Oracle® Fusion Middleware

Securing WebLogic Web Services for Oracle WebLogic Server

15c (15.1.1.0.0)
G31688-01
October 2025

Oracle Fusion Middleware Securing WebLogic Web Services for Oracle WebLogic Server, 15c (15.1.1.0.0)

G31688-01

Contents

3 Configuring Transport-Level Security

Preface

This document describes securing WebLogic web services for Oracle WebLogic Server 15c.

Audience

This documentation is a resource for security software developers who secure WebLogic web services for Oracle WebLogic Server that includes configuring transport- and message-level security.

Documentation Accessibility

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

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

New and Changed WebLogic Server Features

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

Conventions

The following text conventions are used in this document:

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

1

Overview of Web Services Security

The chapter describes how to configure security for WebLogic web services. This chapter includes the following sections:

For definitions of unfamiliar terms found in this and other books, see the *Glossary*.

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.

Thread Safety

JAX-WS clients are not thread safe.

See Are JAX-WS client proxies thread safe? for more information and workarounds regarding JAX-WS thread safety.

2

Configuring Message-Level Security

The chapter describes how to configure message-level security for your WebLogic web service using Jakarta XML Web Services (JAX-WS).
This chapter includes the following sections:

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 Configuring Simple Message-Level Security 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 Associating Policy Files at Runtime for details.

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

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).

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.

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 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 Unsupported WS-SecurityPolicy 1.2 Assertions.

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.

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 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.

This section describes the following topics:

- Using Policy Files With JAX-WS
- WS-Policy Namespace
- WS-SecurityPolicy Namespace
- Version-Independent Policy Supported
- Using the SHA-256 Secure Hash Algorithm
- Using the Extended Algorithm Suite (EAS)

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.

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

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 Specification Backward Compatibility.
- Derived Key using different WSSC versions (200502, 1.3).

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 the following **Version-Independent Matrix** table.

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
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.

Using the SHA-256 Secure Hash Algorithm

The WebLogic Server web service security policies support both the SHA-1 and much stronger SHA-2 (SHA-256) secure hash algorithms for hashing digital signatures. In addition to the SHA-2 secure hash algorithm, FIPS 140-2 mode requires a stronger digital signature method algorithm which is supported by extended algorithm suite policies. See Using the Extended Algorithm Suite. If digital signatures in the FIPS-140 mode are not required in your environment, then you can use the SHA-256 policies.

> **Note**
>
> SHA-1 Secure Hash Algorithm is not supported in FIPS mode. See Enabling FIPS Mode in *Administering Security for Oracle WebLogic Server*.

The predefined web service security policies select which specific algorithm they use in the `<sp:AlgorithmSuite>` element.

WebLogic Server includes policies such as `Wssp1.2-2007-Wss1.1-X509-Basic256Sha256.xml` that specifically use the SHA-256 secure hash algorithm, as shown in Table 2-2.

If an SHA-256 version of a policy you want to use exists, use it instead of the older SHA-1 version.

> **Note**
>
> For maximum security, Oracle recommends the use of SHA-256 instead of SHA-1, where possible.
>
> If you already use the older SHA-1 version of a policy, Oracle recommends that you update your web service to use the SHA-256 version.

Table 2-2 Using the SHA-256 Policies

Instead of this SHA-1 policy	...Use this SHA-256 policy
`Wssp1.2-2007-Https-UsernameToken-Plain.xml`	`Wssp1.2-2007-Https-UsernameToken-Plain-Basic256Sha256.xml`
`Wssp1.2-2007-Wss1.1-X509-Basic256.xml`	`Wssp1.2-2007-Wss1.1-X509-Basic256Sha256.xml`
`Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256.xml`	`Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256Sha256.xml`
`Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256.xml`	`Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256Sha256.xml`
`Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml`	`Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1-Basic256Sha256.xml`
`Wssp1.2-2007-Saml2.0-Bearer-Https.xml`	`Wssp1.2-2007-Saml2.0-Bearer-Https-Basic256Sha256.xml`

Update the Predefined SHA-1 Policies to SHA-256

The predefined policies listed in this section use SHA-1 for hashing digital signatures. This hashing algorithm might not meet your current or future security needs, as outlined in the NIST Special Publication 800-131A, "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths".

If you use any of these policies, Oracle recommends that you:

1. Use the predefined policy as a template to create a custom policy. See Creating and Using a Custom Policy File for information on creating a custom policy file.

   The policy files are located in *ORACLE_HOME*`oracle_common/modules/com.oracle.webservices.wls.wls-soap-stack-impl.jar`. Within `com.oracle.webservices.wls.wls-soap-stack-impl.jar`, the policy files are located in `/weblogic/wsee/policy/runtime`.

2. Edit the custom policy to change the algorithm suite to SHA-256. To do this, change the algorithm suite inside the policy.

   From:

   ```
   <sp:AlgorithmSuite>
   <wsp:Policy>
   <sp:Basic256/>
   </wsp:Policy>
   </sp:AlgorithmSuite>
   ```

   To:

   ```
   <sp:AlgorithmSuite>
   <wsp:Policy>
   <sp:Basic256Sha256/>
   </wsp:Policy>
   </sp:AlgorithmSuite>
   ```

3. Use the custom policy in your web service.

4. Edit the client-side policy to match. The client and web service must use the same hashing algorithm; `<AlgorithmSuite>` must be the same on both sides. Otherwise, the web service rejects the request message sent from the client.

SAML Policies

The following predefined policies use the SHA-1 algorithm. Change them as described in this section to instead use SHA-256.

- `Wssp1.2-2007-Saml2.0-Bearer-Wss1.1.xml`
- `Wssp1.2-2007-Saml2.0-HolderOfKey-Wss1.1-Asymmetric.xml`
- `Wssp1.2-2007-Saml2.0-HolderOfKey-Wss1.1-IssuedToken.xml`
- `Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml`
- `Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1-Asymmetric.xml`

Wss1.0 Policies

The following predefined policies use the SHA-1 algorithm. Change them as described in this section to instead use SHA-256.

- `Wssp1.2-2007-Wss1.0-UsernameToken-Digest-X509-Basic256.xml`
- `Wssp1.2-2007-Wss1.0-UsernameToken-Plain-X509-Basic256.xml`
- `Wssp1.2-2007-Wss1.0-X509-Basic256.xml`
- `Wssp1.2-Wss1.0-UsernameToken-Digest-X509-Basic256.xml`
- `Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic256.xml`
- `Wssp1.2-Wss1.0-X509-Basic256.xml`
- `Wssp1.2-Wss1.0-X509-EncryptRequest-SignResponse.xml`
- `Wssp1.2-Wss1.0-X509-SignRequest-EncryptResponse.xml`

Wss1.1 Policies

The following predefined policies use the SHA-1 algorithm. Change them as described in this section to instead use SHA-256.

- `Wssp1.2-2007-Wss1.1-DK-X509-SignedEndorsing.xml`
- `Wssp1.2-2007-Wss1.1-EncryptedKey-X509-SignedEndorsing.xml`
- `Wssp1.2-2007-Wss1.1-UsernameToken-Digest-DK.xml`
- `Wssp1.2-2007-Wss1.1-UsernameToken-Digest-EncryptedKey.xml`
- `Wssp1.2-2007-Wss1.1-UsernameToken-Digest-X509-Basic256.xml`
- `Wssp1.2-2007-Wss1.1-UsernameToken-Plain-DK.xml`
- `Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey.xml`
- `Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256.xml`
- `Wssp1.2-2007-Wss1.1-X509-Basic256.xml`
- `Wssp1.2-Wss1.1-DK.xml`
- `Wssp1.2-Wss1.1-DK-X509-Endorsing.xml`

- Wssp1.2-Wss1.1-EncryptedKey.xml
- Wssp1.2-Wss1.1-EncryptedKey-X509-SignedEndorsing.xml
- Wssp1.2-Wss1.1-UsernameToken-DK.xml
- Wssp1.2-Wss1.1-X509-Basic256.xml
- Wssp1.2-Wss1.1-X509-EncryptRequest-SignResponse.xml
- Wssp1.2-Wss1.1-X509-SignRequest-EncryptResponse.xml

Secure Conversation Policies

The following predefined policies use the SHA-1 algorithm. Change them as described in this section to instead use SHA-256.

- Wssp1.2-2007-Wssc1.3-Bootstrap-Https.xml
- Wssp1.2-2007-Wssc1.3-Bootstrap-Https-BasicAuth.xml
- Wssp1.2-2007-Wssc1.3-Bootstrap-Https-ClientCertReq.xml
- Wssp1.2-2007-Wssc1.3-Bootstrap-Https-UNT.xml
- Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.0.xml
- Wssp1.2-2007-Wssc1.3-Bootstrap-Wss1.1.xml
- Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256.xml
- Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.1-Saml2.0-Bearer.xml
- Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.1-UsernameToken-Plain-EncryptedKey.xml
- Wssp1.2-Wssc200502-Bootstrap-Wss1.1.xml

Using the Extended Algorithm Suite (EAS)

When using digital signatures, the WebLogic Server web service security policies include a set of policies that support an Extended Algorithm Suite (EAS) as required by the FIPS-140-2 certification. You can attach one of these EAS policies to your web service when FIPS 140-2 certification is required. Alternatively, if one of the policies do not satisfy the requirements of your environment, you can edit the algorithm suite in an existing policy and use that instead.

The standard algorithm suites supported in WebLogic Server web services policies, and the abbreviations used in the algorithm suite tables, are defined in the WS-SecurityPolicy 1.3 specification, which is available at http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/errata01/os/ws-securitypolicy-1.3-errata01-os-complete.html#_Toc325573605.

The extended algorithm suite policies, such as Wssp1.2-2007-Wss1.1-X509-Eas256.xml, use a stronger hash algorithm of SHA-256 and stronger signature method algorithm.

Extended Algorithm Suite Signature Values lists the symmetric signature (Sym Sig) and asymmetric signature (Asym Sig) values, and the associated algorithm URIs, for the extended algorithm suite policies.

Table 2-3 Extended Algorithm Suite Signature Values

Property Algorithm	Value/ Abbreviation	Algorithm URI
[Sym Sig]	HmacSha256	http://www.w3.org/2000/09/xmldsig#hmac-sha256
[Asym Sig]	RsaSha256	http://www.w3.org/2000/09/xmldsig#rsa-sha256

The XML signatures for RSA-SHA256 and HMAC-SHA256 are defined in the w3c XML Security Algorithm Cross-Reference specification, which is available at:

http://www.w3.org/TR/xmlsec-algorithms/.

Table 2-4 lists the algorithm suites for the extended algorithm suite policies.

Table 2-4 Algorithm Suites for Extended Algorithm Suite Policies

Algorithm Suite	Digest	Encryption	Symmetric Key Wrap	Asymmetric Key Wrap	Encrypted Key Derivation	Symmetric Signature	Asymmetric Signature	Signature Key Derivation	Minimum Signature Key Length
Basic256Exn256	Sha256	Aes256	KwAes256	KwRsaOaep	PSha1L256	HmacSha256	RsaSha256	PSha1L192	256
Basic192Exn256	Sha256	Aes192	KwAes192	KwRsaOaep	PSha1L192	HmacSha256	RsaSha256	PSha1L192	192
Basic128Exn256	Sha256	Aes128	KwAes128	KwRsaOaep	PSha1L128	HmacSha256	RsaSha256	PSha1L128	128
TripleDesExn256	Sha256	TripleDes	KwTripleDes	KwRsaOaep	PSha1L192	HmacSha256	RsaSha256	PSha1L192	192
Basic256Exn256Rsa15	Sha256	Aes256	KwAes256	KwRsa15	PSha1L256	HmacSha256	RsaSha256	PSha1L192	256
Basic192Exn256Rsa15	Sha256	Aes192	KwAes192	KwRsa15	PSha1L192	HmacSha256	RsaSha256	PSha1L192	192
Basic128Exn256Rsa15	Sha256	Aes128	KwAes128	KwRsa15	PSha1L128	HmacSha256	RsaSha256	PSha1L128	128
TripleDesExn256Rsa15	Sha256	TripleDes	KwTripleDes	KwRsa15	PSha1L192	HmacSha256	RsaSha256	PSha1L192	192

The predefined web service security policies select which specific algorithm they use in the `<sp:AlgorithmSuite>` element.

> **Note**
>
> The extended algorithm suite policies can also be used in non-FIPS mode for increased security. However, since they use their own namespace for the algorithm suite, there may be interoperability issues with other platforms, if the target platform does not support the extended algorithm suite assertion. Consider the following before using the extended algorithm suite policies:
>
> - If you have web services that require FIPS 140-2 certification, then use the EAS policies.
> - If you have new web services that do not need to interoperate with other platforms but you want increased security, you can use the EAS policies.
>
> For all other web services, you need to assess the security risk, interoperability, and backward compatibility before converting any policy to an EAS policy.

You can either use the EAS policies as is or identify an existing policy without EAS and modify the algorithm suite as shown:

1. Use an existing policy to create a custom policy, see Creating and Using a Custom Policy File.

   The policy files are located in `ORACLE_HOME/oracle_common/modules/com.oracle.webservices.wls.wls-soap-stack-impl.jar`. Within `com.oracle.webservices.wls.wls-soap-stack-impl.jar`, the policy files are located in `/weblogic/wsee/policy/runtime`.

2. Edit the custom policy to change the algorithm suite to FIPS-140-2. To do this, change the algorithm suite inside the policy.

   From

   ```
   <sp:AlgorithmSuite>
   <wsp:Policy>
   <sp:Basic256Sha256/>
   </wsp:Policy>
   </sp:AlgorithmSuite>
   ```

   To

   ```
   <sp:AlgorithmSuite>
   <wsp:Policy>
   <orasp:Basic256Exn256 xmlns:orasp="http://schemas.oracle.com/ws/2006/01/
   securitypolicy"/>
   </wsp:Policy>
   </sp:AlgorithmSuite>
   ```

3. Use the custom policy in your web service.
4. Edit the client-side policy to match. The client and web service must use the same hashing algorithm; *<AlgorithmSuite>* must be the same on both sides. Otherwise, the web service rejects the request message sent from the client.

Configuring Simple Message-Level Security

This section 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 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 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.p12, 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 *Developing JAX-WS Web Services for Oracle WebLogic Server*.

Configuring Simple Message-Level Security: Main Steps

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 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 *Developing JAX-WS 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 keytool utility to perform this step. For development purposes, the keytool utility is the easiest way to get started. See **Keytool** in JDK Tool Specifications .

   See Obtaining Private Keys, Digital Signatures, and Trusted Certificate Authorities in *Administering Security for 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 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 the Keytool utility to perform this step. See **Keytool** in JDK Tool Specifications .

   See Obtaining Private Keys, Digital Signatures, and Trusted Certificate Authorities in *Administering Security for Oracle WebLogic Server*.

5. Using the WebLogic Remote 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 Using a Client-Side Security Policy File.

7. Recompile your client application.

   See *Developing JAX-WS 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:

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

See Using System Properties to Debug Message-Level Security for information on debugging problems with your message-secured web service.

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.

See SSL Certificate Validation in *Administering Security for Oracle WebLogic Server*.

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 Loading a Policy From the CLASSPATH for an additional policy option.

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.

Setting the `uri` Attribute

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 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 Jakarta EE library; this method is useful if you want to share the file amongst multiple web services packaged in different Jakarta EE 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 Jakarta EE 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 Jakarta EE 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 Jakarta 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.

Setting Additional Attributes

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`.

Example of Using the `@Policy` and `@Policies` JWS Annotations

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 jakarta.jws.WebService;
import jakarta.jws.WebMethod;
import jakarta.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.

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'.

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

In the simple message-level configuration procedure, documented in Configuring Simple Message-Level Security, 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 the keytool utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started. See **Keytool** in JDK Tool Specifications .

See Obtaining Private Keys, Digital Signatures, and Trusted Certificate Authorities in *Administering Security for 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 the keytool utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started. See **Keytool** in JDK Tool Specifications .

   See Creating a Keystore and Creating a Keystore Using ImportPrivateKey in *Administering Security for Oracle WebLogic Server*.

3. Using the WebLogic Remote 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 *Administering Security for Oracle WebLogic Server*.

4. Using the WebLogic Remote 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.

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.

   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.

   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.

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 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 Using a Client-Side Security Policy File.

Example 2-2 shows a 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.

. 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

Example 2-2 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 jakarta.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);
    }
}
```

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:

  ```
  import jakarta.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:

```
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.

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

In the simple web services configuration procedure, described in Configuring Simple Message-Level Security, 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 WebLogic Remote 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 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 WebLogic Remote 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 *Administering Security for Oracle WebLogic Server*.
4. Using the WebLogic Remote 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 *Administering Security for 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 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.

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 2-3 shows the web service code.

> **Note**
>
> 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 2-4.

Example 2-3 Web Service SignEncrypt.java

```
package signencrypt;

import java.io.File;

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

import jakarta.activation.DataHandler;
import jakarta.activation.FileDataSource;
import jakarta.jws.WebMethod;
import jakarta.jws.WebService;
import jakarta.xml.ws.BindingType;
import jakarta.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;

 }

}
```

As noted, the web service implements *attachToWsdl=false*, and therefore the web service client needs to load a client-side version of the policy. Example 2-4 shows an example of using the *weblogic.jws.jaxws.ClientPolicyFeature* class to load client-side policies.

The example includes extensive inline comments.

Example 2-4 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 jakarta.activation.DataHandler;
import jakarta.activation.FileDataSource;
import javax.xml.namespace.QName;
import jakarta.xml.ws.BindingProvider;
import jakarta.xml.ws.WebServiceFeature;
import jakarta.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();
     }
   }
 }
}
```

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 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. See 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.

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.

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 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.

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

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/200702/
> IncludeToken/AlwaysToRecipient">
> <sp:Issuer>
> <a:Address>http://example.com/STS</a:Address>
> </sp:Issuer>
> . . .
> </sp:IssuedToken>
> ```

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-WS. 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);
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)
```

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-WS. The example assumes that the location of the STS URI is already known to the client.

```
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)
```

Configuring STS URI Using WLST: Client On Server Side

Example 2-5 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 2-5 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()
```

Configuring STS Security Policy: Standalone Client

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);
```

Configuring STS Security Policy Using WLST: Client On Server Side

Example 2-6 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 Using WS-SecurityPolicy 1.2 Policy Files) or a custom policy file in a Jakarta EE library (see Creating and Using a Custom Policy File).

Example 2-6 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()
```

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 (`https://jakarta.ee/specifications/soap-attachments/2.0/apidocs/jakarta.xml.soap/jakarta/xml/soap/soapconstants`) for the definitions of the constants.) The supported values for `WSEESecurityConstants.TRUST_SOAP_VERSION` are as follows:

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

Example 2-7 shows an example of setting the WS-Trust and SOAP versions.

Example 2-7 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);
```

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 2-8 shows an example of setting the STS server certificate under JAX-WS, assuming the location of the STS sever certificate is known.

Example 2-8 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);
```

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.

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/IncludeToken/
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". 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 2-9 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 2-9 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 jakarta.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());
  ```

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

Similar to Example 2-9, 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-
utility-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/IncludeToken/
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/IncludeToken/
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/IncludeToken/
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/IncludeToken/
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 2-10 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 2-10 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 secure 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);
  ```

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`

- WS-SecureConversation 1.4 policies:
  - `Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256.xml`
  - `Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256Sha256.xml`
  - `Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.1-Saml2.0-Bearer.xml`
  - `Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.1-UsernameToken-Plain-EncryptedKey.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 WS-SecureConversation Policies.

> **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. See Failover and Replication in a Cluster in *Administering 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.

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.

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.

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 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.

Associating Policy Files at Runtime

The simple message-level configuration procedure, documented in Configuring Simple Message-Level Security, 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 WebLogic Remote 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 WebLogic Remote Console, or you can specify some policy files using the annotations and then associate additional ones at runtime.

At runtime, the WebLogic Remote 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 WebLogic Remote 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.

Using Security Assertion Markup Language (SAML) Tokens For Identity

This section describes using SAML tokens for identity. The following topics are described:

SAML Token Overview

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 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 Configuring Simple Message-Level Security, 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 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 Configuring Simple Message-Level Security and now want to enable the additional use case of using SAML tokens, rather than username tokens, for identity.

Using SAML Tokens for Identity: Main Steps

To use SAML tokens for identity:

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 *Administering Security for Oracle WebLogic Server*:
   - *Configuring a SAML Identity Assertion Provider*
   - *Configuring a SAML Credential Mapping Provider*

   > **Note**
   >
   > 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)
```

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 2-13 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 Configuring Transport-Level Security in this case.

  See Table 2-13 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 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 2-13 for examples of predefined policies that use Bearer confirmation.

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 2.0 assertions use <saml2:SubjectConfirmation> elements 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`
- `bearer`

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-wssecurity-
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-wssecurity-
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 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 JAX-WS Web Services in *Developing JAX-WS 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 Invoking a Web Service From a Client Running in a WebLogic Server Instance for details.

Configuring SAML Attributes in a Web Service

A SAML assertion is a piece of data produced by a SAML authority regarding either an act of authentication performed on a subject, attribute information about the subject, or authorization data applying to the subject with respect to a specified resource.

The SAML specification (see `http://www.oasis-open.org`) allows additional, unspecified information about a particular subject to be exchanged between SAML partners as attribute statements in an assertion. A **SAML attribute assertion** is therefore a particular type of SAML assertion that conveys site-determined information about attributes of a Subject.

Attribute data is of type `String`.

Attributes are often name/value pairs (for example name=position, value=team lead), with multiple values being possible, but there is no requirement that they follow this model.

SAML attributes can be examined on the target partner service, and they can be used as extra information for authentication or authorization.

Use of SAML attributes works server-to-server. This means that the client application providing the attributes is running inside of a WebLogic Server instance. It then invokes a web service running in the same or other WebLogic Server instance to consume the attributes. Because the client application is itself a web service, the web services security runtime takes care of all the SAML processing.

Using SAML Attributes: Available Interfaces and Classes

You can use the classes and interfaces listed in Table 2-5 to implement SAML attributes. See Java API Reference for Oracle WebLogic Server.

Table 2-5 SAML Attribute Classes and Interfaces

Interface or Class	Description
`weblogic.wsee.security.saml.SAML2CredentialProvider`	Credential Provider for SAML 2.0 assertions.
`weblogic.wsee.security.saml.SAMLAttributeStatementData`	This interface represents the attributes in a single attribute statement.

Table 2-5 (Cont.) SAML Attribute Classes and Interfaces

Interface or Class	Description
`weblogic.wsee.security.saml.SAMLAttributeStatementDataImpl()`	This class represents the attributes in a single attribute statement.
`weblogic.wsee.security.saml.SAMLAttributeData`	SAML attribute Info interface for SAML 2.0 attributes.
`weblogic.wsee.security.saml.SAMLAttributeDataImpl()`	Class that implements `weblogic.wsee.security.saml.SAMLAttributeData`.
`weblogic.wsee.security.saml.SAMLAttributeStatementDataHelper`	Helper function to get the SAMLAttributeStatementData object

Of the classes and interfaces listed in Table 2-5, the `SAMLAttributeData` interface deserves additional mention. It has the methods shown in Table 2-6.

Table 2-6 SAMLAttributeData Methods

Method	Description
`getAttributeName()`	Get the attribute name.
`getAttributeNameFormat()`	Get the attribute name format.
`getAttributeFriendlyName()`	Get the Attribute friendly name.
`getAttributeValues()`	Get the collection of attribute values.
`isSAML20()`	Check if this is a SAML 2.0 attribute. Return true if it is a SAML 2.0 attribute, false otherwise
`setAttributeName(String attributeName)`	Set the attribute name.
`setAttributeNameFormat(String attributeNameFormat)`	Set the attribute name format.
`setAttributeFriendlyName(String attributeFriendlyName)`	Set the attribute friendly name.
`setAttributeValues(Collection<String> attributeValues)`	Set the collection of attribute values.
`addAttributeValue(String attributeValue)`	Add one attribute value.
`getAttributeNameSpace()`	Get the namespace of the attribute.
`setAttributeNameSpace(String attributeNameSpace)`	Set the namespace of the attribute.
`getSAML2AttributeInfo()`	Get a SAML 2.0 attribute info object from this object.
`isEmpty()`	Check if this attribute data element does not have values.

Using SAML Attributes: Main Steps

The `SAML2CredentialProvider` classes provide mechanisms to add attributes into SAML assertions via the web service context.

On the SAML partner, you then use the `SAMLAttributeStatementDataHelper.getSAMLAttributeStatementData` method to map attributes from incoming SAML assertions based on the web service context.

To do this:

- The SAML2CredentialProvider (on the SAML Identity Provider site) determines the attributes to use and how to package them.

  Implement both the `SAMLAttributeStatementData` and `SAMLAttributeData` interfaces to package the attributes.
- The SAML partner uses the `WebServiceContext` to get the attributes, and determines what to do with them.

  Use the SAMLAttributeStatementDataHelper class to get the `SAMLAttributeStatementData` object, from which you get the `SAMLAttributeData` object.

SAML Attributes Example

This section describes a simple application that implements SAML attributes for SAML 2.0. This example is available in the WebLogic Server installation in *`WLS_HOME\samples\server\examples\src\examples\webservices\saml\saml20sv`*.

Example 2-11 shows an example of a web service (the "client") running on a WebLogic Server instance.

This web service adds four attributes to the `WebServiceContext`. The first attribute has no value; the second uses a static value. The values for attributes three and four are computed based on the authenticated Subject.

Example 2-11 Web Service That Adds Attributes to the WebServiceContext

```
@WebService(serviceName = "ProxyService", name = "IProxy", targetNamespace = "http://
www.oracle.com/2008/12/interop")
 public class ProxyService{

   @WebMethod(operationName = "Echo")
   @WebResult(name = "EchoResponse")
   public String echo(@WebParam(name = "EchoRequest")String hello,
                      @WebParam(name = "partenerWsdlURL") String partenerWsdlURL){
      try{
        PartnerService service =
                new PartnerService(new URL(partenerWsdlURL),
                                           new QName("http://www.oracle.com/2008/12/
interop", "PartnerService"));

        IPartner port = service.getIPartnerPort();
        BindingProvider provider = (BindingProvider) port;
        Map context = provider.getRequestContext();
        context.put(WLStub.SAML_ATTRIBUTE_ONLY, "True");

        List credProviders = buildCredentialProviderList();
        context.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);

        String result = port.echo(hello);
        return result+" I'm ProxyService Echo!\n";

      } catch(Exception ex ){
        throw new RuntimeException(ex);
      }
   }
```

```
    private static List buildCredentialProviderList() throws Exception  {
     List credProviders = new ArrayList();
     credProviders.add(new MySAMLCredentialProvider1());

     return credProviders;
    }

    /**
     * This Credential Provider is for SAML 2.0 Sender Vouches
     */

    private static class MySAMLCredentialProvider1 extends SAML2CredentialProvider {

        public SAMLAttributeStatementData getSAMLAttributeData(Subject subject) {

            System.out.println(" Providing SAML Attributes from
MySAMLCredentialProvider1 for Subject =" + subject);
            // There are four types of attributes in this test

            SAMLAttributeStatementData attributes = new SAMLAttributeStatementDataImpl();

            String xmlns = "www.oracle.com/webservices/saml/test";
            // 1. The attribute without value

            SAMLAttributeData attribute1 = new SAMLAttributeDataImpl();
            attribute1.setAttributeName("test.no.value.attribute");
            // Friendly name is optional. It is set in this example.
            attribute1.setAttributeFriendlyName("Type 1 - No Value");
            attribute1.setAttributeNameSpace(xmlns);
            attributes.addAttributeInfo(attribute1);

            // 2. Static attribute that has static value

            SAMLAttributeData attribute2 = new SAMLAttributeDataImpl();
            attribute2.setAttributeName("test.static.attribute");
            attribute2.setAttributeFriendlyName("Type 2 - Static Attribute");
            attribute2.setAttributeNameSpace(xmlns);
            attribute2.addAttributeValue("static.attribute.value");
            attributes.addAttributeInfo(attribute2);

            // 3. Subjust dependent attributes

            SAMLAttributeData attribute3 = new SAMLAttributeDataImpl();
            attribute3.setAttributeName("test.subject.dependent.attribute");
            attribute3.setAttributeFriendlyName("Type 3 - Subject Dependent Attribute");
            attribute3.setAttributeNameSpace(xmlns);
            if (hasUser("Alice", subject)) {
                 attribute3.addAttributeValue("Alice A");
            }  else if (hasUser("Bob", subject)) {
                 attribute3.addAttributeValue("Bob B");
            }   else {
                attribute3.addAttributeValue("Hacker X");
            }
             attributes.addAttributeInfo(attribute3);

             // 4. Multiple value attributes

            SAMLAttributeData attribute4 = new SAMLAttributeDataImpl();
            attribute4.setAttributeName("test.multi.value.attribute");
            attribute4.setAttributeFriendlyName("Type 4 - Multi-Value Attribute");
            attribute4.setAttributeNameSpace(xmlns);
```

```
            if (hasUser("Alice", subject)) {
                 attribute4.addAttributeValue("Team Lead");
                 attribute4.addAttributeValue("Programmer");
            }  else if (hasUser("Bob", subject)) {
                 attribute4.addAttributeValue("System Admin");
                 attribute4.addAttributeValue("QA");
            }   else {
                attribute4.addAttributeValue("Hacker");
                attribute4.addAttributeValue("meber of unkown");
            }
            attributes.addAttributeInfo(attribute4);
            return attributes;
         }

        private static boolean hasUser(String user, Subject subject) {
           if (null == user || null == subject) {
               return false;
           }
           Set principals = subject.getPrincipals();
           if (null == principals || principals.isEmpty()) {
               return false;
           }
           for (Iterator it = principals.iterator(); it.hasNext(); )  {
               Object obj = it.next();
               if (obj instanceof Principal) {
                   Principal  p = (Principal) obj;
             //        System.out.println("principal =[" + p + "]");
                   if (user.equals(p.getName())) {
                       return true;
                   }
               }   else if (obj instanceof WLSPrincipal)  {
                   WLSPrincipal principal = (WLSPrincipal) obj;
             //        System.out.println("principal =[" + principal + "]");
                   if (user.equals(principal.getName())) {
                       return true;
                   }
               }
            }
            return false;
        }

    }

}
```

This example invokes the `SAMLAttributeStatementDataImpl()` class to get an `SAMLAttributeStatementData` object, and then invokes `SAMLAttributeDataImpl()` to get a `SAML2AttributeStatementInfo` object.

In this example, the `SAMLAttributeData` class uses SAML 2.0. `SAMLAttributeDataImpl()` is shown in Example 2-12.

Example 2-12 SAMLAttributeDataImpl Implementation

```
package weblogic.wsee.security.saml;

import com.bea.security.saml2.providers.SAML2AttributeInfo;

import java.util.Collection;
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;
```

```
/**
 *
 */
public class SAMLAttributeDataImpl implements SAMLAttributeData {
    public static final String SAML_2_0_ATTRNAME_FORMAT_BASIC =
SAML2AttributeInfo.ATTR_NAME_FORMAT_BASIC;

    /**
     * the name of the attribute
     */
    private String attributeName;

    private String attributeNameSpace;
    /**
     * the name format of the attribute for SAML 2.0. Defaults to basic.
     */
    private String attributeNameFormat = SAML_2_0_ATTRNAME_FORMAT_BASIC;
    /**
     * the friendly name of the attribute, this is for SAML 2.0 only.
     */
    private String attributeFriendlyName;
    /**
     * the values of the attribute.
     */
    private Collection<String> attributeValues;
    /**
     * is a SAML 2.0 attribute info
     */
    private boolean isSAML20;

    public SAMLAttributeDataImpl() {

    }

    public SAMLAttributeDataImpl(String attributeName, Collection<String>
attributeValues) {
        this.attributeName = attributeName;
        this.attributeValues = attributeValues;
    }

    public SAMLAttributeDataImpl(String attributeName, String
 attributeNameFormat, String attributeFriendlyName, String namespace,
 Collection<String> attributeValues) {
        this.attributeName = attributeName;
        this.attributeNameFormat = attributeNameFormat;
        this.attributeFriendlyName = attributeFriendlyName;
        this.attributeValues = attributeValues;
        this.attributeNameSpace = namespace;
    }

    public SAMLAttributeDataImpl(SAML2AttributeInfo saml2AttributeInfo)  {
        if (null == saml2AttributeInfo) {
            throw new IllegalArgumentException("Null SAML2AttributeInfo found ");
        }
        this.attributeName = saml2AttributeInfo.getAttributeName();
        this.attributeNameFormat = saml2AttributeInfo.getAttributeNameFormat();
        this.attributeFriendlyName = saml2AttributeInfo.getAttributeFriendlyName();
        this.attributeValues = saml2AttributeInfo.getAttributeValues();
        this.isSAML20 = true;
    }
```

```
    /**
     * get the attribute name
     *
     * @return string of the attribute name
     */
    public String getAttributeName() {
        return attributeName;
    }

    /**
     * set the attribute name
     *
     * @param attributeName string of the attribute name
     */
    public void setAttributeName(String attributeName) {
        if (null == attributeName) {
            throw new IllegalArgumentException("attributeName cannot be null");
        }
        this.attributeName = attributeName;
    }

    /**
     * get the attribute name format for SAML 2.0 only
     *
     * @return String of the attribute name format,
default is SAML_2_0_ATTRNAME_FORMAT_BASIC for SAML 2.0.
     */
    public String getAttributeNameFormat() {
        return attributeNameFormat;
    }
    /**
     * set et the attribute name format
     *
     * @param attributeNameFormat String of the attribute name format
     */
    public void setAttributeNameFormat(String attributeNameFormat) {
        this.attributeNameFormat = attributeNameFormat;
    }
    /**
     * get the Attribute Friendly Name
     *
     * @return String of the Attribute Friendly Name
     */
    public String getAttributeFriendlyName() {
        return attributeFriendlyName;
    }
    /**
     * set the Attribute Friendly Name
     *
     * @param attributeFriendlyName the Attribute Friendly Name
     */
    public void setAttributeFriendlyName(String attributeFriendlyName) {
        this.attributeFriendlyName = attributeFriendlyName;
    }
    /**
     * get the Attribute Value
     *
     * @return collection of attribute values
     */
    public Collection<String> getAttributeValues() {
        return attributeValues;
    }
```

```
/**
 * set collection of attribute values
 *
 * @param attributeValues collection of attribute values to be set
 */
public void setAttributeValues(Collection<String> attributeValues) {
    this.attributeValues = attributeValues;
}
/**
 * add one attribute value
 *
 * @param attributeValue String of attribute value to be added
 */
public void addAttributeValue(String attributeValue) {
    if (this.attributeValues == null) {
        this.attributeValues = new ArrayList();
    }
    if (null == attributeValue) {
        this.attributeValues.add("");
    } else {
        this.attributeValues.add(attributeValue);
    }
}
/**
 * add  attribute values
 *
 * @param newAttributeValues collection of attribute values to be added
 */
public void addAttributeValues(Collection<String> newAttributeValues) {
    if (this.attributeValues == null || this.attributeValues.isEmpty()) {
        this.setAttributeValues(newAttributeValues);
        return;
    }
    if (null == newAttributeValues || newAttributeValues.isEmpty()) {
                    this.attributeValues.add("");
        return;
    }
    Iterator iter = newAttributeValues.iterator();
    while (iter.hasNext()) {
        this.attributeValues.add((String) iter.next());
    }
}
/**
 * get the namespace of the Attribute.
 *
 * @return string of attribute namespace
 */
public String getAttributeNameSpace() {
    return attributeNameSpace;
}
/**
 * set attributeNameSpace.
 *
 * @param attributeNameSpace attributeNameSpace to be set
 */
public void setAttributeNameSpace(String attributeNameSpace) {
    this.attributeNameSpace = attributeNameSpace;
}
/**
 * set this data object to SAML 2.0 attribute object
 * @param saml20 true if it is a SAML 2.0 attribute data
 */
```

```
    public void setSAML20(boolean saml20) {
        this.isSAML20 = saml20;
    }
    /**
     * check if this is a SAML 2.0 Attributes
     *
     * @return true if it is a SAML 2.0 attribute, false otherwise
     */
    public boolean isSAML20() {
        return isSAML20;
    }
    /**
     * get a SAML2AttributeInfo object from this object
     *
     * @return SAML2AttributeInfo for SAML 2.0
     */
    public SAML2AttributeInfo getSAML2AttributeInfo() {
        SAML2AttributeInfo sai = new SAML2AttributeInfo();
        sai.setAttributeFriendlyName(this.attributeFriendlyName);
        sai.setAttributeName(this.attributeName);
        if (null == this.attributeNameFormat || this.attributeNameFormat.length() ==0 ) {
            sai.setAttributeNameFormat(SAML_2_0_ATTRNAME_FORMAT_BASIC);
        } else {
            sai.setAttributeNameFormat(this.attributeNameFormat);
        }
        sai.addAttributeValues(this.attributeValues);
        return sai;
    }
    /**
     * This method will add all attribute values into the first SAMLAttributeData
object, and return a single SAMLAttributeData object.
     * Please note that the attribute name will not be verified in this method.
     *
     * @param attributeList  SAMLAttributeData objects to be merged
     * @return  a single SAMLAttributeData object
     */
    static public SAMLAttributeData consolation(List<SAMLAttributeData>
attributeList) {
        if (null == attributeList || attributeList.size() == 0 ) {
            return null;
        }
        if (attributeList.size() == 1) {
            attributeList.get(0);
        }
        SAMLAttributeData data = attributeList.get(0);
        for (int i=1; i < attributeList.size(); i++ ) {
            data.addAttributeValues(attributeList.get(i).getAttributeValues());
        }
        return data;
    }
    /**
     * Check if this attribute data element does not have vlaues
     * @return true if the data is empty, no values; false otherwise
     */
    public boolean isEmpty() {
        if ((null == this.attributeValues) || (this.attributeValues.isEmpty())) {
            return true;
        }
        if (this.attributeValues.size() == 1)  {
            Object a[] = this.attributeValues.toArray();
            if ("".equals(a[0]))  {
                return true;
```

```
            }
        }
        return false;
    }
    /**
     * Return a String for the array of value String, concatenated with "; "
     * @return a string for all values
     */
    public String valuesToString(String existing) {
        if ((null == this.attributeValues) || (this.attributeValues.isEmpty())) {
            return existing;
        }
        Object a[] = this.attributeValues.toArray();
        if (this.attributeValues.size() == 1) {
           if (a[0] == null)  {
               return existing;
           }
           if (existing == null)  {
             return (String) a[0];
           } else {
               return existing + "; " + (String) a[0];
           }
        }
        StringBuffer sb = new StringBuffer();
        if (existing != null) {
           sb.append(existing);
        }
        for (int i=0; i < a.length; i++) {
           sb.append("; ");
           if (a[i] != null) {
             sb.append((String) a[i]);
           }
        }
        return sb.toString();
    }
    public String toString() {
       StringBuffer sb = new StringBuffer();
       sb.append("Name=" + this.attributeName);
       if (isSAML20()) {
          if (null != this.attributeFriendlyName) {
             sb.append(" FriendlyName=" + this.attributeFriendlyName);
          }
       } else {
          if (null != this.attributeNameSpace) {
             sb.append(" Namespace=" + this.attributeNameSpace);
          }
       }
       String value = this.valuesToString(null);
       if (null != value)   {
           sb.append(" Value=" + value);
       }
       return sb.toString();
    }
}
```

Example 2-13 shows the PartnerService code that determines if the web service context has attributes, and then gets them. This example relies on the `SAMLAttributeStatementDataHelper` class.

The predefined policy used in this example, Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml, is described in Table 2-13.

Example 2-13 Web Service That Gets Attributes From the WebServiceContext

```
package jaxws.interop.saml;

import weblogic.jws.Policies;
import weblogic.jws.Policy;
import weblogic.wsee.util.AccessException;
import weblogic.wsee.security.saml.SAMLAttributeStatementData;
import weblogic.wsee.security.saml.SAMLAttributeStatementDataHelper;
import weblogic.wsee.security.saml.SAMLAttributeData;

import jakarta.jws.WebMethod;
import jakarta.jws.WebParam;
import jakarta.jws.WebResult;
import jakarta.jws.WebService;
import jakarta.annotation.Resource;
import jakarta.xml.ws.WebServiceContext;

/**
 *  ID Propagation using SAML 2.0 token [sender-vouches] with message protection (WSS
11)  .
 *
 * This example will work for canned policy like:
 *    - Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml
 */


@Policies(
    {
      @Policy(uri = "policy:Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml"),
      @Policy(uri = "policy:Wssp1.2-2007-SignBody.xml"),
      @Policy(uri = "policy:Wssp1.2-2007-EncryptBody.xml")
        }
)
@WebService(serviceName = "PartnerService", name = "IPartner", targetNamespace = "http://
www.oracle.com/2008/12/interop")
public class PartnerService{
  @Resource
  WebServiceContext ctx;


  @WebMethod(operationName = "Echo")
  @WebResult(name = "EchoResponse")
  public String echo(@WebParam(name = "EchoRequest")String hello){
    try {
       this.checkSamlAttributesFromRequestMesasge();
       return hello+"! I'm PartnerService for SAML 2.0 SenderVouches WSS1.1!\n";
    }catch(Exception ex ){
        throw new RuntimeException(ex);
    }
  }

    private void checkSamlAttributesFromRequestMesasge() throws AccessException {

       SAMLAttributeStatementData attributes  =
SAMLAttributeStatementDataHelper.getSAMLAttributeStatementData(ctx);
       if (null == attributes)  {
           throw new AccessException("No SAML Attributes Data found");
       }

       SAMLAttributeData testData =
attributes.getAttributeInfo("test.no.value.attribute");
```

```
       if (null == testData) {
          throw new AccessException("Missing SAML Attribute Data of
\"test.no.value.attribute\"");
       }
       if (!attributes.hasAttributeInfo("test.no.value.attribute")) {
          throw new AccessException("Missing SAML Attribute Data of
\"test.no.value.attribute\"");
       }
       if (!attributes.hasAttributeInfo("test.static.attribute")) {
          throw new AccessException("Missing SAML Attribute Data of
\"test.static.attribute\"");
       }
       if (!
attributes.hasAttributeValue("test.static.attribute","static.attribute.value")) {
          throw new AccessException("Missing or wrong SAML Attribute Value of
\"static.attribute.value\" for attribute \"test.static.attribute\" ");
       }
       if (!attributes.hasAttributeValue("test.subject.dependent.attribute","Alice A")) {
          throw new AccessException("Missing or wrong SAML Attribute Value of \"Alice
A\" for attribute - \"test.multi.value.attribute\" ");
       }
       if (!attributes.hasAttributeValue("test.multi.value.attribute","Programmer")) {
          throw new AccessException("Missing or wrong SAML Attribute Value on
\"Programmer\" for attribute \"test.multi.value.attribute\" ");
       }
       if (!attributes.hasAttributeValue("test.multi.value.attribute","Team Lead")) {
          throw new AccessException("Missing or wrong SAML Attribute Value on \"Team
Lead\"  for attribute \"test.multi.value.attribute\" ");
       }
    }
}
```

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

Many use cases previously discussed require you to use WebLogic Remote 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 WebLogic Remote Console 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*.

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

   See Invoking Web Services in *Developing JAX-WS 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.

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`.

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 2-7 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 2-7 (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.

Using a Client-Side Security Policy File

The section 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 2-4 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.

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 *Developing JAX-WS 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 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.

Running with High Contrast and Text Magnification

Running the WebLogic Server Administration Console while using high contrast or text magnification can lead to the following problems in some browsers:

1. When using the Microsoft Windows High Contrast mode, some images and navigation controls are not displayed or are distorted.
2. When running with text magnification some text may be overlapped or difficult to read.

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 *ORACLE_HOME*`oracle_common/modules/com.oracle.webservices.wls.wls-soap-stack-impl.jar`. Within `com.oracle.webservices.wls.wls-soap-stack-impl.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:

- Transport-Level Policies
- Protection Assertion Policies
- WS-Security 1.0 Username and X509 Token Policies
- WS-Security 1.1 Username and X509 Token Policies
- WS-SecureConversation Policies
- SAML Token Profile Policies

In addition, see Choosing a Policy and 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.

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 2-8 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.
`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.
`Wssp1.2-2007-Https-UsernameToken-Plain-Basic256Sha256.xml`	Same as `Wssp1.2-2007-Https-UsernameToken-Plain.xml` but uses a stronger hash algorithm of Sha-256.
`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.

Table 2-8 (Cont.) Transport Level Policies

Policy File	Description
`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.

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 2-9 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.
`Wssp1.2-2007-Wsp1.5-SignAndEncryptWSATHeaders.xml`	WS-AtomicTransaction headers are signed and encrypted. Web Services Policy 1.5 is used.

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 2-10 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
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.

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 2-11 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-X509-Basic256Sha256.xml	Same as Wssp1.2-2007-Wss1.1-X509-Basic256.xml but uses a stronger hash algorithm of Sha-256.
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-UsernameToken-Plain-X509-Basic256Sha256.xml	Same as Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256.xml but uses a stronger hash algorithm of Sha-256.
Wssp1.2-2007-Wss1.1-X509-Eas256.xml	This policy is similar to policy Wssp1.2-2007-Wss1.1-X509-Basic256.xml but uses an extended algorithm suite, which has a stronger hash algorithm of Sha-256 and stronger signature method algorithm. This policy is supported in FIPS-140 mode.
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 2-11 (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.

WS-SecureConversation Policies

The policies in Table 2-12 implement WS-SecureConversation 1.3, 1.4, and WS-SecureConversation 2005/2.

> **Note**
>
> As described in *Developing JAX-WS Web Services for Oracle WebLogic Server*, if you are using a template to configure your domain, the Advanced JAX-WS template (wls_webservice_jaxws) is required for any JAX-WS web service that uses WS-SecureConversation.

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

Table 2-12 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-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256Sha256.xml`	This policy is similar to policy `Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256.xml`, but uses a stronger hash algorithm of Sha-256.
`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.

Table 2-12 (Cont.) WS-SecureConversation Policies

Policy File	Description
`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.
`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.
`Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256.xml`	WS-SecureConversation handshake is protected by WS-Security 1.0 X509 with asymmetric binding and authentication with plain-text Username Token, similar to the `Wssp1.2-2007-Wss1.0-UsernameToken-Plain-X509-Basic256.xml` policy. The SOAP body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages for the handshake are both signed and encrypted. The application messages are signed and encrypted with derived keys from a secure conversation token encrypted key. The WS-Addressing headers are signed. The policy use WS-Policy 1.5 namespace "`http://www.w3.org/ns/ws-policy`".
`Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Eas256.xml`	This policy is similar to policy `Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.0-UsernameToken-Plain-X509-Basic256.xml` but uses an extended algorithm suite, which has a stronger hash algorithm of Sha-256 and stronger signature method algorithm. This policy is supported in FIPS-140 mode.
`Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.1-Saml2.0-Bearer.xml`	WS-SecureConversation handshake is protected by WS-Security 1.1 X509 with asymmetric binding and authentication with SAML 2.0 Bearer Token. The SOAP body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages for the handshake are both signed and encrypted. The application messages are signed and encrypted with derived keys from a secure conversation token encrypted key. The WS-Addressing headers are signed. The policy use WS-Policy 1.5 namespace "`http://www.w3.org/ns/ws-policy`".
`Wssp1.2-2007-Wssc1.4-Bootstrap-Wss1.1-UsernameToken-Plain-EncryptedKey.xml`	WS-SecureConversation handshake is protected by WS-Security 1.1 X509 with asymmetric binding and authentication with plain-text Username Token, which is similar to the `Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey.xml` policy. The SOAP body of the RequestSecurityToken and RequestSecurityTokenResponseCollection messages for the handshake are both signed and encrypted. The application messages are signed and encrypted with derived keys from a secure conversation token encrypted key. The WS-Addressing headers are signed. The policy use WS-Policy 1.5 namespace "`http://www.w3.org/ns/ws-policy`".

Table 2-12 (Cont.) WS-SecureConversation Policies

Policy File	Description
`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.

SAML Token Profile Policies

The policies shown in Table 2-13 implement WS-Security SAML Token Profile 1.0 and 1.1.

Table 2-13 WS-Security SAML Token Profile Policies

Policy File	Description
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.

Table 2-13 (Cont.) WS-Security SAML Token Profile Policies

Policy File	Description
Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1-Basic256Sha256.xml	This policy is similar to policy Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml but uses a stronger hash algorithm of Sha-256.
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-Saml2.0-SenderVouches-Wss1.1-Eas256.xml	This policy is similar to policy Wssp1.2-2007-Saml2.0-SenderVouches-Wss1.1.xml but uses an extended algorithm suite, which has a stronger hash algorithm of Sha-256 and stronger signature method algorithm. This policy is supported in FIPS-140 mode.
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.
Wssp1.2-2007-Saml2.0-Bearer-Https.xml	One-way SSL uses SAML 2.0 token with Bearer confirmation method for Authentication. WebLogic Server supports the SAML 2.0 Bearer confirmation method at the transport level, using Wssp1.2-2007-Saml2.0-Bearer-Https.xml. 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.
Wssp1.2-2007-Saml2.0-Bearer-Https-Basic256Sha256.xml	Same as Wssp1.2-2007-Saml2.0-Bearer-Https.xml but uses a stronger hash algorithm of Sha-256.

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`

Unsupported WS-SecurityPolicy 1.2 Assertions

The WS-SecurityPolicy 1.2 assertions in Version-Independant Policy Supported are not supported in this release of WebLogic Server.

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

Table 2-14 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.
5.4.4	KerberosToken	Not supported in this release.
5.4.5	SpnegoContextToken	Not supported in this release.
5.4.9	RelToken	Not supported in this release.

Table 2-14 (Cont.) Web Services SecurityPolicy 1.2 Unsupported Assertions

Specification	Assertion	Remarks
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.
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: ../sp:SignedParts assertion, ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion
8.2 8.3 8.4 8.5	SignedSupportingTokens EndorsingSupportingTokens SignedEndorsingSupportingTokens SignedEncryptedSupportingTokens	Not supported in this release: ../sp:SignedParts assertion ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion ../sp:SignedEncryptedSupportingTokens assertion 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. 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.

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

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.

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 2-14.

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

Example 2-14 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 2-14 to create a custom security policy file, perhaps called `EncryptCreditCard.xml`.

Consider the example shown in Example 2-15.

Example 2-15 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:creditCard
    </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:creditCard
  ```
- 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.

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 2-16.

Example 2-16 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.

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 Jakarta EE library; this method is useful if you want to share the file amongst multiple web services packaged in different Jakarta EE 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 Jakarta EE 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 Jakarta EE 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 Jakarta 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.

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.

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.
- 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.

Example of Security Policy With Policy Alternatives

Example 2-17 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 2-17 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/IncludeToken/
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/
IncludeToken/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/IncludeToken/
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/
IncludeToken/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>
```

Configuring Smart Policy Selection

You can configure multiple policy alternatives for a single web service by creating a custom policy, as shown in Example 2-17. You then configure the web service client to make a policy selection preference.

You can configure the policy selection preferences for the web service client by using the WebLogic Remote Console, and using stubs.

The following preferences are supported:

- Security
- Performance
- Compatibility

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.

Configuring Smart Policy Selection in the Console

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

1. If you do not already have a functional web services security configuration, create a web services security configuration.
2. In the **Edit Tree**, go to **Security**, then **Web Service Securities** and select the web services security configuration that you want to configure.
3. From the **Policy Selection Preference** drop-down list, select an option:
   - 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)
4. Save and commit your changes.

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 2-18, 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 2-18 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 2-19, 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 2-19 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>
```

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 for JAX-WS:

```
BindingProvider bindingProvider = (BindingProvider) port;
Map<String,Object> rc =
(Map<String,Object>)bindingProvider.getRequestContext();
rc.put(WLStub.POLICY_SELECTION_PREFERENCE,
WLStub.PREFERENCE_COMPATIBILITY_PERFORMANCE_SECURITY);
```

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

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.

Example of Adding Security to Reliable Messaging Web Service

This section describes an update to an example that is optionally included with WebLogic Server:

- *EXAMPLES_HOME*\wl_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.

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.

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.

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.

Files Used by This Example

The example uses the files shown in the **Version-Independent Matrix** table in Version-Independent Policy Supported. The contents of revised source files are shown in subsequent sections.

Table 2-15 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.

Table 2-15 (Cont.) Files Used in WSRM/Security Example

File	Description
`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.

Revised ReliableEchoServiceImpl.java

The `ReliableEchoServiceImpl.java` JWS file is the same as that in *`EXAMPLES_HOME`*`\wl_server\examples\src\examples\webservices\wsrm_security\ReliableEchoServiceImpl.java`, with the revised Policy annotation shown in bold.

Example 2-20 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.

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 *EXAMPLES_HOME*`\wl_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 2-21 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
:
```

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 *EXAMPLES_HOME*`\wl_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 2-22 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
:
```

Building and Running the Example

After you have changed the example to use the new policy namespace, follow the steps in the *EXAMPLES_HOME*`\wl_server\examples\src\examples\webservices\wsrm_security\instructions.html` file to build and run the example.

There are no changes needed to these steps.

Securing Web Services Atomic Transactions

When using web services atomic transactions, as described in Using Web Services Atomic Transactions in *Developing 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 Example of Adding Security to a JAX-WS Web Service.
- At deployment time, use the WebLogic Remote Console to associate policy files at runtime.

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 jakarta.jws.WebService;
import jakarta.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")
@jakarta.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;
}
```

3

Configuring Transport-Level Security

The chapter describes how to configure transport-level security for your WebLogic web service using Jakarta XML Web Services (JAX-WS).
Transport-level security refers to securing the connection between a client application and a web service 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.

See Secure Sockets Layer (SSL) in *Understanding Security for Oracle WebLogic Server* for general information about SSL and the implementations included in WebLogic Server.

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

This chapter includes the following sections:

Configuring Transport-Level Security Through Policy

WebLogic Server includes the predefined transport-level policy files described in Available Transport-Level Policies, which typically satisfy the security needs of most programmers and use cases.

You can also create and use your own WS-SecurityPolicy file if you need additional configuration, as described in Creating and Using a Custom Policy File. If you need to do this, you can use the predefined WS-SecurityPolicy files as templates to create your own custom files. The policy .xml files are located in *`WL_HOME`*`/server/lib/weblogic.jar`. Within `weblogic.jar`, the policy files are located in `/weblogic/wsee/policy/runtime`.

For example, the Oracle-supplied `Wssp1.2-2007-Saml2.0-Bearer-Https.xml` policy file includes the following assertion indicating that the policy requires one-way SSL, as shown here.

Example 3-1 Specifying SSL in a Policy

```
<sp:TransportToken>
<wsp:Policy>
<sp:HttpsToken/>
</wsp:Policy>
</sp:TransportToken>
```

If you needed to instead use two-way SSL, you could create a custom policy that adds the `RequireClientCertificate` assertion, as shown below.

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 SSL and HTTP BASIC Authentication, as shown below.

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>
```

Available Transport-Level Policies

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

Table 3-1 Transport Level 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.
`Wssp1.2-2007-Saml2.0-Bearer-Https-Basic256Sha256.xml`	Same as `Wssp1.2-2007-Saml2.0-Bearer-Https.xml` but uses a stronger hash algorithm of Sha-256.
`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.
`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. Set **Two Way Client Cert Behavior** to **Client Certs Requested But Not Enforced**. See Set Up TLS in *Oracle WebLogic Remote Console Online Help* .
`Wssp1.2-2007-Https-UsernameToken-Digest.xml`	One way SSL with digest Username Token.

Table 3-1 (Cont.) Transport Level Policies

Policy File	Description
`Wssp1.2-2007-Https-UsernameToken-Plain.xml`	One way SSL with plain text Username Token.
`Wssp1.2-2007-Https-UsernameToken-Plain-Basic256Sha256.xml`	Same as `Wssp1.2-2007-Https-UsernameToken-Plain.xml` but uses a stronger hash algorithm of Sha-256.
`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.

Prerequisite: Configure SSL

Before you can use a transport-level policy to protect a web service, you must configure SSL for the core WebLogic Server security subsystem.

The out-of-the-box private key and X.509 certificate pairs are provided for demonstration and testing purposes. For this reason Oracle highly recommends you use your own keystore and key pair in production.

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 *Administering Security for Oracle WebLogic Server*.

If you configure two-way SSL for WebLogic Server, you must also configure SSL for the client application, as described in Configuring Two-Way SSL for a Client Application.

Configuring SSL: Main Steps

This section summarizes the procedure described in Setting Up SSL: Main Steps. The steps are described here for your convenience; see Setting Up SSL: Main Steps for complete information.

To set up SSL:

1. Configure identity and trust, as described in Configuring Keystores:
   a. Obtain digital certificates, private keys, and trusted CA certificates from the CertGen utility, the keytool utility, or a reputable vendor such as Entrust or Verisign. You can also use the digital certificates, private keys, and trusted CA certificates provided by

the WebLogic Server kit. The demonstration digital certificates, private keys, and trusted CA certificates should be used in a development environment only.

b. Store the private keys, digital certificates, and trusted CA certificates. Private keys and trusted CA certificates are stored in a keystore.

c. Configure the identity and trust keystores for WebLogic Server in WebLogic Remote Console. See Configure Keystores in *Oracle WebLogic Remote Console Online Help*.

2. Set SSL configuration options for the private key alias and password in WebLogic Remote Console.

   Optionally, set configuration options that require the presentation of client certificates (for two-way SSL). See Set Up TLS in *Oracle WebLogic Remote Console Online Help*.

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 Java SE 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 *Administering Security for 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 Java SE 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 the keytool utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started. See **Keytool** in JDK Tool Specifications .

  See Obtaining Private Keys, Digital Certificates, and Trusted Certificate Authorities in *Administering Security for Oracle WebLogic Server*.

- Set **Two Way Client Cert Behavior** to "Client Certs Requested But Not Enforced." See Set Up TLS in *Oracle WebLogic Remote Console Online Help*.

- 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

Configuring Transport-Level Security Through Policy: Main Steps

To configure transport-level web services security via one or more policy files:

1. As outlined in Prerequisite: Configure SSL, 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 *Administering Security for Oracle WebLogic Server*.

2. Use `@Policy` or `@Policies` JWS annotations in your JWS file, or associate policy files only at runtime using WebLogic Remote Console, or specify some policy files using the annotations and then associate additional ones at runtime.

   See Table 3-1 for a description of the available transport-level policies.

   > **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.

   The following example attaches the policy at the class level:

   ```
   @Policy(uri="policy:Wssp1.2-2007-Saml2.0-Bearer-Https.xml")
   public class EchoService {
      ....
   ```

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 Sun 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 Configuring Two-Way SSL for a Client Application for additional details about two-way SSL.

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 *Administering Security for Oracle WebLogic Server*
- Set Up TLS in *Oracle WebLogic Remote Console Online Help*
- Configure Keystores in *Oracle WebLogic Remote Console Online Help*

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 jakarta.jws.WebMethod;
import jakarta.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 jakarta.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_PASSWOR
D");
      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 jakarta.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>
```

Persisting the State of a Request over SSL

Oracle WebLogic Server includes a 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.

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 Java SE 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.