm.policy.model.IPropertySet;
import oracle.wsm.policy.model.ISimpleOracleAssertion;
import oracle.wsm.policy.model.impl.SimpleAssertion;
import oracle.wsm.policyengine.impl.AssertionExecutor;

public class IpAssertionExecutor extends AssertionExecutor {
    public IpAssertionExecutor() {
    }
    public void destroy() {
    }

    public void init(oracle.wsm.policy.model.IAssertion assertion,
                     oracle.wsm.policyengine.IExecutionContext econtext,
                     oracle.wsm.common.sdk.IContext context) {
        this.assertion = assertion;
        this.econtext = econtext;
    }
    public oracle.wsm.policyengine.IExecutionContext getExecutionContext() {
        return this.econtext;
    }
    public boolean isAssertionEnabled() {
        return ((ISimpleOracleAssertion)this.assertion).isEnforced();
    }
    public String getAssertionName() {
        return this.assertion.getQName().toString();
    }

    /**
     * @param context
     * @return
     */
    public IResult execute(IContext context) throws WSMException {
        try {
            IAssertionBindings bindings =
                ((SimpleAssertion)(this.assertion)).getBindings();
            IConfig config = bindings.getConfigs().get(0);
            IPropertySet propertyset = config.getPropertySets().get(0);
            String valid_ips =
```

```
                propertyset.getPropertyByName("valid_ips").getValue();
            String ipAddr = ((IMessageContext)context).getRemoteAddr();
            IResult result = new Result();
            if (valid_ips != null && valid_ips.trim().length() > 0) {
                String[] valid_ips_array = valid_ips.split(",");
                boolean isPresent = false;
                for (String valid_ip : valid_ips_array) {
                    if (ipAddr.equals(valid_ip.trim())) {
                        isPresent = true;
                    }
                }
                if (isPresent) {
                    result.setStatus(IResult.SUCCEEDED);
                } else {
                  result.setStatus(IResult.FAILED);
                  result.setFault(new WSMException(WSMException.FAULT_FAILED_CHECK));
                }
            } else {
                result.setStatus(IResult.SUCCEEDED);
            }
            return result;
        } catch (Exception e) {
            throw new WSMException(WSMException.FAULT_FAILED_CHECK, e);
        }
    }

    public oracle.wsm.common.sdk.IResult postExecute(oracle.wsm.common.sdk.IContext p1) {
        IResult result = new Result();
        result.setStatus(IResult.SUCCEEDED);
        return result;
    }
}
```

## A.10.5 Step 4: Create the Custom Assertion Class JAR File

Create a JAR file containing the custom assertion class file created in Step 3. You can use Oracle JDeveloper, other IDE, or the jar tool to generate the JAR file.

The JAR file should contain the class file and manifest file. For example:

```
$ jar -tvf assertionjar.jar
    25 Tue Oct 06 14:40:10 PDT 2009 META-INF/MANIFEST.MF
  3558 Tue Oct 06 14:40:10 PDT 2009 sampleassertion/IpAssertionExecutor.class
```

## A.10.6 Step 5: Update Your CLASSPATH

You need to add the following files to your CLASSPATH:

- Custom policy JAR file created in Step 2.
- Custom assertion JAR file, created in Step 4, so that the custom assertion execution class is available in the server environment.

Add the JAR files to your CLASSPATH by performing the following steps:

1. Stop the WebLogic Server.

   For more information on stopping the WebLogic Server, see *Managing Server Startup and Shutdown for Oracle WebLogic Server*.

2. Copy the custom policy JAR and custom assertion JAR files created in Steps 2 and 4 to the following directory: $DOMAIN_HOME/lib.

3. Restart the WebLogic Server.

For more information on restarting the WebLogic Server, see *Managing Server Startup and Shutdown for Oracle WebLogic Server*.

### A.10.7 Step 6: Develop and Deploy a JAX-WS Web Service

Develop and deploy a JAX-WS Web service. The following provides a very simple example of a JAX-WS Web service:

```
package project1;
import javax.jws.WebService;
@WebService
public class Class1 {
    public Class1() {
        super();
    }

    public String echo1() {
        return "one";
    }
}
```

For more information about developing and deploying JAX-WS Web services, see:

- *Getting Started Using JAX-WS Web Services for Oracle WebLogic Server*
- *Programming Advanced Features of JAX-WS Web Services for Oracle WebLogic Server*
- "Developing with Web Services" in the Oracle JDeveloper Online Help

### A.10.8 Step 7: Attach the Custom Policy to the JAX-WS Web Service

Attach the custom policy to the JAX-WS Web service using the Oracle WebLogic Administration Console. For detailed steps, see "Associating Policy Files at Runtime Using the Administration Console" in *Securing Web Services for Oracle WebLogic Server*.

## A.11 Monitoring and Testing the Web Service

You can use either the Administration Console or  Fusion Middleware Control to monitor and test a WebLogic JAX-WS Web service that is protected with an Oracle WSM WS-Security policy.

To test or monitor the Web service from the Administration Console, follow these steps:

1. From the Summary of Deployments page, select the application for which you want to test the  a Web service.
2. From the settings page, select the **Testing** tab to test the Web service, or **Monitoring** to monitor it.

To test or monitor the Web service from Fusion Middleware Control, see *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.