BEA WebLogic Server®

WebLogic Web Services: Security

Version 10.0
Revised: April 28, 2008

Contents

4. Configuring Transport-Level Security

5. Configuring Access Control Security

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*WebLogic Web Services: Security.*

Document Scope and Audience

This document is a resource for software developers who program and configure security for WebLogic Web Services. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server or considering the use of WebLogic Web Services for a particular application.

The topics in this document are relevant during the design and development phases of a software project. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

This document does not address production phase administration, monitoring, or performance tuning Web Service topics. For links to WebLogic Server® documentation and resources for these topics, see "Related Documentation" on page 1-3.

It is assumed that the reader is familiar with J2EE and Web Services concepts, the Java programming language, Web technologies, and security concepts. This document emphasizes the value-added features provided by WebLogic Web Services and key information about how to use WebLogic Server features and facilities to get a WebLogic Web Service application up and running.

WebLogic Web Services Documentation Set

This document is part of a larger WebLogic Web Services documentation set that covers a comprehensive list of Web Services topics. The full documentation set includes the following documents:

- WebLogic Web Services: Getting Started—Describes the basic knowledge and tasks required to program a simple WebLogic Web Service. This is the first document you should read if you are new to WebLogic Web Services. The guide includes Web Service overview information, use cases and examples, iterative development procedures, typical JWS programming steps, data type information, and how to invoke a Web Service.
- WebLogic Web Services: Security—Describes how to program and configure message-level (digital signatures and encryption), transport-level, and access control security for a Web Service.
- WebLogic Web Services: Advanced Programming—Describes how to program more advanced features, such as Web Service reliable messaging, callbacks, conversational Web Services, use of JMS transport to invoke a Web Service, and SOAP message handlers.
- WebLogic Web Services: Reference—Contains all WebLogic Web Service reference documenation about JWS annotations, Ant tasks, reliable messaging WS-Policy assertions, security policy assertions, and deployment descriptors.

Guide to This Document

This document is organized as follows:

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide and the security features of WebLogic Web Services.

- Chapter 2, “Overview of Web Services Security,” provides overview information about the different types of security you can configure for a Web Service, when you should configure which, and so on.
- Chapter 3, “Configuring Message-Level Security,” describes how to configure message-level security for a Web Service, which includes digital signatures, encryption, SAML, and implementation of various specifications, such as WS-Security, WS-SecureConversations, WS-SecurityPolicy, and so on.
- Chapter 4, “Configuring Transport-Level Security,” describes how to secure the connection between a client application and a Web Service with Secure Sockets Layer (SSL).
- Chapter 5, “Configuring Access Control Security,” describes how to configure a Web Service to control the users who are allowed to access it.

Related Documentation

This document contains information specific to WebLogic Web Services security topics. See “WebLogic Web Services Documentation Set” on page 1-2 for a description of the related Web Services documentation.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- Developing WebLogic Server Applications is a guide to developing WebLogic Server components (such as Web applications and EJBs) and applications.
- Developing Web Applications, Servlets, and JSPs for WebLogic Server is a guide to developing Web applications, including servlets and JSPs, that are deployed and run on WebLogic Server.
- Programming WebLogic Enterprise Java Beans is a guide to developing EJBs that are deployed and run on WebLogic Server.
- Programming WebLogic XML is a guide to designing and developing applications that include XML processing.
- Deploying Applications to WebLogic Server is the primary source of information about deploying WebLogic Server applications. Use this guide for both development and production deployment of your applications.
- Configuring Applications for Production Deployment describes how to configure your applications for deployment to a production WebLogic Server environment.

- WebLogic Server Performance and Tuning contains information on monitoring and improving the performance of WebLogic Server applications.
- Overview of WebLogic Server System Administration is an overview of administering WebLogic Server and its deployed applications.

Samples for the Web Services Developer

In addition to this document, BEA Systems provides a variety of code samples for Web Services developers. The examples and tutorials illustrate WebLogic Web Services in action, and provide practical instructions on how to perform key Web Service development tasks.

BEA recommends that you run some or all of the Web Service examples before programming your own application that use Web Services.

For a description and location of the available code samples, see Samples for the Web Services Developer in the *WebLogic Web Services: Getting Started* document.

Release-Specific WebLogic Web Services Information

For release-specific information, see these sections in *WebLogic Server Release Notes*:

- WebLogic Server Features and Changes lists new, changed, and deprecated features.
- WebLogic Server Known and Resolved Issues lists known problems by general release, as well as service pack, for all WebLogic Server APIs, including Web Services.

Summary of WebLogic Web Services Security Features

For a full list of WebLogic Web Services features, including security features, see Summary of WebLogic Web Services Features in the *WebLogic Web Services: Getting Started* document.

CHAPTER 2

Overview of Web Services Security

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

Overview of Web Services Security

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

- Message-level security, in which data in a SOAP message is digitally signed or encrypted.

  See Chapter 3, "Configuring Message-Level Security."

- Transport-level security, in which SSL is used to secure the connection between a client application and the Web Service.

  See Chapter 4, "Configuring Transport-Level Security."

- Access control security, which specifies which roles are allowed to access Web Services.

  See Chapter 5, "Configuring Access Control Security."

What Type of Security Should You Configure?

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.

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

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 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 parts of the message be signed or encrypted.

CHAPTER 3

Configuring Message-Level Security

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

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 or X.509 tokens

See "Configuring Simple Message-Level Security: Main Steps" on page 3-4 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 Using the Administration Console" on page 3-22 for details.

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

Web Services Security Supported Standards

WebLogic Web Services implement the following OASIS Standard 1.1 Web Services Security (WS-Security 1.1) specifications, dated February 1, 2006:

- WS-Security Core Specification 1.1
- WS-Security 1.0 and 1.1
- Username Token Profile 1.1
- X.509 Token Profile 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 Secure Conversation

WebLogic Web Services also implement the Web Services Trust Language (WS-Trust) and Web Services Secure Conversation Language (WS-SecureConversation 1,2) specifications which together provide secure communication between Web Services and their clients (either other Web Services or standalone Java client applications). In particular, the WS-SecureConversation specification defines mechanisms for establishing and sharing security contexts, and deriving keys from security contexts, to enable a secure conversation. 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" on page 3-4 for detailed information about how the Web Services runtime environment uses security policy files. The Web Services SecurityPolicy specification is not final as of this release of WebLogic Server. For information about the elements of the Web Services SecurityPolicy 1.2 draft dated 21 February 2007, that are not supported in this release of WebLogic Server, see "Unsupported WS-SecurityPolicy 1.2 Assertions" on page 3-42.

Main Use Cases of Message-Level Security

The BEA 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 or encrypted: 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.
- 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 conversation 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 BEA schema for security policy. This release of WebLogic Server supports either security policy files that conform to the WS-SecurityPolicy 1.2 specification or the BEA Web Services security policy schema first included in WebLogic Server 9. For information about the packaged WS-SecurityPolicy 1.2 security policy files, see "Using WS-SecurityPolicy 1.2 Policy Files" on page 3-32. For information about the packaged BEA Web Services security policy schema files, see "BEA Web Services Security Policy Files" on page 3-46.

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.

Configuring Simple Message-Level Security: Main Steps

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

- The message-secured Web Service uses the pre-packaged 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" on page 3-4 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 BEA 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" on page 3-12.

  WARNING: If you plan to deploy the Web Service to a cluster in which different WebLogic Server instances are running on different computers, you *must* use a keystore and key pair other than the out-of-the-box ones, even for testing purposes. The reason is that the key pairs in the default WebLogic Server keystore, `DemoIdentity.jks`, are not guaranteed to be the same across WebLogic Servers running on different machines. If you were to use the default keystore, the WSDL of the deployed Web Service would specify the public key from one of these keystores, but the invoke of the service might actually be handled by a server running on a different computer, and in this case the server's private key would not match the published public key and the invoke would fail. This problem only occurs if you use the default keystore and key pairs in a cluster, and is easily resolved by using your own keystore and key pairs.

- The client invoking the Web Service uses a username token to authenticate itself, rather than an X.509 token.
- The client invoking the Web Service is a stand-alone Java application, rather than a module running in WebLogic Server.

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

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

- Programming the JWS File
- Iterative Development of WebLogic Web Services
- Invoking Web Services

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 pre-packaged 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" on page 3-8, which describes how to specify *any* policy file.

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

   See Iterative Development of WebLogic Web Services.

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

   You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   See *Obtaining Private Keys and Digital Signatures*.

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" on page 3-7 for information about how WebLogic Server ensures that the client's certificate is valid.

   WARNING: BEA requires a key length of 1024 bits or larger.

   You can use Sun Microsystem's `keytool` utility to perform this step.

   See *Obtaining Private Keys and Digital Signatures*.

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

   See *Users, Groups, and Security Roles*.

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" on page 3-27.

7. Recompile your client application.

   See Invoking Web Services 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" on page 3-12
- "Creating and Using a Custom Policy File" on page 3-18
- "Configuring and Using Security Contexts and Derived Keys (WS-SecureConversation)" on page 3-19
- "Associating Policy Files at Runtime Using the Administration Console" on page 3-22
- "Using Security Assertion Markup Language (SAML) Tokens For Identity" on page 3-22
- "Invoking a Message-Secured Web Service From a Client Running in a WebLogic Server Instance" on page 3-16
- "Associating a Web Service with a Security Configuration Other Than the Default" on page 3-26

See "Using System Properties to Debug Message-Level Security" on page 3-27 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 which lists all the individual certificates trusted by WebLogic Server, and then ensure that the client uses one of these registered certificates.

For more information, see SSL Certificate Validation.

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.

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.

WARNING: As is true for all JWS annotations, the `@Policy` annotation cannot be overridden at runtime, which means that the policy file you specify at buildtime using the annotation will always be associated with the Web Service. This means, for example, that although you can view the associated policy file at runtime using the Administration Console, you cannot delete (unassociate) it. You can, however, associate additional policy files, as described in "Associating Policy Files at Runtime Using the Administration Console" on page 3-22.

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

- To specify one of the pre-packaged 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-Https-BasicAuth.xml")
  ```

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

  See "Using Policy Files for Message-Level Security Configuration" on page 3-4 for information on the various types of message-level security provided by the pre-packaged policy files.

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

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

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

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

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

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

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

  See Creating Shared J2EE Libraries and Optional Packages for information on creating shared libraries and setting up your environment so the Web Service can find the shared policy files.

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

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

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

Listing 3-1 Using @Policy and @Policies Annotations

```
package wssp12.wss10;

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

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

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

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

    return s;
  }

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

  @WebMethod
```

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

  @WebMethod
  @Policies({
      @Policy(uri="policy:Wssp1.2-Wss1.0-X509-Basic256.xml",
direction=Policy.Direction.inbound),
      @Policy(uri="policy:Wssp1.2-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-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 pre-packaged 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-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-EncryptBody.xml` policy file specifies that the body of both the request and response SOAP messages be encrypted.

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: Main Steps" on page 3-4, 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, BEA 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.

   WARNING: BEA requires that the key length be 1024 bits or larger.

   You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   See *Obtaining Private Keys and Digital Signatures*.

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 a identity keystore which you can also use in this step.

   You can use WebLogic's `ImportPrivateKey` utility and Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   See *Creating a Keystore and Loading Private Keys and Trusted Certificate Authorities Into the Keystore*.

3. Using the Administration Console, configure WebLogic Server to locate the keystore you created in the preceding step. If you are using a keystore that has already been configured for WebLogic Server, you do not need to perform this step.

   See *Configuring Keystores for Production*.

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

   See Create a Web Service security configuration.

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

   See Specify the Key Pair Used to Sign SOAP Messages. In the procedure, when you create the properties used to identify the keystore and key pair, enter the exact value for the Name of each property (such as `IntegrityKeyStore`, `IntegrityKeyStorePassword`, and so on), but enter the value that identifies your own previously-created keystore and key pair in the Value fields.

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

   See Create keystore used by SOAP message encryption. In the procedure, when you create the properties used to identify the keystore and key pair, enter the exact value for the Name of each property (such as `ConfidentialityKeyStore`. `ConfidentialityKeyStorePassword`, and so on), but enter the value that identifies your own previously-created keystore and key pair in the Value fields.

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

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

If the security policy file of the Web Service specifies that the SOAP request must be encrypted, then the Web Services client runtime automatically gets the server's certificate from the policy file that is attached to the WSDL of the service, and uses it for the encryption. If, however, the policy file is not attached to the WSDL, or the entire WSDL itself is not available, then the client

application must use a client-side copy of the policy file; for details, see"Using a Client-Side Security Policy File" on page 3-27.

Listing 3-2 shows a Java client application that invokes the message-secured WebLogic Web Service described by the JWS file in "Updating the JWS File With the Security-Related Annotations" on page 5-4. The client application takes five arguments:

- Client username for client authentication
- Client password for client authentication
- Client private key file
- Client digital certificate
- WSDL of the deployed Web Service

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

Listing 3-2 Client Application Invoking a Message-Secured Web Service

```
package examples.webservices.security_jws.client;

import weblogic.security.SSL.TrustManager;

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

import javax.xml.rpc.Stub;
import java.util.List;
import java.util.ArrayList;

import java.security.cert.X509Certificate;

/**
 *  Copyright (c) 2005 by BEA Systems. All Rights Reserved.
 */
public class SecureHelloWorldClient {
  public static void main(String[] args) throws Throwable {

    //username or password for the UsernameToken
    String username = args[0];
    String password = args[1];
```

```
    //client private key file
    String keyFile = args[2];

    //client certificate
    String clientCertFile = args[3];

    String wsdl = args[4];

    SecureHelloWorldService service = new SecureHelloWorldService_Impl(wsdl +
"?WSDL" );

    SecureHelloWorldPortType port = service.getSecureHelloWorldServicePort();

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

    //client side BinarySecurityToken credential provider -- x509
    CredentialProvider cp = new ClientBSTCredentialProvider(clientCertFile,
keyFile);
    credProviders.add(cp);

    //client side UsernameToken credential provider
    cp = new ClientUNTCredentialProvider(username, password);
    credProviders.add(cp);

    Stub stub = (Stub)port;
    stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
credProviders);

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

    String response = port.sayHello("World");
    System.out.println("response = " + response);
  }
}
```

The main points to note about the preceding code are:

- Import the WebLogic security `TrustManager` API:

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

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

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

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

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

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

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

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

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

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

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

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

Invoking a Message-Secured 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: Main Steps" on page 3-4, it is assumed that a *stand-alone* client application invokes the message-secured Web Service. Sometimes, however, the client is itself running in a WebLogic Server instance, as part of an EJB, a servlet, or another Web Service. In

this case, you can use the core WebLogic Server security framework to configure the credential providers and trust manager so that your EJB, servlet, or JWS code contains only the simple invoke of the secured operation and no other security-related API usage. The following procedure describes the high level steps you must perform to make use of the core WebLogic Server security framework in this use case.

1. In your EJB, servlet, or JWS code, invoke the Web Service operation as if it were *not* configured for message-level security. Specifically, do not create a `CredentialProvider` object that contains username or X.509 tokens, and do not use the `TrustManager` core security API to validate the certificate from the WebLogic Server hosting the secure Web Service. The reason you should not use these APIs in your client code is that the Web Services runtime will perform this work for you.

2. Using the Administration Console, configure the required credential mapping providers of the core security of the WebLogic Server instance that hosts your client application. The list of required credential mapper providers depends on the policy file that is attached to the Web Service you are invoking. Typically, you must configure the credential mapper providers for both username/password and X.509 certificates. See Configuring a WebLogic Credential Mapping Provider.

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

3. Using the Administration Console, create the actual credential mappings in the credential mapping providers you configured in the preceding step. You must map the user principal, associated with the client running in the server, to the credentials that are valid for the Web Service you are invoking. See Configuring a WebLogic Credential Mapping Provider.

4. Using the Administration Console, configure the core WebLogic Server security framework to trust the X.509 certificate of the invoked Web Service. See Configuring the Credential Lookup and Validation Framework.

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" on page 3-27. 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.

Creating and Using a Custom Policy File

Although WebLogic Server includes a number of pre-packaged 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" on page 3-4 for general information about security policy files and how they are used for message-level security configuration.

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 pre-packaged 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 pre-packaged files for the other categories (`Wssp1.2-SignBody.xml` and `Wssp1.2-EncryptBody`). In other words, you can mix and match the number and content of the policy files that you associate with a Web Service. In this case, however, you must always ensure yourself that the multiple files do not contradict each other.

Your custom policy file needs to comply with the standard format and assertions defined in WS-SecurityPolicy 1.2. Note, however, that this release of WebLogic Server does not completely implement WS-SecurityPolicy 1.2. For more information, see "Unsupported WS-SecurityPolicy 1.2 Assertions" on page 3-42. The root element of your WS-SecurityPolicy file must be `<Policy>` and include the following namespace declarations:

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

You can also use the pre-packaged WS-SecurityPolicy files as templates to create your own custom files. See "Using WS-SecurityPolicy 1.2 Policy Files" on page 3-32.

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.

Configuring and Using Security Contexts and Derived Keys (WS-SecureConversation)

BEA provides three pre-packaged WS-SecurityPolicy files (`Wssp1.2-Wssc200502-Bootstrap-Https.xml`, `Wssp1.2-Wssc200502-Bootstrap-Wss1.0.xml`, and `Wssp1.2-Wssc200502-Bootstrap-Wss1.1.xml`) to configure security contexts and derived keys, as described by the WS-SecureConversation 1.2 (2005/2) specification. It is recommended that you use the pre-packaged 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 2005/2 Policies" on page 3-37 for more information about these files.

WARNING: If you are deploying a Web Service that uses shared security contexts to a cluster, then you are required to also configure cross-cluster session state replication. For details, see Failover and Replication in a Cluster.

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" on page 3-27. The only real difference is that you can use the `weblogic.wsee.security.wssc.utils.WSSCClientUtil` API to explicitly cancel the secure context token.

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.

Listing 3-3 shows a simple example of a client application invoking a Web Service that is associated with a pre-packaged security policy file that enables secure conversations; the sections in bold which are relevant to security contexts are discussed after the example:

Listing 3-3 Client Application Using WS-SecureConversation

```
package examples.webservices.wssc.client;
import weblogic.security.SSL.TrustManager;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.bst.StubPropertyBSTCredProv;
import weblogic.wsee.security.wssc.utils.WSSCClientUtil;
import weblogic.wsee.security.util.CertUtils;

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

/**
 *  Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
public class WSSecureConvClient {
  public static void main(String[] args) throws Throwable {


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

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

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

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

    Stub stub = (Stub)port;
```

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

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

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

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

  }
}
```

The points to notice in the preceding example are:

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

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

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

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

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

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

Associating Policy Files at Runtime Using the Administration Console

The simple message-level configuration procedure, documented in "Configuring Simple Message-Level Security: Main Steps" on page 3-4, describes how to use the `@Policy` and `@Policies` JWS annotations in the JWS file that implements your Web Service to specify one or more policy files that are associated with your service. This of course implies that you must already know, at the time you program your Web Service, which policy files you want to associate with your Web Service and its operations. This might not always be possible, which is why you can also associate policy files at *runtime*, after the Web Service has been deployed, using the Administration Console.

You can use no `@Policy` or `@Policies` JWS annotations at all in your JWS file and associate policy files only at runtime using the Administration Console, or you can specify some policy files using the annotations and then associate additional ones at runtime. However, once you associate a policy file using the JWS annotations, you cannot change this association at runtime using the Administration Console.

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

See Associate a policy file with a Web Service for detailed instructions on using the Administration Console to associate a policy file at runtime.

Using Security Assertion Markup Language (SAML) Tokens For Identity

In the simple Web Services configuration procedure, described in "Configuring Simple Message-Level Security: Main Steps" on page 3-4, it is assumed that users use username tokens to authenticate themselves. Because WebLogic Server implements the Web Services Security: SAML Token Profile 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.

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

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

See SAML Token Profile Support in WebLogic Web Services, as well as the Web Services Security: SAML Token Profile specification itself, for details about these confirmation methods.

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).

It is also assumed in the following procedure that you have followed the steps in "Configuring Simple Message-Level Security: Main Steps" on page 3-4 and now want to enable the additional use case of using SAML tokens, rather than usename tokens, for identity.

To use SAML tokens for identity, follow these steps:

1. Using the Administration Console, configure a SAML identity assertion and credential mapping provider. This step configures the core WebLogic Server security subsystem. For details, see:
   - Configuring a SAML Identity Assertion Provider
   - Configuring a SAML Credential Mapping Provider
2. 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` or `holder-of-key`). Note that this release of WebLogic Server does not support the use of SAML with WS-SecurityPolicy 1.2 policy files. Instead, you must use security policy files written under BEA's security policy schema. The exact syntax depends on the type of confirmation method you want to configure (`sender-vouches` or `holder-of-key`).

   To specify the sender-vouches confirmation method:

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

– 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-w
ssecurity-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:

– 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.

– 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-w
ssecurity-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>
```

– By default, the WebLogic Web Services runtime always validates the X.509 certificate specified in the `<KeyInfo>` assertion of any associated WS-Policy file. To disable this validation when using SAML holder-of-key assertions, you must configure the Web Service security configuration associated with the Web service by setting a property on the SAML token handler. See Disable X.509 certificate validation when using SAML holder_of_key assertions for information on how to do this using the Administration Console.

See “Creating and Using a Custom Policy File” on page 3-18 for additional information about creating your own security policy file. See Security Policy Assertion Reference for reference information about the assertions.

3. 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 BEA'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 pre-packaged `Wssp1.2-EncryptBody.xml` and `Wssp1.2-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.

4. Recompile and redeploy your Web Service as part of the normal iterative development process.

   See Iterative Development of WebLogic Web Services.

5. Create a client application that runs in a WebLogic Server instance to invoke the main Web Service using SAML as identity. See "Invoking a Message-Secured Web Service From a Client Running in a WebLogic Server Instance" on page 3-16 for details.

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

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

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

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

1. Create a Web Service security configuration 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 for additional information and an example.

   WARNING: 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 for more details.

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

   See Iterative Development of WebLogic Web Services.

WARNING: 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.

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 3-1 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.
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" on page 3-4 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.

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 a Web Service from a Stand-alone Client: Main Steps.

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" on page 3-18 for information about creating security policy files.

2. Update the `build.xml` file that builds your client application by specifying to the `clientgen` Ant task that it should generate additional `get`*`XXX`*`Port()` methods in the JAX-RPC stub, where *`XXX`* refers to the name of the Web Service. These methods are later used by the client application to load the client-side policy files.

   See "Updating clientgen to Generate Methods That Load Policy Files" on page 3-29.

3. Update your Java client application to load the client-side policy files using the additional `get`*`XXX`*`Port()` methods that the `clientgen` Ant task generates.

   See "Updating a Client Application To Load Policy Files" on page 3-30.

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.

Updating clientgen to Generate Methods That Load Policy Files

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

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

See "Updating a Client Application To Load Policy Files" on page 3-30 for a description of the additional methods that are generated and how to use them in a client application.

Updating a Client Application To Load Policy Files

When you set `generatePolicyMethods="true"` for `clientgen`, the Ant task generates additional methods in the implementation of the JAX-RPC `Service` interface that you can use to load policy files, where *XXX* refers to the name of the Web Service. You can use either an Array or Set of policy files to associate multiple files to a Web Service. If you want to associate just a single policy file, create a single-member Array or Set.

- `get`*`XXX`*`Port(String operationName, java.util.Set<java.io.InputStream> inbound, java.util.Set<java.io.InputStream> outbound)`

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

- `get`*`XXX`*`Port(String operationName, java.io.InputStream[] inbound, java.io.InputStream[] outbound)`

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

- `get`*`XXX`*`Port(java.util.Set<java.io.InputStream> inbound, java.util.Set<java.io.InputStream> outbound)`

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

- `get`*`XXX`*`Port(java.io.InputStream[] inbound, java.io.InputStream[] outbound)`

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

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

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

- `get`*`XXX`*`Port(java.io.InputStream policyInputStream);`

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

- `get`*`XXX`*`Port(java.io.InputStream policyInputStream, boolean inbound, boolean outbound);`

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

Listing 3-4 shows an example of using these policy methods in a simple client application; the code in bold is described after the example.

Listing 3-4 Loading Policies in a Client Application

```
package examples.webservices.client_policy.client;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;

import java.io.FileInputStream;
import java.io.IOException;

/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the ClientPolicyService Web service.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class Main {

  public static void main(String[] args)
      throws ServiceException, RemoteException, IOException {

      FileInputStream [] inbound_policy_array = new FileInputStream[2];
      inbound_policy_array[0] = new FileInputStream(args[1]);
      inbound_policy_array[1] = new FileInputStream(args[2]);

      FileInputStream [] outbound_policy_array = new FileInputStream[2];
      outbound_policy_array[0] = new FileInputStream(args[1]);
      outbound_policy_array[1] = new FileInputStream(args[2]);

      ClientPolicyService service = new ClientPolicyService_Impl(args[0] +
"?WSDL");
```

```
      // standard way to get the Web Service port
      ClientPolicyPortType normal_port = service.getClientPolicyPort();

      // specify an array of policy files for the request and response
      // of a particular operation
      ClientPolicyPortType array_of_policy_port =
service.getClientPolicyPort("sayHello", inbound_policy_array,
outbound_policy_array);

    try {
      String result = null;
      result = normal_port.sayHello("Hi there!");
      result = array_of_policy_port.sayHello("Hi there!");
      System.out.println( "Got result: " + result );
    } catch (RemoteException e) {
      throw e;
    }
  }
}
```

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

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

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 `BEA_HOME/WL_HOME/server/lib/weblogic.jar`. Within `weblogic.jar`, the policy files are located in `/weblogic/wsee/policy/runtime`.

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/200512"
>
```

Note: This release of WebLogic Server also includes five security policy files (first included in WebLogic Server 9) written under a proprietary BEA Web Services security policy schema. These security policy files, described in "BEA Web Services Security Policy Files" on page 3-46 have the following namespace:

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

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

- "Transport Level Policies" on page 3-33
- "Protection Assertion Policies" on page 3-34
- "WS-Security 1.0 Username and X509 Token Policies" on page 3-34
- "WS-Security 1.1 Username and X509 Token Policies" on page 3-36
- "WS-SecureConversation 2005/2 Policies" on page 3-37

In addition, see "Choosing a Policy" on page 3-38 and "Smart Policy Selection" on page 3-39 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:

Table 3-2 Transport Level Policies

Policy File	Description
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 3-2 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 can not be used alone; they should be used only in combination with X.509 Token Policies. For example, you might use `Wssp1.2-Wss1.1-X509-Basic256.xml` together with `Wssp1.2-SignBody.xml`. The following policy files provide for the protection of message parts by signing or encryption:

Table 3-3 Protection Assertion Policies

Policy File	Description
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.

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 3-4 WS-Security 1.0 Policies

Policy File	Description
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 aes256.
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 aes256.
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 aes256 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 3-5 WS-Security 1.1 Username and X509 Token Policies

Policy File	Description
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-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.
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.

Table 3-5 WS-Security 1.1 Username and X509 Token Policies

Policy File	Description
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 2005/2 Policies

The following policies implement WS-SecureConversation 2005/2:

Table 3-6 WS-SecureConversation Policies

Policy File	Description
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.

Table 3-6 WS-SecureConversation Policies

Policy File	Description
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.

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, BEA 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 BEA WebLogic Server 9.x style policy, unless you require features that are not yet supported by WS-SecurityPolicy 1.2 policies.

Examples of features that may require you to use BEA WebLogic Server 9.x style policies include SAML 1.1 and element-level signature and encryption.

- Use transport-level policies (`Wssp1.2-Https-*.xml`) only where message-level security is not required.
- Use WS-Security 1.0 policies if you require interoperability. Use one of the following, depending on your authentication requirements and credential availability:
  - `Wssp1.2-Wss1.0-UsernameToken-Plain-X509-Basic256.xml`
  - `Wssp1.2-Wss1.0-UsernameToken-Digest-X509-Basic256.xml`
  - `Wssp1.2-Wss1.0-X509-Basic256.xml`
- Use WS-Security 1.1 policies if you have strong security requirements. Use one of the following:
  - `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-Wssc200502-Bootstrap-Wss1.0.xml`
  - `Wssp1.2-Wssc200502-Bootstrap-Wss1.1.xml`

Smart Policy Selection

You can configure multiple policy alternatives for a single Web Service by creating a custom policy. At runtime, WebLogic Server selects which of the configured policies to apply. It excludes policies that are not supported or have conflicting assertions and selects the appropriate policy to verify incoming messages and build the response messages. For example, a single Web Service can be configured with security policies that can handle either WS-Security 1.0 or WS-Security 1.1 requests.

Listing 3-5 gives an example of a security policy that supports both WS-Security 1.0 and WS-Security 1.1. Each policy alternative is enclosed in a `<wsp:All>` element.

Listing 3-5 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/Inc
ludeToken/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/Inc
ludeToken/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/Inc
ludeToken/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/Inc
ludeToken/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>
```

Unsupported WS-SecurityPolicy 1.2 Assertions

The Web Services SecurityPolicy specification is not final as of this release of WebLogic Server. The implementation of WS-SecurityPolicy in this release of WebLogic Server is based on OASIS WS-SecurityPolicy 1.2 Editors Draft dated 19 June 2006 (WS-SX WSSP 1.2 (06/2006). Table 3-7 details the elements of the Web Services SecurityPolicy 1.2 draft dated 21 February 2007, that are not supported in this release of WebLogic Server:

Table 3-7 Web Services SecurityPolicy 1.2 Unsupported Assertions

Spec Section	Assertion	Remarks
4.1.2	SignedElements	Not yet supported
4.2.2	EncryptedElements	Not yet supported
4.3.1	RequiredElements	Not yet supported
4.3.2	RequiredParts	Not yet supported
5.1.1	TokenInclusion	includeTokenPolicy=Once is not supported
5.4.1	UsernameToken	Only <sp:UsernameToken10> is supported in this release, <sp:UsernameToken11> and Password Derived Keys are not yet supported.
5.4.2	IssuedToken	Not yet supported

Table 3-7 Web Services SecurityPolicy 1.2 Unsupported Assertions

Spec Section	Assertion	Remarks
5.4.3	X509Token	Supported: <sp:WssX509V3Token10> <sp:WssX509V3Token11> Not yet supported: <sp:WssX509Pkcs7Token10> <sp:WssX509Pkcs7Token11> <sp:WssX509PkiPathV1Token10> <sp:WssX509PkiPathV1Token11> <sp:WssX509V1Token10> <sp:WssX509V1Token11>
5.4.4	KerberosToken	Not yet supported
5.4.5	SpnegoContextToken	Not yet supported
5.4.6	SecurityContextToken	Not yet supported
5.4.7	SecureConversationToken	Not yet supported: <sp:IssuerName> <sp:RequireExternalUriReference> For <sp:BootStrapPolicy>, we support all the features that are supported in the normal policy binding. <sp:Strict> layout and Derived Key endorsing with Derived Key is not supported in this release.
5.4.8	SamlToken	Not yet supported
5.4.9	RelToken	Not supported
5.4.11	KeyValueToken	Not yet supported
6.3	ProtectionOrder	<sp:SignBeforeEncrypting> is supported. <sp:EncryptBeforeSigning> is not yet supported
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

Table 3-7 Web Services SecurityPolicy 1.2 Unsupported Assertions

Spec Section	Assertion	Remarks
6.7/7.2	Security Header Layout Property	<sp:Strict> is partially supported.
7.1	AlgorithmSuite	Not yet supported: /sp:AlgorithmSuite/wsp:Policy/sp:XPathFilter20 assertion /sp:AlgorithmSuite/wsp:Policy/sp:XPath10 assertion /sp:AlgorithmSuite/wsp:Policy/sp:SoapNormalization10
8.1	SupportingTokens	Not yet supported: ../sp:SignedParts assertion ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion
8.2	SignedSupportingTokens	Not yet supported: ../sp:SignedParts assertion ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion The runtime will not be able to sign the supporting token in cases where the Token is not in the Message (such as for includeTokenPolicy=Never/Once).
8.3	EndorsingSupportingTokens	Not yet supported: ../sp:SignedParts assertion ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion <sp:RequireDerivedKeys/> is not supported inside the EndorsingSupportingTokens. 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).

Table 3-7 Web Services SecurityPolicy 1.2 Unsupported Assertions

Spec Section	Assertion	Remarks
8.4	SignedEndorsingSupportingTokens	Not yet supported: ../sp:SignedParts assertion ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion <sp:RequireDerivedKeys/> is not supported inside the SignedEndorsingSupportingTokens. The runtime will not be able to sign/endorse the supporting token in cases where the Token is not in the Message (such as for includeTokenPolicy=Never/Once).
8.5	SignedEncrtptedSupportingTokens	Not yet supported: ../sp:SignedParts assertion ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion <sp:RequireDerivedKeys/> is not support inside the SignedEncryptedSupportingTokens. The runtime will not be able to sign the supporting token in cases where the Token is not in the Message (such as for includeTokenPolicy=Never/Once).
8.6	EncryptedSupportingTokens	Not yet supported: ../sp:SignedParts assertion ../sp:SignedElements assertion ../sp:EncryptedParts assertion ../sp:EncryptedElements assertion The UserName Token is only supporting token type that is supported.
8.7	EndorsingEncryptedSupportingTokens	Not yet supported
8.8	SignedEndorsingEncryptedSupportingTokens	Not yet supported

Table 3-7 Web Services SecurityPolicy 1.2 Unsupported Assertions

Spec Section	Assertion	Remarks
9.1	WSS10 Assertion	Everything is supported with the exception of <sp:MustSupportRefExternalURI> and <sp:MustSupportRefEmbeddedToken>.
9.2	WSS11 Assertion	Everything is supported with the exception of <sp:MustSupportRefExternalURI> and <sp:MustSupportRefEmbeddedToken>.
10.1	Trust10 Assertion	Not yet supported: MustSupportClientChallenge MustSupportServerChallenge We only support this assertion in the WS-SecureConversation policy.

BEA Web Services Security Policy Files

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 BEA schema for security policy. This release of WebLogic Server supports both security policy files that conform to the WS-SecurityPolicy 1.2 specification and the files written under the BEA Web Services security policy schema first included in WebLogic Server 9.

WARNING: WS-SecurityPolicy 1.2 policy files and BEA proprietary Web Services security policy schema files are not mutually compatible; you cannot define both types of policy file in the same Web Service. If you want to use WS-Security 1.1 features, you must use the WS-SecurityPolicy 1.2 policy file format.

This section describes the set of pre-packaged BEA Web Services security policy schema files included in WebLogic Server. These policy files are all abstract; see "Abstract and Concrete Policy Files" on page 3-47 for details.

Note: The policy assertions used in these security policy files to configure message-level security for a WebLogic Web Service are based on the assertions described in the December 18, 2002 version of the *Web Services Security Policy Language* (WS-SecurityPolicy) specification. This means that although the exact syntax and usage of the assertions in WebLogic Server are different, they are similar in meaning to those

described in the specification. The assertions are *not* based on later updates of the specification.

The pre-packaged BEA Web Services security policy files are:

- Auth.xml—Specifies that the client must authenticate itself. Can be used on its own, or together with `Sign.xml` and `Encrypt.xml`.
- Sign.xml—Specifies that the SOAP messages are digitally signed. Can be used on its own, or together with `Auth.xml` and `Encrypt.xml`.
- Encrypt.xml—Specifies that the SOAP messages are encrypted. Can be used on its own, or together with `Auth.xml` and `Sign.xml`.
- Wssc-dk.xml—Specifies that the client and service share a security context when multiple messages are exchanged and that derived keys are used for encryption and digital signatures, as described by the WS-SecureConversation specification.

  Note: This pre-packaged policy file is meant to be used on its own and *not* together with `Auth.xml`, `Sign.xml`, `Encrypt.xml`, or `Wssc-sct.xml`. Also, BEA recommends that you use this policy file, rather than Wssc-sct.xml, if you want the client and service to share a security context, due to its higher level of security.

- Wssc-sct.xml—Specifies that the client and service share a security context when multiple messages are exchanged, as described by the WS-SecureConversation specification.

  Note: This pre-packaged policy file is meant to be used on its own and *not* together with `Auth.xml`, `Sign.xml`, `Encrypt.xml`, or `Wssc-dk.xml`. Also, BEA provides this policy file to support the various use cases of the WS-SecureConversation specification; however, BEA recommends that you use the Wssc-dk.xml policy file, rather than Wssc-sct.xml, if you want the client and service to share a security context, due to its higher level of security.

Abstract and Concrete Policy Files

The WebLogic Web Services runtime environment recognizes two slightly different types of security policy files: *abstract* and *concrete*.

Abstract policy files do not explicitly specify the security tokens that are used for authentication, encryption, and digital signatures, but rather, the Web Services runtime environment determines the security tokens when the Web Service is deployed. Specifically, this means the `<Identity>` and `<Integrity>` elements (or assertions) of the policy files do not contain a `<SupportedTokens><SecurityToken>` child element, and the `<Confidentiality>` element policy file does not contain a `<KeyInfo><SecurityToken>` child element.

If your Web Service is associated with only the pre-packaged policy files, then client authentication requires username tokens. Web Services support only one type of token for encryption and digital signatures (`X.509`), which means that in the case of the `<Integrity>` and `<Confidentiality>` elements, concrete and abstract policy files end up being essentially the same.

If your Web Service is associated with an abstract policy file and it is published as an attachment to the WSDL (which is the default behavior), the static WSDL file packaged in the Web Service archive file (JAR or WAR) will be slightly different than the dynamic WSDL of the deployed Web Service. This is because the static WSDL, being abstract, does not include specific `<SecurityToken>` elements, but the dynamic WSDL *does* include these elements because the Web Services runtime has automatically filled them in when it deployed the service. For this reason, in the code that creates the JAX-RPC stub in your client application, ensure that you specify the dynamic WSDL or you will get a runtime error when you try to invoke an operation:

```
HelloService service = new HelloService(Dynamic_WSDL);
```

You can specify either the static or dynamic WSDL to the `clientgen` Ant task in this case. See Browsing to the WSDL of the Web Service for information on viewing the dynamic WSDL of a deployed Web Service.

Concrete policy files explicitly specify the details of the security tokens at the time the Web Service is programmed. Programmers create concrete security policy files when they know, at the time they are programming the service, the details of the type of authentication (such as using `x509` or `SAML` tokens); whether multiple private key and certificate pairs from the keystore are going to be used for encryption and digital signatures; and so on.

Auth.xml

The WebLogic Server `Auth.xml` file, shown below, specifies that the client application invoking the Web Service must authenticate itself with one of the tokens (username or X.509) that support authentication.

Because the pre-packaged BEA Web Services security policy schema files are abstract, there is no specific username or X.509 token assertions in the `Auth.xml` file at development-time. Depending on how you have configured security for WebLogic Server, either a username token, an X.509 token, or both will appear in the actual runtime-version of the `Auth.xml` policy file associated with your Web Service. Additionally, if the runtime-version of the policy file includes an X.509 token and it is applied to a client invoke, then the entire body of the SOAP message is signed.

If you want to specify that *only* X.509, and never username tokens, be used for identity, or want to specify that, when using X.509 for identity, only certain parts of the SOAP message be signed, then you must create a custom security policy file.

Listing 3-6 Auth.xml

```
<?xml version="1.0"?>

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

  >

  <wssp:Identity/>

</wsp:Policy>
```

Sign.xml

The WebLogic Server `Sign.xml` file specifies that the body and WebLogic-specific system headers of the SOAP message be digitally signed. It also specifies that the SOAP message include a Timestamp, which is digitally signed, and that the token used for signing is also digitally signed. The token used for signing is included in the SOAP message.

The following headers are signed when using the `Sign.xml` security policy file:

- `SequenceAcknowledgement`
- `AckRequested`
- `Sequence`
- `Action`
- `FaultTo`
- `From`
- `MessageID`
- `RelatesTo`
- `ReplyTo`
- `To`
- `SetCookie`

- `Timestamp`

The WebLogic Server `Sign.xml` file is shown below:

Listing 3-7 Sign.xml

```
<?xml version="1.0"?>

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
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://www.bea.com/wls90/security/policy/wsee#part">
          wls:SystemHeaders()
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1" />
      <wssp:MessageParts
          Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
          wls:SecurityHeader(wsu:Timestamp)
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1" />
      <wssp:MessageParts
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
          wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>

  </wssp:Integrity>
```

```
  <wssp:MessageAge/>
</wsp:Policy>
```

Encrypt.xml

The WebLogic Server `Encrypt.xml` file specifies that the entire body of the SOAP message be encrypted. By default, the encryption token is *not* included in the SOAP message.

Listing 3-8 Encrypt.xml

```
<?xml version="1.0"?>

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

  <wssp:Confidentiality>
    <wssp:KeyWrappingAlgorithm URI="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
    <wssp:Target>
      <wssp:EncryptionAlgorithm
         URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <wssp:MessageParts
         Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
         wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:KeyInfo/>
  </wssp:Confidentiality>

</wsp:Policy>
```

Wssc-dk.xml

Specifies that the client and Web Service share a security context, as described by the WS-SecureConversation specification, and that a derived key token is used. This ensures the highest form of security.

This policy file provides the following configuration:

- A derived key token is used to sign all system SOAP headers, the timestamp security SOAP header, and the SOAP body.

- A derived key token is used to encrypt the body of the SOAP message. This token is different from the one used for signing.
- Each SOAP message uses its own pair of derived keys.
- For both digital signatures and encryption, the key length is 16 (as opposed to the default 32)
- The lifetime of the security context is 12 hours.

If you need to change the default security context and derived key behavior, you will have to create a custom security policy file, described in later sections.

WARNING: If you specify this pre-packaged security policy file, you should not also specify any other pre-packaged security policy file.

Listing 3-9 Wssc-dk.xml

```
<?xml version="1.0"?>

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
  >

  <wssp:Integrity SupportTrust10="true">
    <wssp:SignatureAlgorithm
URI="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
    <wssp:CanonicalizationAlgorithm
URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>

    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
          wls:SystemHeaders()
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
```

```
          wls:SecurityHeader(wsu:Timestamp)
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
     <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
       wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:SupportedTokens>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"
        DerivedFromTokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
        <wssp:Claims>
          <wssp:Label>WS-SecureConversationWS-SecureConversation</wssp:Label>
          <wssp:Length>16</wssp:Length>
        </wssp:Claims>
      </wssp:SecurityToken>
    </wssp:SupportedTokens>

  </wssp:Integrity>

  <wssp:Confidentiality SupportTrust10="true">

    <wssp:Target>
      <wssp:EncryptionAlgorithm
URI="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
     <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
wsp:Body()</wssp:MessageParts>
    </wssp:Target>

    <wssp:KeyInfo>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"
        DerivedFromTokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
        <wssp:Claims>
          <wssp:Label>WS-SecureConversationWS-SecureConversation</wssp:Label>
          <wssp:Length>16</wssp:Length>
        </wssp:Claims>
      </wssp:SecurityToken>
    </wssp:KeyInfo>

  </wssp:Confidentiality>

  <wssp:MessageAge/>

</wsp:Policy>
```

Wssc-sct.xml

Specifies that the client and Web Service share a security context, as described by the WS-SecureConversation specification. In this case, security context tokens are used to encrypt and sign the SOAP messages, which differs from Wssc-dk.xml in which derived key tokens are used. The `Wssc-sct.xml` policy file is provided to support all the use cases of the specification; for utmost security, however, BEA recommends you always use Wssc-dk.xml when specifying shared security contexts due to its higher level of security.

This security policy file provides the following configuration:

- A security context token is used to sign all system SOAP headers, the timestamp security SOAP header, and the SOAP body.
- A security context token is used to encrypt the body of the SOAP message.
- The lifetime of the security context is 12 hours.

If you need to change the default security context and derived key behavior, you will have to create a custom security policy file, described in later sections.

WARNING: If you specify this pre-packaged security policy file, you should not also specify any other pre-packaged security policy file.

Listing 3-10 Wssc-sct.xml

```
<?xml version="1.0"?>

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
  xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
  >

  <wssp:Integrity SupportTrust10="true">
    <wssp:SignatureAlgorithm
URI="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
    <wssp:CanonicalizationAlgorithm
URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>

    <wssp:Target>
     <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

```
<wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
         wls:SystemHeaders()
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts
Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
         wls:SecurityHeader(wsu:Timestamp)
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
     <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
      wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>

    <wssp:SupportedTokens>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
      </wssp:SecurityToken>
    </wssp:SupportedTokens>

  </wssp:Integrity>

  <wssp:Confidentiality SupportTrust10="true">

    <wssp:Target>
      <wssp:EncryptionAlgorithm
URI="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
     <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
wsp:Body()</wssp:MessageParts>
    </wssp:Target>

    <wssp:KeyInfo>
      <wssp:SecurityToken IncludeInMessage="true"
        TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
      </wssp:SecurityToken>
    </wssp:KeyInfo>
  </wssp:Confidentiality>

  <wssp:MessageAge />

</wsp:Policy>
```

CHAPTER 4

Configuring Transport-Level Security

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

Configuring Transport-Level Security: Main Steps

Transport-level security refers to securing the connection between a client application and a Web Service with Secure Sockets Layer (SSL).

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

To configure transport-level Web Services security:

1. Configure SSL for the core WebLogic Server security subsystem.

   You can configure one-way SSL where WebLogic Server is required to present a certificate to the client application, or two-way SSL where both the client applications and WebLogic server present certificates to each other.

   To configure two-way or one-way SSL for the core WebLogic Server security subsystem, see Configuring SSL.

2. In the JWS file that implements your Web Service, add the `@weblogic.jws.security.UserDataConstraint` annotation to require that the Web Service be invoked using the HTTPS transport.

   For details, see weblogic.jws.security.UserDataConstraint.

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

   See Iterative Development of WebLogic Web Services.

4. Update the `build.xml` file that invokes the `clientgen` Ant task to use a static WSDL to generate the JAX-RPC stubs of the Web Service, rather than the dynamic deployed WSDL of the service.

   The reason `clientgen` cannot generate the stubs from the dynamic WSDL in this case is that when you specify the `@UserDataConstraint` annotation, all client applications are required to specify a truststore, including `clientgen`. However, there is currently no way for `clientgen` to specify a truststore, thus the Ant task must generate its client components from a static WSDL that describes the Web Service in the same way as the dynamic WSDL.

5. When you run the client application that invokes the Web Service, specify certain properties to indicate the SSL implementation that your application should use. In particular:

   – To specify the Certicom SSL implementation, use the following properties

     ```
     -Djava.protocol.handler.pkgs=weblogic.net
     -Dweblogic.security.SSL.trustedCAKeyStore=trustStore
     ```

     where *`trustStore`* specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server's certificate).To disable host name verification, also specify the following property:

     ```
     -Dweblogic.security.SSL.ignoreHostnameVerification=true
     ```

   – To specify Sun's SSL implementation, use the following properties:

     ```
     -Djavax.net.ssl.trustStore=trustStore
     ```

     where *`trustStore`* specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server's certificate). To disable host name verification, also specify the following property:

     ```
     -Dweblogic.wsee.client.ssl.stricthostchecking=false
     ```

   See "Configuring Two-Way SSL for a Client Application" on page 4-3 for details about two-way SSL.

Configuring Two-Way SSL for a Client Application

If you configured two-way SSL for WebLogic Server, the client application must present a certificate to WebLogic Server, in addition to WebLogic Server presenting a certificate to the client application as required by one-way SSL. You must also follow these requirements:

- Create a client-side keystore that contains the client's private key and X.509 certificate pair.

  The SSL package of J2SE requires that the password of the client's private key must be the same as the password of the client's keystore. For this reason, the client keystore can include only *one* private key and X.509 certificate pair.

- Configure the core WebLogic Server's security subsystem, mapping the client's X.509 certificate in the client keystore to a user. See Configuring a User Name Mapper.

- Create a *truststore* which contains the certificates that the client trusts; the client application uses this truststore to validate the certificate it receives from WebLogic Server. Because of the J2SE password requirement described in the preceding bullet item, this truststore must be different from the keystore that contains the key pair that the client presents to the server.

  You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

  See *Obtaining Private Keys and Digital Signatures*.

- 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`*

Additional Web Services SSL Examples

The dev2dev CodeShare is a community of developers that share ideas, code and best practices related to BEA technologies. The site includes code examples for a variety of BEA technologies, including using SSL with Web Services.

To view and download the SSL Web Services code examples on the dev2dev site, go to the main Projects page and click on **Web Services** in the *By Technology* column.

CHAPTER 5

Configuring Access Control Security

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

Configuring Access Control Security: Main Steps

Access control security refers to configuring the Web Service to control the users who are allowed to access it, and then coding your client application to authenticate itself, using HTTP/S or username tokens, to the Web Service when the client invokes one of its operations.

You specify access control security for your Web Service by using one or more of the following annotations in your JWS file:

- `weblogic.jws.security.RolesAllowed`
- `weblogic.jws.security.SecurityRole`
- `weblogic.jws.security.RolesReferenced`
- `weblogic.jws.security.SecurityRoleRef`
- `weblogic.jws.security.RunAs`

Note: The `@weblogic.security.jws.SecurityRoles` and `@weblogic.security.jws.SecurityIdentity` JWS annotations are deprecated as of WebLogic Server 9.1.

The following procedure describes the high-level steps to use these annotations to enable access control security; later sections in the chapter describe the steps in more detail.

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

- Programming the JWS File
- Iterative Development of WebLogic Web Services
- Invoking Web Services

1. Update your JWS file, adding the `@weblogic.jws.security.RolesAllowed`, `@weblogic.jws.security.SecurityRole`, `@weblogic.jws.security.RolesReferenced`, or `@weblogic.jws.security.SecurityRoleRef` annotations as needed at the appropriate level (class or operation).

   See "Updating the JWS File With the Security-Related Annotations" on page 5-4.

2. Optionally specify that WebLogic Server internally run the Web Service using a specific role, rather than the role assigned to the user who actually invokes the Web Service, by adding the `@weblogic.jws.security.RunAs` JWS annotation.

   See "Updating the JWS File With the @RunAs Annotation" on page 5-6.

3. Optionally specify that your Web Service can be, or is required to be, invoked using HTTPS by adding the `@weblogic.jws.security.UserDataConstraint` JWS annotation.

   See "Configuring Access Control Security: Main Steps" on page 5-1 for details. This section also discusses how to update your client application to use SSL.

4. Recompile and redeploy your Web Service as part of the normal iterative development process.

   See Iterative Development of WebLogic Web Services.

5. Using the Administration Console, create valid WebLogic Server users, if they do not already exist. If the mapping of users to roles is external, also use the Administration Console to create the roles specified by the `@SecurityRole` annotation and map the users to the roles.

   Note: The mapping of users to roles is defined externally if you do *not* specify the `mapToPrincipals` attribute of the `@SecurityRole` annotation in your JWS file to list all users who can invoke the Web Service.

   See Users, Groups, and Security Roles.

6. Update your client application to use the `HttpTransportInfo` WebLogic API to specify the appropriate user and password when creating the JAX-RPC `Service` object.

   See "Setting the Username and Password When Creating the JAX-RPC Service Object" on page 5-7.

7. Update the `clientgen` Ant task in your `build.xml` file to specify the username and password of a valid WebLogic user (in the case where your Web Service uses the `@RolesAllowed` annotation) and the trust store that contains the list of trusted certificates, including WebLogic Server's (in the case you specify `@UserDataConstraint`).

   You do this by adding the standard Ant `<sysproperty>` nested element to the `clientgen` Ant task, and set the `key` attribute to the required Java property, as shown in the following example:

   ```
   <clientgen
       wsdl="http://example.com/myapp/myservice.wsdl"
       destDir="/output/clientclasses"
       packageName="myapp.myservice.client"
       serviceName="StockQuoteService"
       <sysproperty key="javax.net.ssl.trustStore"
                    value="/keystores/DemoTrust.jks"/>
       <sysproperty key="weblogic.wsee.client.ssl.stricthostchecking"
                    value="false"/>
       <sysproperty key="javax.xml.rpc.security.auth.username"
                    value="juliet"/>
       <sysproperty key="javax.xml.rpc.security.auth.password"
                    value="secret"/>
   </clientgen>
   ```

8. Regenerate client-side components and recompile client Java code as usual.

Updating the JWS File With the Security-Related Annotations

Use the WebLogic-specific `@weblogic.jws.security.RolesAllowed` annotation in your JWS file to specify an array of `@weblogic.jws.security.SecurityRoles` annotations that list the roles that are allowed to invoke the Web Service. You can specify these two annotations at either the class- or method-level. When set at the class-level, the roles apply to all public operations. You can add additional roles to a particular operation by specifying the annotation at the method level.

The `@SecurityRole` annotation has the following two attributes:

- `role`—Name of the role that is allowed to invoke the Web Service.
- `mapToPrincipals`—List of users that map to the role. If you specify one or more users with this attribute, you do not have to externally create the mapping between users and roles, typically using the Administration Console. However, the mapping specified with this attribute applies only within the context of the Web Service.

The `@RolesAllowed` annotation does not have any attributes.

You can also use the `@weblogic.jws.security.RolesReferenced` annotation to specify an array of `@weblogic.jws.security.SecurityRoleRef` annotations that list references to existing roles. For example, if the role `manager` is already allowed to invoke the Web Service, you can specify that the `mgr` role be linked to the `manager` role and any user mapped to `mgr` is also able to invoke the Web Service. You can specify these two annotations only at the class-level.

The `@SecurityRoleRef` annotation has the following two attributes:

- `role`—Name of the role reference.
- `link`—Name of the already-specified role that is allowed to invoke the Web Service. The value of this attribute corresponds to the value of the `role` attribute of a `@SecurityRole annotation` specified in the same JWS file.

The `@RolesReferenced` annotation does not have any attributes.

The following example shows how to use the annotations described in this section in a JWS file, with the relevant sections shown in bold:

```
package examples.webservices.security_roles;
```

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

// WebLogic JWS annotations

import weblogic.jws.WLHttpTransport;

import weblogic.jws.security.RolesAllowed;
import weblogic.jws.security.RolesReferenced;
import weblogic.jws.security.SecurityRole;
import weblogic.jws.security.SecurityRoleRef;

@WebService(name="SecurityRolesPortType",
            serviceName="SecurityRolesService",
            targetNamespace="http://example.org")

@WLHttpTransport(contextPath="security",
                 serviceUri="SecurityRolesService",
                  portName="SecurityRolesPort")

@RolesAllowed (  {
    @SecurityRole (role="manager",
                   mapToPrincipals={ "juliet","amanda" }),
    @SecurityRole (role="vp")
} )

@RolesReferenced (
    @SecurityRoleRef (role="mgr", link="manager")
)

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 */

public class SecurityRolesImpl {

  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";

  }
```

```
}
```

The example shows how to specify that only the `manager`, `vp`, and `mgr` roles are allowed to invoke the Web Service. The `mgr` role is actually a reference to the `manager` role. The users `juliet` and `amanda` are mapped to the `manager` role within the context of the Web Service. Because no users are mapped to the `vp` role, it is assumed that the mapping occurs externally, typically using the Administration Console to update the WebLogic Server security realm.

See JWS Annotation Reference for reference information on these annotations.

Updating the JWS File With the @RunAs Annotation

Use the WebLogic-specific `@weblogic.jws.security.RunAs` annotation in your JWS file to specify that the Web Service is always run as a particular role. This means that regardless of the user, and the role to which the user is mapped, initially invokes the Web Service, the service is internally executed as the specified role.

You can set the `@RunAs` annotation only at the class-level. The annotation has the following attributes:

- `role`—Role which the Web Service should run as.
- `mapToPrincipal`—Principal user that maps to the role.

The following example shows how to use the `@RunAs` annotation in a JWS file, with the relevant sections shown in bold:

```
package examples.webservices.security_roles;

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

// WebLogic JWS annotations

import weblogic.jws.WLHttpTransport;

import weblogic.jws.security.RunAs;

@WebService(name="SecurityRunAsPortType",
            serviceName="SecurityRunAsService",
            targetNamespace="http://example.org")

@WLHttpTransport(contextPath="security_runas",
                 serviceUri="SecurityRunAsService",
                 portName="SecurityRunAsPort")
```

```
@RunAs (role="manager", mapToPrincipal="juliet")
/**
 * This JWS file forms the basis of simple WebLogic
 * Web Service with a single operation: sayHello
 *
 */
public class SecurityRunAsImpl {

  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

Setting the Username and Password When Creating the JAX-RPC Service Object

When you use the `@RolesAllowed` JWS annotation to secure a Web Service, only the specified roles are allowed to invoke the Web Service operations. This means that you must specify the username and password of a user that maps to the role when creating the JAX-RPC `Service` object in your client application that invokes the protected Web Service.

WebLogic Server provides the `HttpTransportInfo` class for setting the username and password and passing it to the `Service` constructor. The following example is based on the standard way to invoke a Web Service from a standalone Java client (as described in Invoking Web Services) but also shows how to use the `HttpTransportInfo` class to set the username and password. The sections in bold are discussed after the example.

```
package examples.webservices.sec_wsdl.client;

import weblogic.wsee.connection.transport.http.HttpTransportInfo;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;

/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the SecWsdlService Web service.
```

```
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
public class Main {
  public static void main(String[] args)
      throws ServiceException, RemoteException{
    HttpTransportInfo info = new HttpTransportInfo();
    info.setUsername("juliet".getBytes());
    info.setPassword("secret".getBytes());
    SecWsdlService service = new SecWsdlService_Impl(args[0] + "?WSDL",
info);
    SecWsdlPortType port = service.getSecWsdlPort();
    try {
      String result = null;
      result = port.sayHello("Hi there!");
      System.out.println( "Got result: " + result );
    } catch (RemoteException e) {
      throw e;
    }
  }
}
```

The main points to note in the preceding example are as follows:

- Import the `HttpTransportInfo` class into your client application:

  ```
  import weblogic.wsee.connection.transport.http.HttpTransportInfo;
  ```

- Use the `setXXX()` methods of the `HttpTransportInfo` class to set the username and password:

  ```
  HttpTransportInfo info = new HttpTransportInfo();
  info.setUsername("juliet".getBytes());
  info.setPassword("secret".getBytes());
  ```

  In the example, it is assumed that the user `juliet` with password `secret` is a valid WebLogic Server user and has been mapped to the role specified in the `@RolesAllowed` JWS annotation of the Web Service.

  If you are accessing a Web Service using a proxy, the Java code would be similar to:

```
HttpTransportInfo info = new HttpTransportInfo();
Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
Integer.parseInt(proxyPort)));
info.setProxy(p);
info.setProxyUsername(user.getBytes());
info.setProxyPassword(pass.getBytes());
```

- Pass the `info` object that contains the username and password to the `Service` constructor as the second argument, in addition to the standard WSDL first argument:

  ```
  SecWsdlService service = new SecWsdlService_Impl(args[0] + "?WSDL",
  info);
  ```

See Invoking Web Services for general information about invoking a non-secured Web Service.