BEA WebLogic Server™

Programming WebLogic Security

Release 8.1
Document Date: December 9, 2002
Revised: NA

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic Security

Part Number	Date	Software Version
N/A	December 9, 2002	BEA WebLogic Server Version 8.1

Contents

About This Document

Audience for this Guide vii
e-docs Web Site viii
How to Print the Document viii
Related Information ix
Contact Us! x
Documentation Conventions x

1. Introduction

Audience for this Guide 1-1
Application Developers 1-2
Security Vendors Or Sophisticated Application Developers 1-3
Administrators 1-3
Why Implement Security? 1-4
Security APIs 1-5
Major Tasks Covered in this document 1-10

2. Securing Web Applications (Thin Clients)

Authentication with Web Browsers 2-1
Username and Password Authentication 2-2
Digital Certificate Authentication 2-4
Developing Secure Web Applications 2-6
Developing BASIC Authentication Web Applications 2-6
Developing FORM Authentication Web Applications 2-11
Developing CLIENT-CERT Authentication Web Applications 2-15
Deploying Web Applications 2-15
Using the <global-role/> Tag With Web Applications 2-17

Adding Declarative Security to Web Applications 2-19
Adding Programmatic Security to Web Applications 2-20
Programmatic Authentication 2-22

3. Writing Secure Java Clients (Fat Clients)

Introduction 3-1
Use of JSSE with WebLogic Server 3-2
JAAS Authentication 3-3
Supported JAAS Classes 3-3
Overview of Specific JAAS Programming Steps 3-4
Step 1: Authenticate the User 3-4
Step 2: Retrieve Subject and Associate it with the Client Actions 3-4
Step 3: Implement the CallbackHandler Interface 3-4
Writing a Client Application Using JAAS Authentication 3-5
Sample LoginModule Implementation 3-7
Sample Implementation of the CallbackHandler Interface 3-13
Sample LoginModule Configuration File 3-17
Sample LoginContext Implementation 3-17
Sample Login Method Implementation 3-18
Sample Implementation of the getSubject and runAS Methods 3-19
Sample PrivilegedAction Implementation 3-19
Sample Implementation of a Java Client That Uses JAAS 3-22
Using JNDI Authentication 3-25
Writing Applications that Use SSL 3-27
Communicating Securely with SSL-Enabled Web Browsers 3-28
Writing SSL Clients 3-29
SSL Client Sample 3-29
SSLSocketClient Sample 3-35
SSLClientServlet Sample 3-39
Using Two-Way SSL Authentication 3-40
Two-Way SSL Authentication with JNDI 3-41
Using Two-Way SSL Authentication Between WebLogic Server Instances 3-43
Using Two-Way SSL Authentication with Servlets 3-45
Using a Custom Host Name Verifier 3-46

Using a Trust Manager 3-48
Using an SSLContext 3-50
Using an SSLServerSocketFactory 3-50
Using URLs to Make Outbound SSL Connections 3-51

4. Securing EJB Applications

Adding Declarative Security to EJBs 4-1
Using the <global-role/> Tag With EJBs 4-3
Adding Programmatic Security to EJBs 4-5

5. Protecting Application Server Resources

Using Network Connection Filters to Protect Application Server Resources ... 5-1
Connection Filter Interfaces 5-2
ConnectionFilter interface 5-2
ConnectionFilterRulesListener interface 5-2
Connection Filter Classes 5-3
ConnectionFilterImpl Class 5-3
ConnectionEvent Class 5-3
Guidelines for Writing Connection Filter Rules 5-3
Connection Filter Rules Syntax 5-4
Types of Connection Filter Rules 5-5
How Connection Filter Rules are Evaluated 5-5
Configuring the Default Connection Filter 5-6
Developing Custom Connection Filters 5-7
Connection Filter Examples 5-8
SimpleConnectionFilter Example 5-8
SimpleConnectionFilter2 Example 5-9
Example of the Accept Method Used in Filtering Network Connections 5-9
Using J2EE Sandbox Security to Protect Application Server Resources 5-10

A. Deprecated Security APIs

Index

About This Document

This document is organized as follows:

- "Introduction" discusses the audiences of this document, the need for security, and the WebLogic Security application programming Interfaces (APIs).
- "Securing Web Applications (Thin Clients)" describes how to implement security in Web applications.
- "Writing Secure Java Clients (Fat Clients)" describes how to implement security in Java clients.
- "Securing EJB Applications" describes how to implement security in Enterprise JavaBeans.
- "Protecting Application Server Resources" discusses network connection filters and J2EE sandbox security.
- "Deprecated Security APIs" provides list of `weblogic.security` packages in which APIs have been deprecated.

Audience for this Guide

This document is intended for programmers who want to incorporate security into their WebLogic Server applications. It provides an overview of the WebLogic security APIs and describes how to use them. The procedures and code examples provided in this document assume that you are using the WebLogic security providers that are included in the WebLogic Server distribution, not Custom security providers. The usage of the

WebLogic security APIs does not change if you elect to use Custom security providers, however, the management procedures of the custom security providers may be different.

Note: This document does not provide comprehensive instructions on how to configure WebLogic Security providers and Custom security providers. For information on configuring WebLogic security providers and Custom security providers, see Managing WebLogic Security.

Note: This document is not intended for developers who want to write Custom security providers for use with WebLogic Server. It does not describe how to write Custom security providers. For information on developing Custom security providers, see Developing Security Providers for WebLogic Server.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File—Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

In addition to this document, Programming WebLogic Security, the following documents provide user information on WebLogic Security: These documents are available on the Web at `edocs.beasys.com`.

- Introduction to WebLogic Security—This document summarizes the features of the WebLogic Security Service and presents an overview of the architecture and capabilities of the WebLogic Security Service. It is the starting point for understanding the WebLogic Security Service.
- Managing WebLogic Security—This document explain how to configure security for WebLogic Server and how to use compatibility security.
- Developing Security Providers for WebLogic Server—This document provides security vendors and application developers with the information needed to develop custom security providers that can be use with WebLogic Server.
- Securing a WebLogic Server Deployment —This document explains how to use the security features of WebLogic Server to protect a WebLogic Server deployment.
- Upgrading Security in WebLogic Server Version 6.x to Version 8.1—This document provides procedures and other information you need to upgrade earlier versions of WebLogic Server to WebLogic Server 8.1. It also provides information about moving applications from an earlier version of WebLogic Server to 8.1.
- Security FAQ—This document introduces WebLogic Server Frequently Asked Questions.
- Security Javadocs—This document provides reference documentation for the WebLogic security packages that are provided with and supported by this release of WebLogic Server.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and filenames and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Introduction

The following topics are discussed in this section:

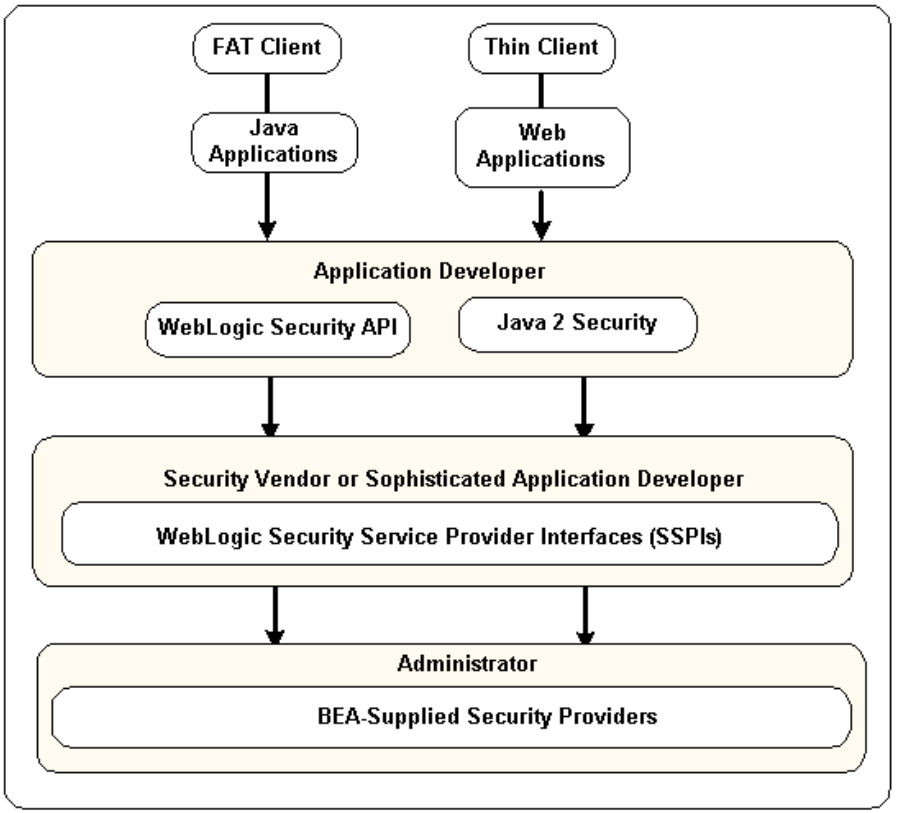
- Audience for this Guide
- Why Implement Security?
- Security APIs
- Major Tasks Covered in this document

Audience for this Guide

This document is intended for application developers who want to incorporate security into their WebLogic Server applications. It provides an overview of the WebLogic security APIs and describes how to use them.

Figure 1-1 illustrates how different types of users would interact with the software architecture of the BEA WebLogic Server security service. The new security architecture has benefits for three categories of users: application developers, third-party security service vendors, and administrators.

Figure 1-1 How Different Users use WebLogic Server Security



Application Developers

Application developers use the `WebLogic.Security` and Java 2 security application programming interfaces (APIs) to secure their applications. Therefore, this document provides instructions for using those APIs for securing Web applications, Java applications, and Enterprise JavaBeans (EJBs).

Security Vendors Or Sophisticated Application Developers

Security vendors use the Security Service Provider Interfaces (SSPIs) to develop custom security providers for use with WebLogic Server. Sophisticated application developers may be tasked with performing this function as well, however, this document does not address this task. For information on how to use the SSPIs to develop custom security providers, see *Developing Security Providers for WebLogic Server*.

Administrators

While administrators typically use the Administration Console to deploy, configure, and manage applications when they put the applications into production, application developers may also use the Administration Console to test their applications before they are put into production. At a minimum, testing requires that applications be deployed and configured. While this document does mention some aspects of administration as it relates to security, it references *Managing WebLogic Security*, *WebLogic Server Administration Guide*, and other documents for descriptions of how to use the Administration Console to perform security tasks.

The procedures and code examples provided in this document assume that you are using the WebLogic security providers that are included in the WebLogic Server distribution, not custom security providers. The usage of the Weblogic security APIs does not change if you elect to use custom security providers, however, the management procedures of the custom security providers may be different.

Note: This document does not provide comprehensive instructions on how to configure WebLogic Security providers or custom security providers. For information on configuring WebLogic Security providers and custom security providers, see *Managing WebLogic Security*.

Why Implement Security?

Security refers to techniques for ensuring that data stored in a computer or passed between computers is not compromised. Most security measures involve proof material and data encryption. Proof material is typically a secret word or phrase that gives a user access to a particular application or system. Data encryption is the translation of data into a form that cannot be interpreted.

Distributed applications, such as those used for electronic commerce (e-commerce), offer many access points at which malicious people can intercept data, disrupt operations, or generate fraudulent input. As a business becomes more distributed the probability of security breaches increases. Accordingly, as a business distributes its applications, it becomes increasingly important for the distributed computing software upon which such applications are built to provide security.

An application server resides in the sensitive layer between end users and your valuable data and resources. WebLogic Server provides authentication, authorization, and encryption services with which you can guard your resources. These services cannot provide protection, however, from an intruder who gains access by discovering and exploiting a weakness in your deployment environment.

Therefore, whether you deploy WebLogic Server on the Internet or on an intranet, it is a good idea to hire an independent security expert to go over your security plan and procedures, audit your installed systems, and recommend improvements.

Another good strategy is to read as much as possible about security issues. For the latest information about securing Web servers, BEA recommends reading the Security Improvement Modules, Security Practices, and Technical Implementations information available from the CERT™ Coordination Center operated by Carnegie Mellon University.

BEA suggests that you apply the remedies recommended in our security advisories. In the event of a problem with a BEA product, BEA distributes an advisory and instructions with the appropriate course of action. If you are responsible for security related issues at your site, please register to receive future notifications. BEA has established an e-mail address (`security-report@bea.com`) to which you can send reports of any possible security issues in BEA products. In addition, you are advised to apply every Service Pack as they are released. Service Packs include a roll up of all bug fixes for each version of the product, as well as each of the previously released Service Packs.

Partner products that can also help you in your effort to secure the WebLogic Server production environment. For more information, see the BEA Partner's Page.

Security APIs

This section lists the Security packages and classes that are implemented and supported by WebLogic Server. Application programmers use these packages to secure interactions between WebLogic Server and client applications, Enterprise JavaBeans (EJBs), and Web applications. Table 1-1 lists the supported Sun Java security packages. Table 1-2 lists the supported WebLogic security packages.

Note: Several of the WebLogic security packages, classes, and methods are being deprecated in this release of WebLogic Server. For more detailed information on deprecated packages and classes, see the particular package, class, or method in WebLogic javadoc.

Table 1-1 Supported Java Security Packages and Classes

Package or Class	Description
java.security.cert	Provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. It contains support for X.509 v3 certificates and X.509 v2 CRLs.
java.security.KeyStore	This class represents an in-memory collection of keys and certificates. It is used to manage two types of keystore entries: Key and Trusted Certificate.
java.security.PrivateKey	A private key. Private keys are used by entities for self-authentication.

Table 1-1 Supported Java Security Packages and Classes (Continued)

Package or Class	Description
java.security.Provider	This class represents a "Cryptographic Service Provider" for the Java Security API, where a provider implements some or all parts of Java Security. To supply implementations of cryptographic services, a team of developers writes the implementation code and creates a subclass of the Provider class.
javax.crypto	Provides the classes and interfaces for cryptographic operations. The cryptographic operations defined in this package include encryption, key generation and key agreement, and Message Authentication Code (MAC) generation. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. This package also supports secure streams and sealed objects. Many classes provided in this package are provider-based (see the `java.security.Provider` class). The class itself defines a programming interface to which applications may write. The implementations themselves may then be written by independent third-party vendors and plugged in seamlessly as needed. Therefore, application developers may take advantage of any number of provider-based implementations without having to add or rewrite code.
javax.naming	Provides the classes and interfaces for accessing naming services.
javax.net	Provides classes for networking applications. These classes include factories for creating sockets. Using socket factories you can encapsulate socket creation and configuration behavior.

Table 1-1 Supported Java Security Packages and Classes (Continued)

Package or Class	Description
javax.net.SSL	The classes and interfaces in this package are supported by WebLogic Server, but BEA recommends that you use the `weblogic.security.SSL` package when you use SSL with WebLogic Server.
javax.security.cert	Provides classes for public key certificates.
javax.servlet	Contains classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.
javax.servlet.http	Contains classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container.
javax.security.auth	Provides classes for performing JAAS-style LoginContext and Subject based authentication. This package provides a framework for authentication. The framework allows authentication to be performed in pluggable fashion. Different authentication modules can be plugged in for use with an application without requiring modifications to the application itself. **Note:** WebLogic Server provides full container support for JAAS authentication and supports full use of JAAS authentication and authorization in application code.

Table 1-2 Supported WebLogic Security Packages

Package	Description
weblogic.security	Several classes deprecated in this release of WebLogic Server. Provides interfaces and classes for mapping digital certificates sent from Web browsers and Java clients to WebLogic Server. This class makes it unnecessary for a user with a valid digital certificate to enter a username and password when accessing resources in WebLogic Server. **Note:** Several classes in this package are deprecated in this release of WebLogic Server; instead, the WebLogic security infrastructure and default WebLogic security providers are used to enforce security in WebLogic Server.
weblogic.security.acl	Provides interfaces and classes for creating custom security realms to access WebLogic Server users, groups, or ACLs from an external store. In addition, this package is used to test custom ACLs in server-side programs. **Note:** The use of ACLs is deprecated in this release of WebLogic Server; instead, security policies and a default WebLogic security Authorization provider are used to control access to server resources in WebLogic Server.

Table 1-2 Supported WebLogic Security Packages (Continued)

Package	Description
weblogic.security.audit	Provides interfaces and classes for auditing security events. WebLogic Server calls the Audit class with information about authentication and authorization requests. The package can be used to filter the authorization and authentication requests and direct them to a log file or other administrative facility. **Note:** This package is being deprecated in this release of WebLogic Server. It is replaced by the new security infrastructure which includes a WebLogic security Auditing provider.
weblogic.security.auth	Provides an authorization programming interface. **Note:** This package is being deprecated in this release of WebLogic Server. It is replaced by the new security infrastructure which includes a WebLogic security Authorization provider.
weblogic.security.net	Provides interfaces and classes that are used to examine connections to WebLogic Server and allow or deny the connections based on attributes such as the IP address, domain, or protocol of the initiator of the network connection.
weblogic.net.http. HTTPsURLConnection	Provides a class to represent an HTTPS connection to a remote object. This class is used to make an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server.

Table 1-2 Supported WebLogic Security Packages (Continued)

Package	Description
weblogic.security.SSL	Provides classes for the secure socket package. Using these classes, you can communicate using SSL to reliably detect any errors introduced into the network byte stream and to optionally encrypt the data and/or authenticate the communicating peers. **Note:** Some interfaces and classes in this package are deprecated in this release of WebLogic Server; instead the WebLogic security infrastructure and the Administration Console are used to configure SSL in WebLogic Sever.
weblogic.security.service	Provides the ContextHandler and ContextElement classes for developing custom security providers.
weblogic.security.services	Provides server-side authentication class.

Major Tasks Covered in this document

This document covers the following tasks:

- Developing secure Web application clients. This section includes instructions for using WebLogic security to secure the different types of Web application clients as defined by the J2EE specification, specifically Web browsers that use BASIC and FORM authentication. Instructions for the following tasks are provided:
  - Developing Secure Web Applications
  - Using the <global-role/> Tag With Web Applications
  - Adding Declarative Security to Web Applications

- Adding Programmatic Security to Web Applications

- Developing secure Java clients. This section includes instructions for using WebLogic security to secure the different types of Java application clients as defined by the J2EE specification, specifically:
  - Writing a Client Application Using JAAS Authentication
  - Writing Applications that Use SSL
- Developing secure EJBs. This section includes instructions for using WebLogic security to secure EJBs. Instructions for the following tasks are provided:
  - Adding Declarative Security to EJBs
  - Using the <global-role/> Tag With EJBs
  - Adding Programmatic Security to EJBs
- Protecting Application Server Resources. This section includes instructions for protecting resources at the server level. Instructions for the following tasks are provided:
  - Using Network Connection Filters to Protect Application Server Resources
  - Using J2EE Sandbox Security to Protect Application Server Resources

CHAPTER

2 Securing Web Applications (Thin Clients)

The following topics are discussed in this section:

- Authentication with Web Browsers
- Developing Secure Web Applications
- Using the <global-role/> Tag With Web Applications
- Adding Declarative Security to Web Applications
- Adding Programmatic Security to Web Applications
- Programmatic Authentication

Authentication with Web Browsers

Web browsers can connect to WebLogic Server over either an HyperText Transfer Protocol (HTTP) port or an HTTP Secure (HTTPS) port. The benefits of using an HTTPS port versus an HTTP port is two-fold. With HTTPS connections:

- All communication on the network between the Web browser and the server is encrypted. None of the communication including the username and password flows in clear text.
- As a minimum authentication requirement, the server is required to present a digital certificate to the Web browser client to establish its trusted identity.

If the server is configured for two-way SSL authentication, both the server and client are required to present a digital certificate to each other to establish their trusted identity.

Username and Password Authentication

WebLogic Server performs username and password authentication when users use a Web browser to connect to the server via the HTTP port. In this scenario, the browser and server interact in the following manner to authenticate a user (see Figure 2-1):

1. A user invokes a resource in WebLogic Server by entering the URL for that resource in a Web browser. The URL contains the HTTP listen port and the HTTP schema, for example, http://myserver:7001.
2. The Web server in WebLogic Server receives the request. WebLogic Server provides its own Web server but also supports the use of Apache Server, Microsoft Internet Information Server, and Netscape Enterprise Server as Web servers.
3. The Web server checks whether the WebLogic Server resource is protected by an security policy. If the WebLogic Server resource is protected, the Web server uses the established HTTP connection to request a username and password from the user.
4. When the user's Web browser receives the request from WebLogic Server, it prompts the user for a username and password.
5. The Web browser sends the request to the server again, along with the username and password.
6. The Web server forwards the request to the Web server plug-in. WebLogic Server provides the following plug-ins for Web servers:
   - Apache-Weblogic Server plug-in

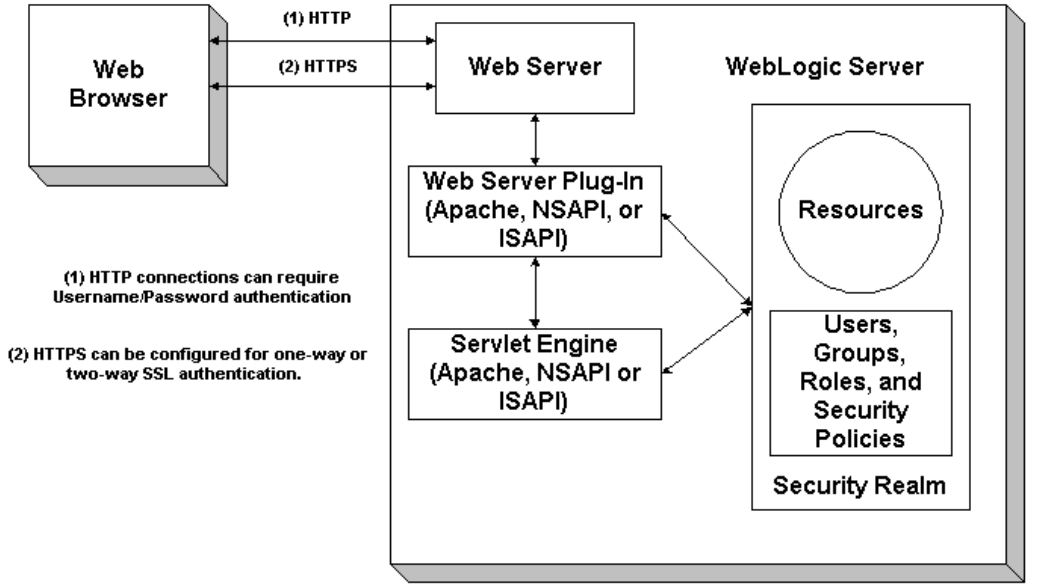
- Netscape Server Application Programming Interface (NSAPI)
- Internet Information Server Application Programming Interface (ISAPI)

The Web server plug-in performs authentication by sending the request, via the HTTP protocol, to the resource in WebLogic Server, along with the authentication data (username and password) received from the user.

7. Upon successful authentication, WebLogic Server determines whether the user has the permissions necessary to access the resource.
8. Before invoking a method on the server resource, the server performs an security authorization check. During this check, the server extracts the user's credentials from the security context, determines the user's role, compares the user's role to the security policy for the requested resource and verifies that the user is authorized to invoke the method on the resource.
9. If authorization succeeds, the server fulfills the request.

Figure 2-1 illustrates the secure login process for Web browsers.

Figure 2-1 Secure Login for Web Browsers



Digital Certificate Authentication

WebLogic Server uses encryption and digital certificate authentication when Web browser users connect to the server via the HTTPS port. In this scenario, the browser and server interact in the following manner to authenticate and authorize a user (see Figure 2-1):

1. A user invokes a resource in WebLogic Server by entering the URL for that resource in a Web browser. The URL contains the SSL listen port and the HTTPS schema, for example, https://myserver:7002.
2. The Web server in WebLogic Server receives the request. WebLogic Server provides its own Web server but also supports the use of Apache Server, Microsoft Internet Information Server, and Netscape Enterprise Server as Web servers.

3. The Web server checks whether the WebLogic Server resource is protected by an security policy. If the WebLogic Server resource is protected, the Web server uses the established HTTPs connection to request a username and password from the user.
4. When the user's Web browser receives the request from WebLogic Server, it prompts the user for a username and password. (This step is optional when using two-way authentication.)
5. The Web browser sends the request again, along with the username and password. (Only supplied if requested by the server.)
6. WebLogic Server presents its digital certificate to the Web browser.
7. The Web browser checks that the server's name matches the name in the digital certificate and that the digital certificate was issued by a trusted third party, that is, a trusted CA
8. If two-way authentication is in force on the server, the server requests a digital certificate from the client.
9. WebLogic Server checks that the client's name matches the name in the digital certificate and that the digital certificate was issued by a trusted third party, that is, a trusted CA.
10. The Web server forwards the request to the Web server plug-in. The Web server plug-in performs authentication by sending the request, via the HTTPS protocol, to the resource in WebLogic Server, along with the authentication data (username and password) received from the user.
11. Upon successful authentication, WebLogic Server determines whether the user has the permissions necessary to access the resource.
12. Before invoking a method on the server resource, the server performs an security authorization check. During this check, the server extracts the user's credentials from the security context, determines the user's role, compares the user's role to the security policy for the requested resource and verifies that the user is authorized to invoke the method on the resource.
13. If authorization succeeds, the server fulfills the request.

For more information, see the following sections:

- Managing WebLogic Security

- Configuring the Apache Server Plug-In
- Configuring the Microsoft-IIS Plug-In
- Configuring the Netscape Plug-In

Developing Secure Web Applications

WebLogic Server supports three types of authentication for Web browsers:

- BASIC
- FORM
- CLIENT-CERT

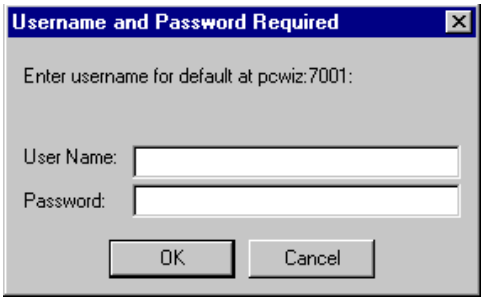
The following sections cover these topics:

- Developing BASIC Authentication Web Applications
- Developing FORM Authentication Web Applications
- Developing CLIENT-CERT Authentication Web Applications
- Deploying Web Applications

Developing BASIC Authentication Web Applications

With basic authentication, the Web browser pops up a login screen in response to a resource request. The login screen prompts the user for username and password. Figure 2-2 shows a typical login screen.

Figure 2-2 Basic Authentication Login Screen



To develop a Web application that provides basic authentication, perform these steps:

1. Create the `web.xml` deployment descriptor. In this file you include the following security information (see Listing 2-1):

   a. Define the welcome file. The welcome file name is Welcome.jsp.

   b. Define a security constraint for each set of Web application resources that you plan to protect. Each set of resources share a common URL. Resources such as HTML pages, JSPs, and servlets are the most commonly protected, but other types of resources are supported. In Listing 2-1, the URL pattern points to the `Welcome.jsp` file located in the Web application's top-level directory, the HTTP methods that are allowed to access the resource, POST and GET, and the role name, webuser.

   Note: Do not use hyphens in role names. Role names with hyphens cannot be modified in the Administration Console.

   c. Define the type of authentication you want to use and the security realm to which the security constraints will be applied. In this case, the BASIC type is specified and the realm is the default realm, which means that the security constraints will apply to the active security realm when the WebLogic Server boots.

   d. Define one or more security roles and map them to your security constraints. In our sample, only one role is defined, webuser, is defined in the security constraint so only one role name is defined here. However, any number of roles can be defined.

Listing 2-1 Basic Authentication web.xml File

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

    <web-app>
          <welcome-file-list>
               <welcome-file>welcome.jsp</welcome-file>
          </welcome-file-list>

          <security-constraint>
                <web-resource-collection>
                    <web-resource-name>Success</web-resource-name>
                      <url-pattern>/welcome.jsp</url-pattern>
                     <http-method>GET</http-method>
                     <http-method>POST</http-method>
                </web-resource-collection>
               <auth-constraint>
                    <role-name>webuser</role-name>
              </auth-constraint>
          </security-constraint>

          <login-config>
              <auth-method>BASIC</auth-method>
              <realm-name>default</realm-name>
          </login-config>

           <security-role>
               <role-name>webuser</role-name>
           </security-role>

           </web-app>
```

2. Create the `weblogic.xml` deployment descriptor. In this file you map security role names to users and groups. Listing 2-2 shows a sample `weblogic.xml` file that maps the webuser security role defined in the `<security-role>` tag in the `web.xml` file to myGroup. With this configuration, WebLogic Server will only allow users in myGroup to access the protected resource—`Welcome.jsp`. However, you can use the Administration Console to modify the Web application's security role so that other groups can be allowed to access the protected resource.

Listing 2-2 BASIC Authentication weblogic.xml File

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 6.0//EN"
"http://www.bea.com/servers/wls600/dtd/weblogic-web-jar.dtd">

<weblogic-web-app>

     <security-role-assignment>
         <role-name>webuser</role-name>
         <principal-name>myGroup</principal-name>
     </security-role-assignment>

</weblogic-web-app>
```

3. Create a file that produces the Welcome screen that displays when the user enters a username and password and is granted access. Listing 2-3 shows a sample `welcome.jsp` file. Figure 2-3 shows the Welcome screen.

Listing 2-3 BASIC Authentication welcome.jsp File

```
<html>
  <head>
   <title>Browser Based Authentication Example Welcome Page</title>
  </head>
  <h1> Browser Based Authentication Example Welcome Page </h1>

  <p> Welcome <%= request.getRemoteUser() %>!

  </blockquote>
  </body>
</html>
```

Figure 2-3 Welcome screen

4. Start WebLogic Server and define the users and Groups that will have access to the Web application resource. In the weblogic.xml file (see Listing 2-2), the `<principal-name>` tag defines `myGroup` as the group that has access to the Welcome.jsp. Therefore, use the Administration Console to define the myGroup group, define a user, and add that user to the myGroup group. For information on adding users and groups, see *Configuring WebLogic Security*.

5. Deploy the Web application and use the user defined in the previous step to access the protected resource.

   a. For deployment instructions, see "Deploying Web Applications" on page 2-15.

   b. Open a Web browser and enter this URL:

      ```
      http://localhost:7001/basicauth/welcome.jsp
      ```

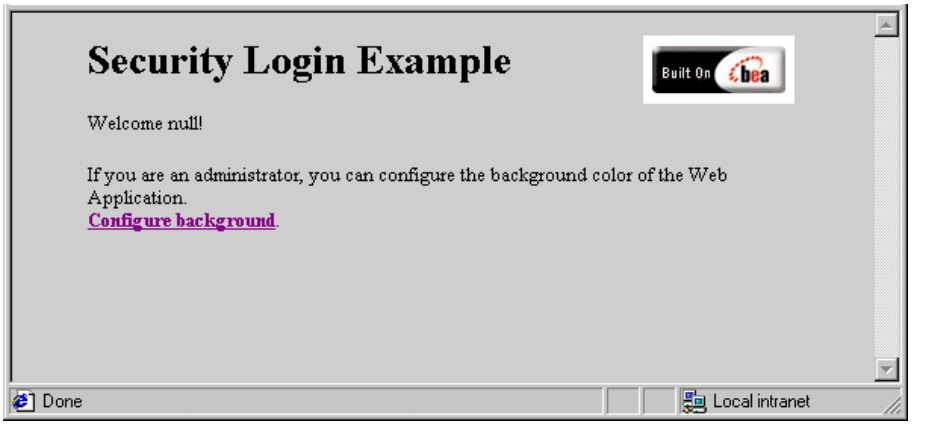
   c. Enter the username and password. The Welcome screen displays.

Developing FORM Authentication Web Applications

With FORM authentication, you provide a custom login screen that the Web browser displays in response to a resource request and error screen that displays if the login fails. The login screen prompts the user for username and password. Figure 2-2 shows a typical login screen. The benefit is that you have complete control over these screens so that you can design them to meet the requirements of your application.

Figure 2-4 shows the login screen for the form-based authentication sample application.

Figure 2-4 Form Authentication Login Screen



To develop a Web application that provides FORM authentication, perform these steps:

1. Create the `web.xml` deployment descriptor. In this file you include the following security information (see Listing 2-4):

   a. Define the welcome file. The welcome file name is Welcome.jsp.

   b. Define a security constraint for each set of Web application resources that you plan to protect. Each set of resources share a common URL. Resources such as HTML pages, JSPs, and servlets are the most commonly protected, but other types of resources are supported. In Listing 2-4, the URL pattern points `to /admin/edit.jsp` thus protecting the edit.jsp file located in the Web application's admin sub-directory, defines the HTTP method that is allowed to access the resource, `GET`, and defines the role name, `admin`.

Note: Do not use hyphens in role names. Role names with hyphens cannot be modified in the Administration Console.

c. Define the type of authentication you want to use and the security realm to which the security constraints will be applied. In this case, the FORM type is specified and no realm is specified so the realm is the default realm, which means that the security constraints will apply to the security realm that is activated when WebLogic Server boots.

d. Define one or more security roles and map them to your security constraints. In our sample, only one role is defined, admin, is defined in the security constraint so only one role name is defined here. However, any number of roles can be defined.

Listing 2-4 FORM Authentication web.xml File

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <welcome-file-list>
        <welcome-file>welcome.jsp</welcome-file>
    </welcome-file-list>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>AdminPages</web-resource-name>
            <description>
                These pages are only accessible by authorized
                 administrators.
            </description>
            <url-pattern>/admin/edit.jsp</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <description>
                These are the roles who have access.
            </description>
            <role-name>
                admin
            </role-name>
        </auth-constraint>
        <user-data-constraint>
            <description>
                This is how the user data must be transmitted.
```

```
            </description>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

    <login-config>
        <auth-method>FORM</auth-method>
        <form-login-config>
            <form-login-page>/login.jsp</form-login-page>
            <form-error-page>/fail_login.html</form-error-page>
        </form-login-config>
    </login-config>

    <security-role>
        <description>
            An administrator
        </description>
        <role-name>
            admin
        </role-name>
    </security-role>
</web-app>
```

2. Create the `weblogic.xml` deployment descriptor. In this file you map security role names to users and groups. Listing 2-5 shows a sample `weblogic.xml` file that maps the `admin` security role defined in the `<security-role>` tag in the `web.xml` file to the group `supportGroup`. With this configuration, WebLogic Server will only allow users in `support` group to access the protected resource. However, you can use the Administration Console to modify the Web application's security role so that other groups can be allowed to access the protected resource.

Listing 2-5 FORM Authentication weblogic.xml File

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 6.0//EN"
"http://www.bea.com/servers/wls600/dtd/weblogic-web-jar.dtd">

<weblogic-web-app>

     <security-role-assignment>
         <role-name>admin</role-name>
```

```
            <principal-name>supportGroup</principal-name>
        </security-role-assignment>

</weblogic-web-app>
```

3. Create a file that produces the Welcome screen when the user requests the protected resource by entering the URL. Listing 2-6 shows a sample `welcome.jsp` file. Figure 2-4 shows the Welcome screen.

Listing 2-6 Form Authentication welcome.jsp File

```
<html>
  <head>
    <title>Security login example</title>
  </head>

  <%
    String bgcolor;
   if ((bgcolor=(String)application.getAttribute("Background")) ==
        null)
    {
        bgcolor="#cccccc";
    }
  %>

  <body bgcolor=<%="\""+bgcolor+"\""%>>

  <blockquote>
  <img src=BEA_Button_Final_web.gif align=right>
  <h1> Security Login Example </h1>

  <p> Welcome <%= request.getRemoteUser() %>!

  <p> If you are an administrator, you can configure the background
  color of the Web Application.
  <br> <b><a href="admin/edit.jsp">Configure background</a></b>.

  <% if (request.getRemoteUser() != null) { %>
    <p> Click here to <a href="logout.jsp">logout</a>.
  <% } %>

  </blockquote>
  </body>
</html>
```

4. Start WebLogic Server and define the users and Groups that will have access to the Web application resource. In the weblogic.xml file (see Listing 2-5), the `<role-name>` tag defines `admin` as the group that has access to the edit.jsp file and defines the user joe as a member of that group. Therefore, use the Administration Console to define the `admin` group, and define user joe and add joe to the `admin` group. You can also define other users add them to the group and they will also have access to the protected resource. For information on adding users and groups, see *Configuring WebLogic Security*.
5. Deploy the Web application and use the user(s) defined in the previous step to access the protected resource.
   a. For deployment instructions, see "Deploying Web Applications" on page 2-15.
   b. Open a Web browser and enter this URL:

      `http://hostname:7001/security/welcome.jsp`

   c. Enter the username and password. The Welcome screen displays.

Developing CLIENT-CERT Authentication Web Applications

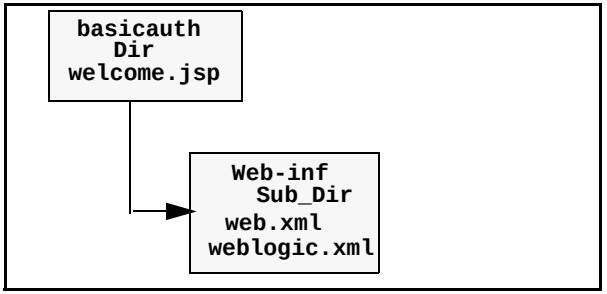
Use client certificates involves using the SSL and digital certificates to secure your network traffic and verify that clients are who they claim to be. For information on using SSL and digital certificates, see "Writing SSL Clients" on page 3-29.

Deploying Web Applications

To deploy a Web application on a server running in development mode, perform the following steps:

1. Set up a directory structure for the application's files. Figure 2-5 shows the directory structure for the Web application named basicauth. The top-level directory must be assigned the name of the Web application and the sub-directory must be named `Web-inf`.

Figure 2-5 Basicauth Application Directory Structure



2. To deploy the application in exploded directory format, simply move your directory to the applications directory on your server. For example, you would deploy the `basicauth` application in the following location:

   WL_HOME`\user_projects\mydomain\applications\basicauth`

   If the server is running, the application should auto-deploy. Use the Administration Console to verify that the application deployed.

   If the server is not running, the application should auto-deploy when you start the server

3. If you have not yet done so already, use the Administration Console to configure the users and groups that will have access to the application. To determine the users and groups that are allowed access to the protected resource, examine the `weblogic.xml` file. For example, the `weblogic.xml` file for the basicauth sample (see Listing 2-2) defines `myGroup` as the only group to have access to the `welcome.jsp` file.

For more information on deploying Web applications, see *Deployment Tools and Procedures*.

Using the <global-role/> Tag With Web Applications

With WebLogic Server versions 7.0 SP1 and later, there are four different options, or approaches, that you can use to configure security in Web applications:

- **Approach #1: Always use the deployment descriptors in the web.xml and weblogic.xml files.** In this case, you simply elect to never use the Administration Console to configure security and, instead, use the deployment descriptor files. The security configuration settings are read from the deployment descriptor files every time the Web application is deployed or redeployed. To use this approach, the Administration Console's Ignore Security Data in Deployment Descriptors attribute on the General tab of a security realm must be left in its default state—unchecked.
- **Approach #2: Always use the Administration Console.** In this case, before you deploy the application for the first time, you must check the Administration Console's Ignore Security Data in Deployment Descriptors attribute so that the deployment descriptors are ignored on initial deployment and subsequent redeployments. Then you use the Administration Console to configure Global Roles after the application is deployed.
- **Approach #3: Use the deployment descriptor files to configure security at initial application deployment and then use the Administration Console to configure security from that point forward.** In this case, the security configuration settings are read from the deployment descriptors upon initial deployment. Then you check the Ignore Security Data in Deployment Descriptors attribute so that deployment descriptors are ignored on subsequent redeployments and you use the Administration Console to make all subsequent changes to the application's security configuration.
- **Approach #4: Use the deployment descriptors to configure policies and use the Administration Console to configure roles.** In WebLogic Server 7.0 SP1 and later, support was added for the `<global-role/>` tag for use in the `weblogic.xml` and `weblogic-ejb-jar.xml` deployment descriptors. You can use this tag, instead of the `<principal-name>` tag, to explicitly indicate that you want the roles defined in the deployment descriptors by the `<role-name>` tag to use the mappings that you specify in the Administration Console.

Thus, the `<global-role/>` tag gives you the flexibility of not having to specify a specific role mapping for each role defined in the deployment descriptors for a particular Web application. Rather, you can use the Administration Console to specify and modify a specific role mapping for each defined role at anytime. Additionally, because you may elect to use this tag on some applications and not others, it is not necessary to check the Ignore Security Data In Deployment Descriptors attribute on the General tab of the security realm. Thus, within the same security realm, deployment descriptors can be used to specify and modify security for some applications while the Administration Console can be use to specify and modify security for others.

Listing 2-7 and Listing 2-8 show by comparison how to use the <global-role> tag.

Listing 2-7 Using the web.xml and weblogic.xml Files to Map Security Roles and Principals to a Security Realm

web.xml entries:

```
<web-app>
           ...
           <security-role>
               <role-name>webuser</role-name>
           </security-role>
           ...

</web-app>
```

<weblogic.xml entries:

```
<weblogic-web-app>

     <security-role-assignment>
         <role-name>webuser</role-name>
         <principal-name>myGroup</principal-name>
         <principal-name>Bill</principal-name>
         <principal-name>Mary</principal-name>
     </security-role-assignment>

</weblogic-web-app>
```

.

Listing 2-8 Using the <global-role> tag in Web Application Deployment Descriptors

web.xml entries:

```
<web-app>
           ...
           <security-role>
               <role-name>webuser</role-name>
           </security-role>
           ...

</web-app>
```

<weblogic.xml entries:

```
<weblogic-web-app>

     <security-role-assignment>
         <role-name>webuser</role-name>
         <global-role/>
     </security-role-assignment>
```

For information about how to use the Administration Console to configure security for EJBs, See *Managing WebLogic Security*.

Adding Declarative Security to Web Applications

To implement declarative security in Web application you use deployment descriptors (`web.xml` and `weblogic.xml`) to define security requirements. The deployment descriptors map the application's logical security requirements to its runtime definitions. And at runtime, the servlet container uses the security definitions to enforce the requirements. For a discussion of using deployment descriptors, see "Developing Secure Web Applications" on page 2-6.

For information about how to use deployment descriptors and the <global-role/> tag to configure security in Web applications declaratively, see “Using the <global-role/> Tag With Web Applications” on page 2-17.

For information about how to use the Administration Console to configure security in Web applications, See *Managing WebLogic Security*.

Adding Programmatic Security to Web Applications

You can write your servlets to access users and roles programmatically in your servlet code. To do this, use the following method in your servlet code: `javax.servlet.http.HttpServletRequest.isUserInRole(String role)`. This method returns a boolean indicating whether the authenticated user is included in the specified logical "role". If the user has not been authenticated, this method returns false.

This method maps roles to the group names in the security realm. Listing 2-9 shows the elements that are used with the `<servlet>` element to define the user role in the `web.xml` file.

Listing 2-9 IsUserInRole Web.xml and Weblogic.xml Elements

Begin web.xml entries:

```
...
<servlet>
        <security-role-ref>
             <role-name>user-rolename</role-name>
             <role-link>rolename-link</role-link>
         </security-role-ref>
</servlet>

<security-role>
         <role-name>rolename-link</role-name>
</security-role>
...
```

```
Begin weblogic.xml entries:

...
<security-role-assignment>
         <role-name>rolename-link</role-name>
         <principal-name>groupname</principal>
         <principal-name>username</principal>
</security-role-assignment>
...
```

The string `role` is mapped to the name supplied in the `<role-name>` element which is nested inside the `<security-role-ref>` element of a `<servlet>` declaration in the `web.xml` deployment description. The <role-name> element defines the name of the security role or principal that is used in the servlet code. The `<role-link>` element maps to a `<role-name>` defined in the `<security-role-assignment>` element in the `weblogic.xml` deployment descriptor.

For example, if the client has successfully logged in as user Bill with the role of manager, the following method would return true:

```
request.isUserInRole(“manager”)
```

The following listing provides an example.

Listing 2-10 Example of Security Role Mapping

```
Servlet code:
out.println("Is the user a Manager? " +
                        request.isUserInRole("manager"));

web.xml entries:

<servlet>
. . .
   <role-name>manager</role-name>
   <role-link>mgr</role-link>
. . .
</servlet>

<security-role>
   <role-name>mgr</role-name>
</security-role>

weblogic.xml entries:
```

```
<security-role-assignment>
   <role-name>mgr</role-name>
   <principal-name>bostonManagers</principal-name>
   <principal-name>Bill</principal-name>
   <principal-name>Ralph</principal-name>
</security-role-ref>
```

Programmatic Authentication

There are some applications where programmatic authentication is appropriate. A typical example is an application that supports user self-registration, that is, an application that requires an automated means for users to register an authentication identity for themselves and then be given immediate access to the site's protected resources. Usually, to self register, users are required to provide their identity and a password to protect the account, and perhaps some personal information that only they would know, for example, their mother's maiden name.

WebLogic Server provides a server-side API that supports programmatic authentication from within a servlet application:

```
weblogic.servlet.security.ServletAuthentication
```

Using this API, you can write servlet code that authenticates the user, logs in the user, and associates the user with the current session so that the user is registered in the active WebLogic Server security realm. Once the login is completed, it appears as if the user logged in using the standard mechanism. Listing 2-11 shows an example of how to use this API.

Listing 2-11 Programmatic Authentication Code Fragment

```
CallbackHandler handler = new SimpleCallbackHandler(username,
                                                    password);
Subject mySubject =
        weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject);
```

CHAPTER

3 Writing Secure Java Clients (Fat Clients)

The following topics are discussed in this section:

- Introduction
- Use of JSSE with WebLogic Server
- JAAS Authentication
- Writing a Client Application Using JAAS Authentication
- Using JNDI Authentication
- Writing Applications that Use SSL

Introduction

Whether the client is an application, applet, Enterprise JavaBean (EJB), or servlet that requires authentication, WebLogic Server uses the Java Authentication and Authorization Service (JAAS) classes to reliably and securely authenticate to the client. JAAS implements a Java version of the Pluggable Authentication Module (PAM) framework, which permits applications to remain independent from underlying authentication technologies. Therefore, the PAM framework allows the use of new or updated authentication technologies without requiring modifications to your application.

WebLogic Server uses JAAS for remote fat-client authentication, and internally for authentication. Therefore, only developers of custom Authentication providers and developers of remote fat client applications need to be involved with JAAS directly. Users of thin clients or developers of within-container fat client applications (for example, those calling an Enterprise JavaBean (EJB) from a servlet) do not require the direct use or knowledge of JAAS.

By default, WebLogic Server to uses SSL on the HTTPS port. The SSL protocol encrypts the data transmitted between the client and WebLogic Server so that the username and password do not flow in clear text.

Note: In order to implement security in a WebLogic client you must install the WebLogic Server software distribution kit on the Java client.

Use of JSSE with WebLogic Server

JSSE is a set of packages that support and implement the SSL and TLS v1, making those capabilities programmatically available. BEA WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between WebLogic Server clients and servers, Java clients, Web browsers, and other servers.

WebLogic's Java Secure Socket Extension (JSSE) implementation can be used by WebLogic clients, but is not required. Other JSSE implementations can be used for their client-side code outside the server as well.

The SSL implementation that WebLogic Server uses is static to the server configuration and is not replaceable by customer applications.

The WebLogic packages that implement JSSE are as follows:

- weblogic.security.SSL
- weblogic.net.http

JAAS Authentication

JAAS is a standard extension to the security in the Java Software Development Kit version 1.3. JAAS provides the ability to enforce access controls based on user identity. JAAS is provided in WebLogic Server as an alternative to the JNDI authentication mechanism.

WebLogic Server uses the authentication portion of the standard JAAS. The JAAS LoginContext provides support for the ordered execution of all configured authentication provider LoginModule instances and is responsible for the management of the completion status of each configured provider.

Note: JAAS is the preferred method of authentication, however, the WebLogic-supplied LoginModule only supports username and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client authentication, you must write a custom LoginModule that does certificate authentication.

Note: Before compiling your JAAS applications on WebLogic Server, add the following paths to your system's `CLASSPATH` to obtain required JAR files:

- `%WL_HOME%\server\lib\mbeantypes\wlManagement.jar`
- `%WL_HOME%\server\lib\mbeantypes\wlSecurityProviders.jar`

Note: WebLogic Server provides full container support for JAAS authentication and supports full use of JAAS authentication and authorization in application code.

Supported JAAS Classes

WebLogic Server supports the full JAAS 1.0 Reference Implementation with respect to authentication and authorization. While WebLogic Server does not protect any resources using JAAS authorization (it uses WebLogic security), you can use JAAS authorization in application code to protect the application's resources.

For more information about JAAS, see the *Java Authentication and Authorization Service Developer's Guide*.

Overview of Specific JAAS Programming Steps

The major steps for writing a secure application client using JAAS are as follows.

Step 1: Authenticate the User

The steps to authenticate the user are as follows:

1. Instantiate a LoginContext
2. Invoke the login by calling the LoginContext's login method.

These steps are executed using the JAAS classes and are described in Sun's *JAAS Authentication Tutorial* available at http://java.sun.com/j2se/1.4/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html.

Step 2: Retrieve Subject and Associate it with the Client Actions

If authentication is successful, the LoginModule populates the Subject with a Principal representing the user. The Principal the LoginModule places in the Subject is an instance of Principal, which is a class implementing the `java.security.Principal` interface. The calling application can subsequently retrieve the authenticated Subject by calling the LoginContext's `getSubject` method. The `weblogic.security.Security` class `runAs()` method is then used to associate the Subject identity with the `PrivilegedAction` or `PrivilegedExceptionAction` to be executed on behalf of the user identity.

Note: Use of the JAAS `javax.security.auth.Subject.doAs` methods in WebLogic Server applications do not associate the Subject with the client actions. You may use the `doAs` methods to implement J2SE security in WebLogic Server applications, but such usage is independent of the need to use the `Security.runAs()` method.

Step 3: Implement the CallbackHandler Interface

The LoginModule uses the CallbackHandler to communicate with the user and obtain the requested information, such as the username and password.

This step is executed using the JAAS CallbackHandler interface and is described in Sun's *JAAS Authentication Tutorial* available at `http://java.sun.com/j2se/1.4/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html`.

Writing a Client Application Using JAAS Authentication

To use JAAS in a Java client to authenticate a Subject, perform the following procedure:

1. Implement a LoginModule class for the authentication mechanism you want to use with WebLogic Server (see Listing 3-1). You need a LoginModule class for each type of authentication mechanism. You can have multiple LoginModule classes for a single WebLogic Server deployment.

   The `weblogic.security.auth.Authenticate` class uses a JNDI Environment object for initial context as described in Table 3-1.

2. Implement the CallbackHandler class that the LoginModule will use to communicate with the user and obtain the requested information, such as the username and password. See Listing 3-2 for the sample CallbackHandler used in the JAAS client sample provided in the WebLogic Server distribution.

3. Implement a configuration file that specifies which LoginModule classes should be used for your WebLogic Server and in which order the LoginModule classes should be invoked. See Listing 3-3 for the sample configuration file used in the JAAS client sample provided in the WebLogic Server distribution.

4. In the Java client, instantiate a `LoginContext`. The `LoginContext` consults the configuration file, `sample_jaas.config`, to load the default LoginModule configured for WebLogic Server. See Listing 3-4 for an example `LoginContext` instantiation.

Note: If you use another means to authenticate the user such as a perimeter authenticator or a remote WebLogic Server, the default LoginModule is determined by the remote resource.

5. Invoke the `login()` method of `LoginContext`. The `login()` method invokes all the loaded LoginModules. Each LoginModule attempts to authenticate the Subject. The `LoginContext` throws a `LoginException` if the configured login conditions are not met. See Listing 3-5 for an example of the `login()` method.

6. Retrieve the authenticated Subject from the LoginContext and call the action as the Subject. Upon successful authentication of a Subject, access controls can be placed upon that Subject by invoking the `runAs()` method of the `weblogic.security.Security` class. The `runAs()` method associates the specified Subject with the current thread's access permissions and then executes the action. See Listing 3-6 for an example implementation of the `getSubject()` and `runAs()` methods.

7. Write code to execute an action if the Subject has the required privileges. See Listing 3-7 for a sample implementation of the `javax.security.PrivilegedAction` class that executes an EJB to trade stocks.

8. Logout the user. The LoginModule logs out the user. See Listing 3-1 for a sample LoginModule.

Note: Before compiling your JAAS applications on WebLogic Server, add the following paths to your system's `CLASSPATH` to obtain required JAR files:

- `%WL_HOME%\server\lib\mbeantypes\wlManagement.jar`
- `%WL_HOME%\server\lib\mbeantypes\wlSecurityProviders.jar`

The `examples.security.jaas` example in the `SAMPLES_HOME\server\src\examples\security\jaas` directory provided with WebLogic Server shows how to implement JAAS authentication in a Java client.

The following sections show how each of these steps is implemented in the JAAS sample.

- Sample LoginModule Implementation
- Sample Implementation of the CallbackHandler Interface
- Sample LoginModule Configuration File
- Sample LoginContext Implementation
- Sample Login Method Implementation

- Sample Implementation of the getSubject and runAS Methods
- Sample PrivilegedAction Implementation

Sample LoginModule Implementation

Listing 3-1 shows the `UsernamePasswordLoginModule` that performs password authentication in the `examples.security.jaas` example. This login module class is provided in the WebLogic Server distribution in the `weblogic.jar` file located in the `WL_HOME\server\lib` directory.

Listing 3-1 Example of LoginModule for Password Authentication

```
package weblogic.security.auth.login;

import java.util.Map;
import java.util.Hashtable;
import java.io.IOException;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
import java.security.AccessController;
import java.security.Principal;
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.spi.LoginModule;
import weblogic.security.auth.Authenticate;
import weblogic.security.auth.callback.URLCallback;
import weblogic.jndi.Environment;

/**
 * <code>UsernamePasswordLoginModule</code> is used in a WLS client
 * to authenticate to the WLS server. This LoginModule
 * authenticates users with a username/password.
 * It can be used for both the T3 and IIOP clients.
 *
 * Callers of this module must implement callbacks to pass the
```

```
 * username, password, and optional URL for the WLS server. Caller
 * must implement a NameCallback for the username, a
 * PasswordCallback for the username, and a URLCallback for
 * the URL. If no URL is available, then the callback handler should
 * return null for the URLCallback.
 *
 * @author Copyright (c) 2002 by BEA. All Rights Reserved.
 */
public class UsernamePasswordLoginModule implements LoginModule
{
  private Subject subject = null;
  private CallbackHandler callbackHandler = null;
  private Map sharedState = null;
  private Map options = null;
  private boolean debug = true;
  private String url = null;

  // Authentication status
  private boolean succeeded = false;
  private boolean commitSucceeded = false;

  // Username and password
  private String username = null;
  private String password = null;

  /**
   * Initialize the login module.
   *
   * @param subject   Subject to contain principals returned from
   * WLS server.
   * @param callbackHandler CallbackHandler containing Name,
   * Password, and URL callbacks.
   * @param sharedState Map used to share state among different
   * login modules.
   *        This is not used by this login module.
   * @param options Map used to specify options to this login module.
   * Supported options
   *        debug and URL. The debug option can be used to display
   * additional debugging information. The URL option can be used
   * instead of the URL callback.
   */
  public void initialize(Subject subject,
                         CallbackHandler callbackHandler,
                         Map sharedState,
                         Map options)
  {
    this.subject = subject;
    this.callbackHandler = callbackHandler;
```

```
    this.sharedState = sharedState;
    this.options = options;

    // Check options for login module debug enabled
    if(options != null)
    {
      Object val = options.get("debug");
      if((val != null) && ((String) val).equalsIgnoreCase("true"))
      {
        debug = true;
     System.out.println("UsernamePasswordLoginModule.initialize(),
                         debug enabled");
      }
      val = options.get("URL");
      if(val != null)
      {
        url = (String) val;
        if(debug)
     System.out.println("UsernamePasswordLoginModule.initialize(),
                         URL " + url);
      }
    }
  }
  /**
   * Authenticate the user by username and password passed in
   *
   * @return true in all cases
   *
   * @exception FailedLoginException if the authentication fails.
   *
   * @exception LoginException if this LoginModule is unable to
   * perform the authentication.
   */
  public boolean login() throws LoginException
  {
    // Verify that the client supplied a callback handler
    if(callbackHandler == null)
    {
      if(debug)
       System.out.println("UsernamePasswordLoginModule.login(), no
callback handler specified");
      throw new LoginException("No CallbackHandler Specified");
    }
    // Populate callback list
    Callback[] callbacks = new Callback[3];
    callbacks[0] = new NameCallback("username: ");
    callbacks[1] = new PasswordCallback("password: ", false);
    callbacks[2] = new URLCallback("URL: ");
    try
```

```
{
  // Prompt for username and password and URL
  callbackHandler.handle(callbacks);

  // Retrieve username
  username = ((NameCallback) callbacks[0]).getName();
  if(debug)
    System.out.println("UsernamePasswordLoginModule.login(),
                          username " + username);

  // Retrieve password, converting from char[] to String
  char[] charPassword = ((PasswordCallback)
  callbacks[1]).getPassword();
  if(charPassword == null)
  {
    // Treat a NULL password as an empty password, not NULL
    charPassword = new char[0];
  }
  password = new String(charPassword);
  if(debug)
    System.out.println("UsernamePasswordLoginModule.login(),
                           password " + password);

  // Retrieve URL
  String callbackURL = ((URLCallback) callbacks[2]).getURL();
  if(callbackURL != null)
  {
    url = callbackURL;
  }
}
catch(IOException ioe)
{
  throw new LoginException(ioe.toString());
}
catch(UnsupportedCallbackException uce)
{
  throw new LoginException("Error: Callback " +
              uce.getCallback().toString() + " Not Available");
}
// Populate weblogic environment and authenticate
if (url != null) {
  Environment env = new Environment();
  env.setProviderUrl(url);
  env.setSecurityPrincipal(username);
  env.setSecurityCredentials(password);
  try
  {
    // Authenticate user credentials, populating Subject
    Authenticate.authenticate(env, subject);
```

```
    }
    catch(RemoteException re)
    {
      if(debug)
        System.out.println("Error: Remote Exception on
                           authenticate, " + re.getMessage());
      throw new LoginException(re.toString());
    }
    catch(IOException ioe)
    {
      if(debug)
        System.out.println("Error: IO Exception on authenticate,
                              " + ioe.getMessage());
      throw new LoginException(ioe.toString());
    }
    catch(LoginException le)
    {
      if(debug)
      System.out.println("Error: Login Exception on authenticate,
                             " + le.getMessage());
      throw new LoginException(le.toString());
    }
  }
  // Successfully authenticated subject with supplied info
  succeeded = true;
  return succeeded;
}
/**
 * This method is called if the LoginContext's overall
 * authentication succeeded (the relevant REQUIRED, REQUISITE,
 * SUFFICIENT and OPTIONAL LoginModules succeeded).
 *
* If this LoginModule's own authentication attempted failed, then
 * this method removes any state that was originally saved.
 *
 * @exception LoginException if the commit fails.
 *
 * @return true if this LoginModule's own login and commit
 *          attempts succeeded, or false otherwise.
 */
public boolean commit() throws LoginException
{
  if(succeeded)
  {
  // Just put the username and password into the private
  //credentials.
  final PasswordCredential passwordCred = new
                        PasswordCredential(username, password);
    AccessController.doPrivileged(new PrivilegedAction() {
```

```
            public java.lang.Object run() {
              subject.getPrivateCredentials().add(passwordCred);
              return null;
            }
          });

          url = null;
          commitSucceeded = true;
          return true;
        }
        else
        {
          username = null;
          password = null;
          url = null;
          return false;
        }
      }

      /**
       * This method is called if the LoginContext's
       * overall authentication failed.
       * If this LoginModule's own authentication attempt
       * succeeded (checked by retrieving the private state saved by the
       * login and commit methods), then this method cleans up any state
       * that was originally saved.
       *
       * @exception LoginException if the abort fails.
       *
       * @return false if this LoginModule's own login and/or commit
       * attempts failed, and true otherwise.

       */
      public boolean abort() throws LoginException
      {
        if(succeeded == false)
        {
          return false;
        }
        else if((succeeded == true) && (commitSucceeded == false))
        {
          // Login succeeded but overall authentication failed
          succeeded = false;
          username = null;
          password = null;
          url = null;
        }
        else
        {
```

```
      // Overall authentication succeeded and commit succeeded,
      // but a commit further down the authentication chain failed
      logout();
    }
    return true;
  }
/**
   * Logout the user.
   *
   * @exception LoginException if the logout fails.
   *
   * @return true in all cases since this LoginModule
   *          should not be ignored.
   */
  public boolean logout() throws LoginException
  {
    succeeded = false;
    commitSucceeded = false;
    username = null;
    password = null;
    url = null;
    return true;
  }
}
```

Note: WebLogic Server supports all callback types defined by JAAS as well as all callback types that extend the JAAS specification.

The UsernamePasswordLoginModule that is part of the WebLogic Server product checks for existing system user authentication definitions prior to execution and does nothing if they are already defined.

Sample Implementation of the CallbackHandler Interface

Listing 3-2 contains an example of a CallbackHandler interface used in JAAS authentication. The code in Listing 3-2 is taken from the `SampleCallbackHandler.java` file in the *`SAMPLES_HOME`*`\server\src\examples\security\jaas` directory.

Listing 3-2 Implementation of the CallbackHandler Interface

```
package examples.security.jaas;

import java.io.*;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.callback.TextOutputCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.TextInputCallback;
import javax.security.auth.callback.NameCallback;
import weblogic.security.auth.callback.URLCallback;
import examples.utils.common.ExampleUtils;

/**
 * SampleCallbackHandler.java
 * Implementation of the CallbackHandler Interface
 *
 * @author Copyright (c) 2000-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */
class SampleCallbackHandler implements CallbackHandler
{
  private String username = null;
  private String password = null;
  private String url = null;

  public SampleCallbackHandler() { }

  public SampleCallbackHandler(String pUsername, String pPassword,
                               String pUrl)
  {
    username = pUsername;
    password = pPassword;
    url = pUrl;
  }

  public void handle(Callback[] callbacks) throws IOException,
                                    UnsupportedCallbackException
  {
    for(int i = 0; i < callbacks.length; i++)
    {
      if(callbacks[i] instanceof TextOutputCallback)
      {
        // Display the message according to the specified type
        TextOutputCallback toc = (TextOutputCallback)callbacks[i];
        switch(toc.getMessageType())
        {
```

```
    case TextOutputCallback.INFORMATION:
      ExampleUtils.log(toc.getMessage());
      break;
    case TextOutputCallback.ERROR:
      ExampleUtils.log("ERROR: " + toc.getMessage());
      break;
    case TextOutputCallback.WARNING:
      ExampleUtils.log("WARNING: " + toc.getMessage());
      break;
    default:
      throw new IOException("Unsupported message type: " +
                              toc.getMessageType());
    }
  }
  else if(callbacks[i] instanceof NameCallback)
  {
    // If username not supplied on cmd line, prompt the user
    // for the username.
    NameCallback nc = (NameCallback)callbacks[i];
    if (ExampleUtils.isEmpty(username)) {
      System.err.print(nc.getPrompt());
      System.err.flush();
      nc.setName((new BufferedReader(new
                  InputStreamReader(System.in))).readLine());
    }
    else {
      ExampleUtils.log("username: "+username);
      nc.setName(username);
    }
  }
  else if(callbacks[i] instanceof URLCallback)
  {
    // If url not supplied on cmd line, prompt the user for the
    // url.
    // This example requires the url.
    URLCallback uc = (URLCallback)callbacks[i];
    if (ExampleUtils.isEmpty(url)) {
      System.err.print(uc.getPrompt());\
      System.err.flush();
      uc.setURL((new BufferedReader(new
                 InputStreamReader(System.in))).readLine());\
    }
    else {
      ExampleUtils.log("URL: "+url);
      uc.setURL(url);
    }
  }
  else if(callbacks[i] instanceof PasswordCallback)
```

```
        {
          PasswordCallback pc = (PasswordCallback)callbacks[i];

          // If password not supplied on cmd line, prompt the user
          // for the password.
          if (ExampleUtils.isEmpty(password)) {
            System.err.print(pc.getPrompt());
            System.err.flush();

            // Note: JAAS specifies that the password is a char[]
            // rather than a String.
            String tmpPassword = (new BufferedReader(new
                          InputStreamReader(System.in))).readLine();
            int passLen = tmpPassword.length();
            char[] passwordArray = new char[passLen];
            for(int passIdx = 0; passIdx < passLen; passIdx++)
              passwordArray[passIdx] = tmpPassword.charAt(passIdx);
            pc.setPassword(passwordArray);
          }
          else {
            String tPass = new String();
            for(int p = 0; p < password.length(); p++)
              tPass += "*";
            ExampleUtils.log("password: "+tPass);
            pc.setPassword(password.toCharArray());
          }
        }
        else if(callbacks[i] instanceof TextInputCallback)
        {
          // Prompt the user for the username
          TextInputCallback callback =
                                (TextInputCallback)callbacks[i];
          System.err.print(callback.getPrompt());
          System.err.flush();
          callback.setText((new BufferedReader(new
                        InputStreamReader(System.in))).readLine());
        }
        else
        {
          throw new UnsupportedCallbackException(callbacks[i],
                                           "Unrecognized Callback");
        }
      }
    }
  }
```

Sample LoginModule Configuration File

Listing 3-3 shows an implementation of the LoginModule configuration file that you use with a Java client to define the out-of-the-box UsernamePasswordLoginModule. This code is excerpted from the `sample_jaas.config` file located in the `SAMPLES_HOME\server\src\examples\security\jaas` directory.

Listing 3-3 Sample_jaas.config Code Example

```
/** Login Configuration for the JAAS Sample Application **/

Sample {
   weblogic.security.auth.login.UsernamePasswordLoginModule
         required debug=false;
};
```

Sample LoginContext Implementation

Listing 3-4 shows an implementation of the `LoginContext` class. This code is excerpted from the `JavaClient.java` located in the `SAMPLES_HOME\server\src\examples\security\jaas` directory.

Listing 3-4 LoginContext Code Fragment

```
...
import javax.security.auth.login.LoginContext;
...

    LoginContext loginContext = null;

    try
    {
     // Create LoginContext; specify username/password login module
       loginContext = new LoginContext("Sample",
              new SampleCallbackHandler(username, password, url));
    }
```

Sample Login Method Implementation

Listing 3-5 shows an implementation of the `login()` method. This code is excerpted from the `JavaClient.java` file located in the `SAMPLES_HOME\server\src\examples\security\jaas` directory.

Listing 3-5 Login Method Code Fragment

```
...
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.AccountExpiredException;
import javax.security.auth.login.CredentialExpiredException;
...
 /**
     * Attempt authentication
     */
    try
    {
     // If we return without an exception, authentication succeeded
      loginContext.login();

    }

    catch(FailedLoginException fle)

    {
      System.out.println("Authentication Failed, " +
                          fle.getMessage());
      System.exit(-1);
    }
    catch(AccountExpiredException aee)
    {
     System.out.println("Authentication Failed: Account Expired");
      System.exit(-1);
    }
    catch(CredentialExpiredException cee)
    {
      System.out.println("Authentication Failed: Credentials
                           Expired");
      System.exit(-1);
    }
    catch(Exception e)
    {
```

```
      System.out.println("Authentication Failed: Unexpected
                          Exception, " + e.getMessage());
      e.printStackTrace();
      System.exit(-1);
    }
```

Sample Implementation of the getSubject and runAS Methods

Listing 3-6 shows an implementation of the `getSubject()` and `runAs()` methods. This code is excerpted from the `JavaClient.java` file located in the *`SAMPLES_HOME`*`\server\src\examples\security\jaas` directory.

Listing 3-6 GetSubject and RunAs Methods Code Fragment

```
...
/**
 * Retrieve authenticated subject, perform SampleAction as Subject
 */
   Subject subject = loginContext.getSubject();
   SampleAction sampleAction = new SampleAction(url);
   Security.runAs(subject, sampleAction);
   System.exit(0);

...
```

Sample PrivilegedAction Implementation

Listing 3-7 contains an implementation of the `javax.security.PrivilegedAction` class.

If the user is valid, has provided the correct password, and has sufficient permissions to execute the EJB, then the SampleClient uses the `weblogic.security.Security.runAs()` method to call SampleAction to execute of the Trader EJB.

The code in Listing 3-7 is excerpted from the `SampleAction.java` file in the *SAMPLES_HOME*`\server\src\examples\security\jaas` directory.

Listing 3-7 Example of a PrivilegedAction Implementation

```
package examples.security.jaas;

import java.security.PrivilegedAction;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.FinderException;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import examples.ejb20.basic.statelessSession.TraderHome;
import examples.ejb20.basic.statelessSession.Trader;
import examples.utils.common.ExampleUtils;

/**
 * SampleAction.java
 *
 * JAAS sample PrivilegedAction Implementation
 *
 * @author Copyright (c) 2000-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */
public class SampleAction implements PrivilegedAction
{
  private static final String JNDI_NAME =
            "ejb20-statelessSession-TraderHome";
  private String url;

  public SampleAction(String url)
  {
    this.url = url;
  }

  public Object run()
  {
    Object obj = null;
```

```
    try {
      callTraderEJB();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    return obj;
  }

  /**
   * Call Trader EJB.
   */
  public void callTraderEJB()
    throws NamingException, CreateException, RemoteException,
               RemoveException
  {
    TraderHome home = lookupTraderHome();

    // create a Trader
    ExampleUtils.log("Creating a trader");
    Trader trader = (Trader)ExampleUtils.narrow(home.create(),
                      Trader.class);

    String [] stocks = {"BEAS", "MSFT", "AMZN", "HWP" };

      // execute some buys
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      ExampleUtils.log("Buying "+shares+" shares of
                        "+stocks[i]+".");
      trader.buy(stocks[i], shares);
    }

    // execute some sells
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      ExampleUtils.log("Selling "+shares+" shares of
                          "+stocks[i]+".");
      trader.sell(stocks[i], shares);
    }

    // remove the Trader
    ExampleUtils.log("Removing the trader");
    trader.remove();
  }

  /**
   * Look up the bean's home interface using JNDI.
   */
  private TraderHome lookupTraderHome()
```

```
    throws NamingException
  {
    Context ctx = ExampleUtils.getInitialContext(url);
    Object home = (TraderHome)ctx.lookup(JNDI_NAME);
   return (TraderHome)ExampleUtils.narrow(home, TraderHome.class);
  }
}
```

Sample Implementation of a Java Client That Uses JAAS

Listing 3-8 contains an example of a Java client that uses JAAS authentication. The code in Listing 3-8 is excerpted from the `SampleClient.java` file in the *SAMPLES_HOME*`\server\src\examples\security\jaas` directory.

Listing 3-8 Example of Java Client That Uses JAAS Authentication

```
package examples.security.jaas;

import java.util.*;
import javax.security.auth.Subject;
import javax.security.auth.callback.TextInputCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.AccountExpiredException;
import javax.security.auth.login.CredentialExpiredException;
import weblogic.security.acl.internal.AuthenticatedSubject;
import weblogic.security.Security;
import examples.utils.common.ExampleUtils;

/**
 * SampleClient.java
 * Sample client for JAAS user authentication.
 * Usage:   java examples.security.jaas.SampleClient [url]
 *             [username(optional)] [password(optional)]
 * Example: java examples.security.jaas.SampleClient
 *            t3://localhost:7001 weblogic weblogic
 *
 * @author Copyright (c) 2000-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */
```

```
public class SampleClient
{
  /**
  * Attempt to authenticate the user.
  */
  public static void main(String[] args)
  {
    String username = null;
    String password = null;
    String url = null;

    // Url is required
    if(args.length < 1) {
      System.out.println("Usage:
                  java examples.security.jaas.SampleClient [url]
                      [username(optional)] [password(optional)]");
      System.out.println("Example:
                  java examples.security.jaas.SampleClient
                       t3://localhost:7001");
      System.exit(0);
    }

    // Parse the argument list
    switch(args.length) {
      case 3:
        password = args[2];
      // fall through
      case 2:
        username = args[1];
      // fall through
      case 1:
        url      = args[0];
      break;
    }

    LoginContext loginContext = null;

    /**
     * Set JAAS server url system property and create a LoginContext
      */
    try
    {
     // Create LoginContext; specify username/password login module
      loginContext = new LoginContext("Sample",
             new SampleCallbackHandler(username, password, url));
    }
    catch(SecurityException se)
    {
      se.printStackTrace();
```

```
  System.exit(-1);
}
catch(LoginException le)
{
  le.printStackTrace();
  System.exit(-1);
}

/**
 * Attempt authentication
 */
try
{
 // If we return without an exception, authentication succeeded
  loginContext.login();

}

catch(FailedLoginException fle)

{
  System.out.println("Authentication Failed, " +
                          fle.getMessage());
  System.exit(-1);
}
catch(AccountExpiredException aee)
{
 System.out.println("Authentication Failed: Account Expired");
  System.exit(-1);
}
catch(CredentialExpiredException cee)
{
  System.out.println("Authentication Failed: Credentials
                           Expired");
  System.exit(-1);
}
catch(Exception e)
{
  System.out.println("Authentication Failed: Unexpected
                         Exception, " + e.getMessage());
  e.printStackTrace();
  System.exit(-1);
}

/**
 * Retrieve authenticated subject, perform SampleAction as
 * Subject
 */
Subject subject = loginContext.getSubject();
SampleAction sampleAction = new SampleAction(url);
```

```
    Security.runAs(subject, sampleAction);
    System.exit(0);
  }
}
```

For more information about using JAAS, see the *Java Authentication and Authorization Service Developer's Guide*.

Using JNDI Authentication

Java clients use the Java Naming and Directory Interface (JNDI) to pass credentials to WebLogic Server. A Java client establishes a connection with WebLogic Server by getting a JNDI InitialContext. The Java client then uses the InitialContext to look up the resources it needs in the WebLogic Server JNDI tree.

Note: JAAS is the preferred method of authentication, however, the WebLogic-supplied LoginModule only supports username and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client authentication, you must write a custom LoginModule that does certificate authentication.

To specify a user and the user's credentials set the JNDI properties listed in the following table.

Table 3-1 JNDI Properties Used for Authentication

Property	Meaning
`INITIAL_CONTEXT_FACTORY`	Provides an entry point into the WebLogic Server environment. The class `weblogic.jndi.WLInitialContextFactory` is the JNDI SPI for WebLogic Server.
`PROVIDER_URL`	Specifies the host and port of the WebLogic Server. For example: `t3://weblogic:7001`.

Table 3-1 JNDI Properties Used for Authentication (Continued)

Property	Meaning
SECURITY_AUTHENTICATION	Indicates the types of authentication to be used. The following values are valid: ■ None indicates that no authentication is performed. ■ Simple indicates that password authentication is performed. ■ Strong indicates that certificate authentication is performed. **Note:** If you try to access a secure component on WebLogic Server, user authentication will be required by WebLogic Server regardless of the type of authentication indicated by the SECURITY_AUTHENTICATION setting. For example, if you set SECURITY_AUTHENTICATION to None, you will still be required to supply the correct password to access a secure component.
SECURITY_PRINCIPAL	Specifies the identity of the User when that User authenticates to the WebLogic Server security realm.
SECURITY_CREDENTIALS	Specifies the credentials of the User when that User authenticates to the WebLogic Server security realm. ■ For password authentication enabled via SECURITY_AUTHENTICATION="simple", this property specifies a string that is either the User's password or a User object used by WebLogic Server to verify credentials. ■ For certificate authentication enabled via SECURITY_AUTHENTICATION="strong", this property specifies the name of the X509 object that contains the digital certificate and private key for the WebLogic Server.

These properties are stored in a hash table that is passed to the `InitialContext` constructor.

Listing 3-9 demonstrates how to use certificate authentication in a Java client. Notice the use of T3S, which is a WebLogic Server proprietary version of SSL. T3S uses encryption to protect the connection and communication between WebLogic Server and the Java client.

Listing 3-9 Example of Certificate Authentication

```
...
Hashtable env = new Hashtable();
       env.put(Context.INITIAL_CONTEXT_FACTORY,
              "weblogic.jndi.WLInitialContextFactory");
       env.put(WLContext.PROVIDER_URL, “t3s://weblogic:7001”);
       env.put(WLContext.SECURITY_AUTHENTICATION “strong”);
       env.put(Context.SECURITY_PRINCIPAL, “javaclient”);
       env.put(Context.SECURITY_CREDENTIALS, “certforclient”);

      ctx = new InitialContext(env);
```

The code in Listing 3-9 generates a call to the Identity Asserter class that implements the `weblogic.security.providers.authentication.UserNameMapper interface`. The class that implements the `UserNameMapper` interface returns a User object if the digital certificate is valid. WebLogic Server stores this authenticated User object on the Java client's thread in WebLogic Server and uses it for subsequent authorization requests when the thread attempts to use resources protected by the security realm.

Note: The implementation of the `weblogic.security.providers.authentication.UserNameMapper` interface must be specified in your CLASSPATH.

Writing Applications that Use SSL

This section covers the following topics:

- Communicating Securely with SSL-Enabled Web Browsers

- Writing SSL Clients
- Using Two-Way SSL Authentication
- Two-Way SSL Authentication with JNDI
- Using a Custom Host Name Verifier
- Using a Trust Manager
- Using an SSLContext
- Using an SSLServerSocketFactory
- Using URLs to Make Outbound SSL Connections

Communicating Securely with SSL-Enabled Web Browsers

You can use a URL object to make an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server. The `weblogic.net.http.HttpsURLConnection` class provides a way to specify the security context information for a client including the digital certificate and private key of the client.

The `weblogic.net.http.HttpsURLConnection` class provides methods for determining the negotiated cipher suite, getting/setting a HostName Verifier, getting the server's certificate chain, and getting/setting an SSLSocketFactory in order to create new SSL sockets.

The SSLClient code example demonstrates using the `weblogic.net.http.HttpsURLConnection` class to make an outbound SSL connection. In addition, the SSL client code example demonstrates using the Java Secure Socket Extension (JSSE) application programming interface (API) to make the outbound SSL connection. The SSLclient code example is available in the `examples.security.sslclient` package in the *`SAMPLES_HOME`*`\server\src\examples\security\sslclient` directory.

Writing SSL Clients

This section describes, by way of example, how to write various types of SSL clients. Examples of the following types of SSL clients are provided:

- SSL Client Sample
- SSLSocketClient Sample
- SSLClientServlet Sample

SSL Client Sample

The SSLClient sample demonstrates how to use the WebLogic SSL library to make outgoing SSL connections. It shows both how to do this from a stand-alone application as well as from within WebLogic Server, that is, in a servlet.

You can compile this sample using WebLogic's implementation or Sun's JSSE implementation. To use Sun's JSSE implementation, you must uncomment the following code in the `SSLClient.java` file and then recompile the sample:

- The JSSE Java package import statements.
- The code for the `jsseURLConnect()` method.
- The `NullHostnameVerifier` inner class.

Note: When making an outgoing SSL connection, WebLogic Server will use that instance of the server's certificate. When communicating to either the same or another WebLogic Server with two-way SSL, the originating server's certificate will be verified against the client root CA list in the receiving WebLogic Server.

Listing 3-10 shows a sample SSLClient. This code taken from the `SSLClient.java` file located at *`SAMPLES_HOME`*`\server\src\examples\security\sslclient`.

Listing 3-10 SSL Client Sample

```
package examples.security.sslclient;
```

```
import java.io.File;
import java.net.URL;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.Hashtable;
import java.security.Provider;
import javax.naming.NamingException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletOutputStream;
//import com.sun.net.ssl.internal.www.protocol.https.*;
//import com.sun.net.ssl.*;
import weblogic.net.http.*;
import weblogic.jndi.Environment;

/** SSLClient is a short example of how to use the SSL library of
 * WebLogic to make outgoing SSL connections. It shows both how
 * to do this from a stand-alone application as well as from within
 * WebLogic (in a Servlet).
 *
 * If you are using the JSSE implementation from Sun Microsystems, you must
 * uncomment three places in the SSLClient.java code and re-compile:
 *
 *     1. JSSE import statements at the top of this file.
 *     2. The code in method jsseURLConnect.
 *     3. The NullHostnameVerifier inner class.
 *
 *  Be careful to notice that the WebLogic Server, when making an
 *    outgoing SSL connection, will use that instance of the server's
 *    certificate. When communicating to either the same or another
 *    WebLogic Server with two-way SSL, the originating server's
 *    certificate will be verified against the client root CA list in
 *    the receiving WebLogic Server.
 *
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */

public class SSLClient {
  public void SSLClient() {}
  public static void main (String [] argv)
    throws IOException {
    if ((!((argv.length == 4) || (argv.length == 5))) ||
        (!((argv[0].equals("wls") || argv[0].equals("jsse"))))
        ) {
      System.out.println("usage:   java SSLClient jsse|wls host port
```

```
                                    sslport <query>");
    System.out.println("example: java SSLClient wls server2.weblogic.com 80 443
                          /examplesWebApp/SnoopServlet.jsp");
     System.exit(-1);
   }
   try {
     System.out.println("----");
     if (argv.length == 5) {
       if (argv[0].equals("wls"))
         wlsURLConnect(argv[1], argv[2], argv[3], argv[4],
                       System.out);
       else
         jsseURLConnect(argv[1], argv[2], argv[3], argv[4],
                        System.out);
     } else {  // for null query, default page returned...
       if (argv[0].equals("wls"))
         wlsURLConnect(argv[1], argv[2], argv[3], null, System.out);
       else
         jsseURLConnect(argv[1], argv[2], argv[3], null,
                        System.out);
     }
     System.out.println("----");
   } catch (Exception e) {
     e.printStackTrace();
     printSecurityProviders(System.out);
     System.out.println("----");
   }
 }
 private static void printOut(String outstr, OutputStream stream) {
   if (stream instanceof PrintStream) {
     ((PrintStream)stream).print(outstr);
     return;
   } else if (stream instanceof ServletOutputStream) {
     try {
       ((ServletOutputStream)stream).print(outstr);
       return;
     } catch (IOException ioe) {
       System.out.println(" IOException: "+ioe.getMessage());
     }
   }
   System.out.print(outstr);
 }
 private static void printSecurityProviders(OutputStream stream) {
   StringBuffer outstr = new StringBuffer();
   outstr.append(" JDK Protocol Handlers and Security
                    Providers:\n");
   outstr.append("   java.protocol.handler.pkgs - ");
   outstr.append(System.getProperties().getProperty(
                                     "java.protocol.handler.pkgs"));
```

```
  outstr.append("\n");
  Provider[] provs = java.security.Security.getProviders();
  for (int i=0; i<provs.length; i++)
      outstr.append("   provider[" + i + "] - " + provs[i].getName()
                    + " - " + provs[i].getInfo() + "\n");
  outstr.append("\n");
  printOut(outstr.toString(), stream);
}

private static void tryConnection(java.net.HttpURLConnection
                                  connection, OutputStream stream)
  throws IOException {
    connection.connect();
    String responseStr = "\t\t" +
                         connection.getResponseCode() + " -- " +
                         connection.getResponseMessage() + "\n\t\t" +
                         connection.getContent().getClass().getName()
                    + "\n";
    connection.disconnect();
    printOut(responseStr, stream);
}
/**
 * This method contains an example of how to use the URL and
 *  URLConnection objects to create a new SSL connection, using
 *  JSSE SSL client classes.
 *
 * This method is commented out for compilation without the JSSE
 * classes. To observe JSSE in action, see instructions in
 * package.html.
 */
public static void jsseURLConnect(String host, String port,
                                  String sport, String query,
                                  OutputStream out) {
  /*
  try {
    if (query == null)
      query = "/examplesWebApp/index.jsp";
    String handlers =
        System.getProperty("java.protocol.handler.pkgs");
    System.setProperty("java.protocol.handler.pkgs",
              "com.sun.net.ssl.internal.www.protocol|" + handlers);
    java.security.Security.addProvider(new
             com.sun.net.ssl.internal.ssl.Provider());
    printSecurityProviders(out);
    printOut(" Trying a new HTTP connection using JDK client
            classes - \n\thttp://" +
             host + ":" + port + query + "\n", out);
    URL url = null;
    try {
```

```
      url = new URL( "http", host,
                     Integer.valueOf(port).intValue(), query);
      java.net.HttpURLConnection connection =
        (java.net.HttpURLConnection) url.openConnection();
      tryConnection(connection, out);
    } catch (Exception e) {
      printOut(e.getMessage(), out);
      e.printStackTrace();
      printSecurityProviders(System.out);
      System.out.println("----");
    }
    printOut(" Trying a new HTTPS connection using JDK client
             classes - \n\thttps://" +
             host + ":" + sport + query +"\n", out);
    URL jsseUrl = new URL( "https", host,
                        Integer.valueOf(sport).intValue(), query,
      new com.sun.net.ssl.internal.www.protocol.https.Handler() );
    java.net.HttpURLConnection sconnection =
      (java.net.HttpURLConnection) jsseUrl.openConnection();

    /* Uncomment this and the inner class at the bottom of the
     * file to observe how the HostnameVerifier works with JSSE.
    *//*
    if (sconnection instanceof com.sun.net.ssl.HttpsURLConnection)
      ((com.sun.net.ssl.HttpsURLConnection)sconnection)
        .setHostnameVerifier(new NullHostnameVerifier());
    *//*
    printOut("\t using a "+sconnection.getClass().getName() +
             "\n", out);
    tryConnection(sconnection, out);
  } catch (IOException ioe) {
    ioe.printStackTrace();
    printOut(ioe.getMessage(), out);
  }
  // end of jsseURLConnect method */
}
/*
 * This method contains an example of how to use the URL and
 *  URLConnection objects to create a new SSL connection, using
 *  WebLogic SSL client classes.
 */
public static void wlsURLConnect(String host, String port,
                                 String sport, String query,
                                 OutputStream out) {
  try {
    if (query == null)
      query = "/examplesWebApp/index.jsp";
    // The following protocol registration is taken care of in the
   //  normal startup sequence of WebLogic. It can be turned off
```

```
    //  using the console SSL panel.
    //
    // We duplicate it here as a proof of concept in a stand alone
    //  java application. Using the URL object for a new connection
    //  inside of WebLogic would work as expected.
     java.util.Properties p = System.getProperties();
     String s = p.getProperty("java.protocol.handler.pkgs");
     if (s == null) {
       s = "weblogic.net";
     } else if (s.indexOf("weblogic.net") == -1) {
       s += "|weblogic.net";
     }
     p.put("java.protocol.handler.pkgs", s);
     System.setProperties(p);
     printSecurityProviders(out);
     // end of protocol registration
     printOut(" Trying a new HTTP connection using WLS client classes -
                 \n\thttp://" +  host + ":" + port + query + "\n", out);
     URL wlsUrl = null;
     try {
       wlsUrl = new URL("http", host, Integer.valueOf(port).intValue(), query);
       weblogic.net.http.HttpURLConnection connection =
         new weblogic.net.http.HttpURLConnection(wlsUrl);
       tryConnection(connection, out);
     } catch (Exception e) {
       printOut(e.getMessage(), out);
       e.printStackTrace();
       printSecurityProviders(System.out);
       System.out.println("----");
     }
     printOut(" Trying a new HTTPS connection using WLS client classes -
               \n\thttps://" + host + ":" + sport + query + "\n", out);
     wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(), query);
     weblogic.net.http.HttpsURLConnection sconnection =
       new weblogic.net.http.HttpsURLConnection(wlsUrl);
    // Only when one has a two-way SSL connection, i.e. ClientCertificateEnforced
    // is selected in the server under the SSL tab, the following private key
    // and the client cert chain is used.
     File ClientKeyFile  = new File ("clientkey.pem");
     File ClientCertsFile  = new File ("client2certs.pem");
     if (!ClientKeyFile.exists() || !ClientCertsFile.exists())
{
      System.out.println("Error : clientkey.pem/client2certs.pem is not present
                          in this directory.");
      System.out.println("To create it run - ant createmycerts.");
      System.exit(0);
}
     InputStream [] ins = new InputStream[2];
     ins[0] = new FileInputStream("client2certs.pem");
```

```
      ins[1] = new FileInputStream("clientkey.pem");
      String pwd = "clientkey";
      sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());

      tryConnection(sconnection, out);
    } catch (Exception ioe) {
      printOut(ioe.getMessage(), out);
      ioe.printStackTrace();
    }
  }
  /* This inner class is a null version which always returns true when checking
      the server certificate SubectDN CommonName against the hostname of the
      server we're connecting to. Uncomment this class and the registration of
      this class in JSSE (also commented out above in the jsseURLConnect method)
      to observe how it works.
  *//*
  static class NullHostnameVerifier implements com.sun.net.ssl.HostnameVerifier {
    public boolean verify(String urlHostname, String certHostname) {
      return true;
    }
    }*/
}
```

SSLSocketClient Sample

The SSLSocketClient sample demonstrates how to use the secure port to connect to a JSP served by WebLogic Server and display the results of that connection. It shows how to implement the following functions:

- Initializing an SSLContext with client identity, a HostnameVerifierJSSE, and a NulledTrustManager
- Creating a keystore and retrieving the private key and certificate chain
- Using an SSLSocketFactory
- Using HTTPS to connect to a JSP served by WebLogic Server
- Implementing the `javax.net.ssl.HandshakeCompletedListener` interface
- Creating a dummy implementation of the `weblogic.security.SSL.HostnameVerifierJSSE` class to verify that the server the example connects to is running on the desired host

Listing 3-11 shows a sample SSLSocketClient. This code taken from the `SSLSocketClient.java` file located at `SAMPLES_HOME\server\src\examples\security\sslclient`.

Listing 3-11 SSLSocketClient Sample

```
package examples.security.sslclient;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Hashtable;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.Certificate;
import weblogic.security.SSL.HostnameVerifierJSSE;
import weblogic.security.SSL.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSession;
import weblogic.security.SSL.SSLSocketFactory;
import weblogic.security.SSL.TrustManagerJSSE;

  /**
  * A Java client demonstrates connecting to a JSP served by WebLogic Server
  * using the secure port and displays the results of the connection.
  *
  * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
  */

public class SSLSocketClient {
  public void SSLSocketClient() {}
  public static void main (String [] argv)
    throws IOException {
    if ((argv.length < 2) || (argv.length > 3)) {
      System.out.println("usage:   java SSLSocketClient host sslport
                                      <HostnameVerifierJSSE>");
      System.out.println("example: java SSLSocketClient server2.weblogic.com 443
                                      MyHVClassName");
      System.exit(-1);
    }
    ////////////////////////////////

    try {
      System.out.println("\nhttps://" + argv[0] + ":" + argv[1]);
```

```
      System.out.println(" Creating the SSLContext");
      SSLContext sslCtx = SSLContext.getInstance("https");

      File KeyStoreFile  = new File ("mykeystore");
      if (!KeyStoreFile.exists())
{
   System.out.println("Keystore Error : mykeystore is not present in this
                        directory.");
   System.out.println("To create it run - ant createmykeystore.");
   System.exit(0);
}

      System.out.println(" Initializing the SSLContext with client\n" +
                         "  identity (certificates and private key),\n" +
                         "  HostnameVerifierJSSE, AND NulledTrustManager");

     // Open the keystore, retrieve the private key, and certificate chain
      KeyStore ks = KeyStore.getInstance("jks");
      ks.load(new FileInputStream("mykeystore"), null);
      PrivateKey key = (PrivateKey)ks.getKey("mykey", "testkey".toCharArray());
      Certificate [] certChain = ks.getCertificateChain("mykey");
      sslCtx.loadLocalIdentity(certChain, key);

      HostnameVerifierJSSE hVerifier = null;
      if (argv.length < 3)
        hVerifier = new NulledHostnameVerifier();
      else
        hVerifier = (HostnameVerifierJSSE) Class.forName(argv[2]).newInstance();
      sslCtx.setHostnameVerifierJSSE(hVerifier);
      TrustManagerJSSE tManager = new NulledTrustManager();
      sslCtx.setTrustManagerJSSE(tManager);
      System.out.println(" Creating new SSLSocketFactory with SSLContext");
      SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();
      System.out.println(" Creating and opening new SSLSocket with
                              SSLSocketFactory");

      // using createSocket(String hostname, int port)
      SSLSocket sslSock = (SSLSocket) sslSF.createSocket(argv[0],
                                              new Integer(argv[1]).intValue());
      System.out.println(" SSLSocket created");
      sslSock.addHandshakeCompletedListener(new MyListener());
      OutputStream out = sslSock.getOutputStream();

      // Send a simple HTTP request
      String req = "GET /examplesWebApp/ShowDate.jsp HTTP/1.0\r\n\r\n";
      out.write(req.getBytes());

      // Retrieve the InputStream and read the HTTP result, displaying
      // it on the console
      InputStream    in = sslSock.getInputStream();
```

```
      byte buf[] = new byte[1024];
      try
      {
        while (true)
        {
          int amt = in.read(buf);
          if (amt == -1) break;
          System.out.write(buf, 0, amt);
        }
      }
      catch (IOException e)
      {
        return;
      }
      sslSock.close();
      System.out.println(" SSLSocket closed");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
/**
 * MyListener implements the interface
 * <code>javax.net.ssl.HandshakeCompletedListener</code> and shows the user how
 * to receive notifications about the completion of an SSL protocol handshake
 * on a given SSL connection.
 *
 * It also shows the user the number of times an SSL handshake takes
 * place on a given SSL connection.
 *
 */
  static class MyListener implements javax.net.ssl.HandshakeCompletedListener
  {
    public void handshakeCompleted(javax.net.ssl.HandshakeCompletedEvent event)
    {
      SSLSession session = event.getSession();
      System.out.println("Handshake Completed with peer " +
                                 session.getPeerHost());
      System.out.println("cipher: " + session.getCipherSuite());
      javax.security.cert.X509Certificate[] certs = null;
      try
      {
        certs = session.getPeerCertificateChain();
      }
      catch (javax.net.ssl.SSLPeerUnverifiedException puv)
      {
        certs = null;
      }
      if  (certs != null)
      {
```

```
        System.out.println("peer certificates:");
        for (int z=0; z<certs.length; z++) System.out.println(
                                            "certs["+z+"]: " + certs[z]);
      }
      else
      {
        System.out.println("No peer certificates presented");
      }
    }
  }
}
```

SSLClientServlet Sample

SSLClientServlet is a simple servlet wrapper of SSLClient sample.

Listing 3-12 shows a sample SSLClientServlet. This code taken from the `SSLSClientServlet.java` file located at `SAMPLES_HOME\server\src\examples\security\sslclient`.

Listing 3-12 SSLClientServlet Sample Code

```
package examples.security.sslclient;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * SSLClientServlet is a simple servlet wrapper of
 * examples.security.sslclient.SSLClient.
 *
 * @see SSLClient
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */

public class SSLClientServlet extends HttpServlet {
  public void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    response.setHeader("Pragma", "no-cache"); // HTTP 1.0
```

```
    response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
    ServletOutputStream out = response.getOutputStream();

    out.println("<br><h2>ssl client test</h2><br><hr>");
    String[] target = request.getParameterValues("url");
    try {

      out.println("<h3>wls ssl client classes</h3><br>");
      out.println("java SSLClient wls localhost 7001 7002
                       /examplesWebApp/SnoopServlet.jsp<br>");
      out.println("<pre>");
      SSLClient.wlsURLConnect("localhost", "7001", "7002",
                                "/examplesWebApp/SnoopServlet.jsp", out);
      out.println("</pre><br><hr><br>");

    } catch (IOException ioe) {
      out.println("<br><pre> "+ioe.getMessage()+"</pre>");
      ioe.printStackTrace();
    }
  }
}
```

Using Two-Way SSL Authentication

When using certificate authentication, WebLogic Server sends a digital certificate to the requesting client. The client examines the digital certificate to ensure that it is authentic, has not expired, and matches the WebLogic Server instance that presented it.

With two-way SSL authentication (a form of mutual authentication), the requesting client also presents a digital certificate to WebLogic Server. When WebLogic Server is configured for two-way SSL authentication, requesting clients are required to present digital certificates from a specified set of certificate authorities. WebLogic Server accepts only digital certificates that are signed by root certificates from the specified certificate authorities.

For information on how to configure WebLogic Server for two-way SSL authentication, see the "Configuring the SSL Protocol" section in *Managing WebLogic Security*.

The following sections describe the different ways two-way SSL authentication can be implemented in WebLogic Server.

Two-Way SSL Authentication with JNDI

When using JNDI for two-way SSL authentication in a Java client, use the `setSSLClientCertificate()` method of the WebLogic JNDI Environment class. This method sets a private key and chain of X.509 digital certificates for client authentication. To supply the Java client's digital certificate and private key read the Definite Encoding Rules (DER) files that contain the digital certificate and private key into an X509 object, and then set the X509 object in a JNDI hash table. Use the JNDI properties described in "Using JNDI Authentication" to specify the information required for authentication.

To pass digital certificates to JNDI, create an array of `InputStreams` opened on files containing DER-encoded digital certificates and set the array in the JNDI hash table. The first element in the array must contain an `InputStream` opened on the Java client's private key file. The second element must contain an `InputStream` opened on the Java client's digital certificate file. (This file contains the public key for the Java client.) Additional elements may contain the digital certificates of the root certificate authority and the signer of any digital certificates in a certificate chain. A certificate chain allows WebLogic Server to authenticate the digital certificate of the Java client if that digital certificate was not directly issued by a certificate authority registered for the Java client in the WebLogic Server keystore file.

You can use the `weblogic.security.PEMInputStream` class to read digital certificates stored in Privacy Enhanced Mail (PEM) files. This class provides a filter that decodes the base 64-encoded DER certificate into a PEM file.

Listing 3-13 demonstrates how to use mutual authentication in a Java client.

Listing 3-13 Example of Mutual Authentication

```
import java.io.FileInputStream;
import java.io.InputStream;
import javax.naming.Context;
import weblogic.jndi.Environment;
import weblogic.security.PEMInputStream;


public class JavaClient
{

  public static void main(String[] args)
```

```
  {
    Context ctx = null;

    String url = args[0];
    try
    {
      Environment env = new Environment();

      env.setProviderUrl(url);

      // The second and third args are username and password
      if (args.length >= 3)
      {
env.setSecurityPrincipal(args[1]);
env.setSecurityCredentials(args[2]);
      }

      // Fourth and fifths arguments are private key and
      // public key.
      if (url.startsWith("t3s") && args.length >= 5)
      {
InputStream[] certs = new InputStream[args.length - 3];
for (int q = 3; q < args.length; q++)
{
  String file = args[q];
  InputStream is = new FileInputStream(file);

  if (file.toLowerCase().endsWith(".pem"))
  {
    is = new PEMInputStream(is);
  }
  certs[q - 3] = is;
}
env.setSSLClientCertificate(certs);
      }
      ctx = env.getInitialContext();
      ...
```

When the JNDI `getInitialContext()` method is called, the Java client and WebLogic Server execute mutual authentication in the same way that a Web browser performs mutual authentication to get a secure Web server connection. An exception is thrown if the digital certificates cannot be validated or if the Java client's digital certificate cannot be authenticated in the security realm. The authenticated User object is stored on the Java client's server thread and is used for checking the permissions governing the Java client's access to any protected WebLogic Server resources.

When you use the WebLogic JNDI Environment class, you must create a new Environment object for each call to the `getInitialContext()` method. Once you specify a User and security credentials, both the user and their associated credentials remain set in the Environment object. If you try to reset them and then call the JNDI `getInitialContext()` method, the original User and credentials are used.

When you use mutual authentication from a Java client, WebLogic Server gets a unique Java Virtual Machine (JVM) ID for each client JVM so that the connection between the Java client and WebLogic Server is constant. Unless the connection times out from lack of activity, it persists as long as the JVM for the Java client continues to execute. The only way a Java client can negotiate a new SSL connection reliably is by stopping its JVM and running another instance of the JVM.

If you have not configured an Identity Asserter to do certificate-based authentication, a Java client running in a JVM with an SSL connection can change the WebLogic Server User identity by creating a new JNDI `InitialContext` and supplying a new username and password in the JNDI `SECURITY_PRINCIPAL` and `SECURITY_CREDENTIALS` properties. Any digital certificates passed by the Java client after the SSL connection is made are not used. The new WebLogic Server User continues to use the SSL connection negotiated with the initial User's digital certificate.

If you have configured an Identity Asserter to do certificate-based authentication, WebLogic Server passes the digital certificate from the Java client to the implementation of the Identity Asserter class that implements the UserNameMapper interface and the user-name-mapper class maps the digital certificate to a WebLogic Server user name. Therefore, if you want to set a new user identity when you use the certificate-based identity assertion, the Java client must create a new Environment because the original Environment processed the digital certificate and cached the first set of credentials. This is because the digital certificate is processed only at the time of the first connection request from the JVM for each Environment.

Using Two-Way SSL Authentication Between WebLogic Server Instances

You can use two-way SSL authentication in server-to-server communication in which one WebLogic Server instance is acting as the client of another WebLogic Server instance. Using two-way SSL authentication in server-to-server communication enables you to have dependable, highly-secure connections, even without the more familiar client/server environment.

Listing 3-14 shows an example of how to establish a secure connection from a servlet running in one instance of WebLogic Server to a second WebLogic Server instance called `server2.weblogic.com`.

Listing 3-14 Establishing a Secure Connection to Another WebLogic Server

```
FileInputStream [] f = new FileInputStream[3];
   f[0]= new FileInputStream("demokey.pem");
   f[1]= new FileInputStream("democert.pem");
   f[2]= new FileInputStream("ca.pem");

Environment e = new Environment ();
e.setProviderURL("t3s://server2.weblogic.com:443");
e.setSSLClientCertificate(f);
e.setSSLServerName("server2.weblogic.com");
e.setSSLRootCAFingerprints("ac45e2d1ce492252acc27ee5c345ef26");


e.setInitialContextFactory
       ("weblogic.jndi.WLInitialContextFactory");
Context ctx = new InitialContext(e.getProperties())
```

In Listing 3-14, the WebLogic JNDI Environment class creates a hash table to store the following parameters:

- `setProviderURL`—the URL of the WebLogic Server functioning as a client. The URL specifies the T3S protocol which is a WebLogic Server proprietary protocol built on the SSL protocol. The SSL protocol protects the connection and communication between two WebLogic Servers.
- `setSSLClientCertificate`—specifies a certificate chain to use for the connection. A certificate chain is an array that contains a private key, the matching public key, and a chain of digital certificates for trusted certificate authorities, each of which is the issuer of the previous digital certificate.
- `setSSLServerName`—specifies the name of the WebLogic Server presenting the digital certificate. The name is compared to the common name field in the digital certificate of the WebLogic Server. This parameter is used to prevent man-in-the-middle attacks.

- `setSSLRootCAFingerprint`—specifies that the digital certificate must be issued by a certificate authority with the specified fingerprint. This parameter is used to prevent man-in-the-middle attacks.

Using Two-Way SSL Authentication with Servlets

To authenticate Java clients in a servlet (or any other server-side Java class), you must check whether the client presented a digital certificate and if so, whether the certificate was issued by a trusted certificate authority. The servlet writer is responsible for asking whether the Java client has a valid digital certificate. When writing servlets with the WebLogic Servlet API, you must access information about the SSL connection through the `getAttribute()` method of the `HTTPServletRequest` object.

The following attributes are supported in WebLogic Server servlets:

- `javax.servlet.request.X509Certificate` `java.security.cert.X509Certificate []`—returns an array of the X509Certificate.
- `javax.servlet.request.cipher_suite`—returns a string representing the cipher suite used by HTTPS.
- `javax.servlet.request.key_size`— returns an integer (0, 40, 56, 128, 168) representing the bit size of the algorithm.
- weblogic.servlet.request.SSLSession javax.net.ssl.SSLSession—returns the SSL Session object that contains the cipher suite and the dates on which the object was created and last used.

You have access to the user information defined in the digital certificates. When you get the `javax.servlet.request.X509Certificate` attribute, it is an array of `java.security.cert` X509Certificate. You simply cast it to that and look at the certificates.

A digital certificate includes information, such as the following:

- The name of the subject (holder, owner) and other identification information required to verify the unique identity of the subject, such as the uniform resource locator (URL) of the Web server using the digital certificate, or an individual user's e-mail address
- The subject's public key

- The name of the certificate authority that issued the digital certificate
- A serial number
- The validity period (or lifetime) of the digital certificate (as defined by a start date and an end date)

Using a Custom Host Name Verifier

A Host Name Verifier validates that the host to which an SSL connection is made is the intended or authorized party. A Host Name Verifier is useful when a WebLogic client or a WebLogic Server is acting as an SSL client to another application server. It prevents man-in-the-middle attacks.

By default, WebLogic Server, as a function of the SSL handshake, compares the common name in the SubjectDN of the SSL server's digital certificate with the host name of the SSL server used to initiate the SSL connection. If these names do not match, the SSL connection is dropped.

The dropping of the SSL connection is caused by the SSL client which validates the host name of the server against the digital certificate of the server. If anything but the default behavior is desired, you can either turn off host name verification or register a custom host name verifier. Turning off host name verification leaves WebLogic Server vulnerable to man-in-the-middle attacks.

Note: Turn off host name verification when using the demonstration digital certificates shipped with WebLogic Server.

You can turn off host name verification in the following ways:

- In the Administration Console, check the Hostname Verification Ignored attribute under the SSL tab on the Server node.
- On the command line of the SSL client, enter the following argument:

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

You can write a custom Host Name Verifier. The `weblogic.security.SSL.HostnameVerifierJSSE` interface provides a callback mechanism so that you can define a policy for handling the case where the server name that is being connected to does not match the server name in the SubjectDN of the server's digital certificate.

To use a custom Host Name Verifier, create a class that implements the `weblogic.security.SSL.HostnameVerifierJSSE` interface and define the methods that capture information about the server's security identity.

Note: This interface takes new style certificates and replaces the `weblogic.security.SSL.HostnameVerifier` interface, which is being deprecated in this release of WebLogic Server.

Before you can use a custom Host Name Verifier, you need to define the class for your implementation in the following ways:

- In the Administration Console, define the class for your Host Name Verifier in the Hostname Verifier attribute under the SSL tab on the Server node.
- On the command line, enter the following argument:

  `-Dweblogic.security.SSL.HostnameVerifierJSSE=`*`hostnameverifier`*

  where *`hostnameverifier`* is the name of the class that implements the custom Host Name Verifier.

An example of a custom Host Name Verifier (See Listing 3-15) is available in the SSLclient code example in the `examples.security.sslclient` package in the `SAMPLES_HOME\server\src\examples\security\sslclient` directory. This code example contains a NulledHostnameVerifier class which always returns true for the comparison. This sample allows the WebLogic SSL client to connect to any SSL server regardless of the server's host name and digital certificate SubjectDN comparison.

Listing 3-15 HostNameVerifierJSSE Sample Code

```
package examples.security.sslclient;

/**
 * HostnameVerifierJSSE provides a callback mechanism so that
 * implementers of this interface can supply a policy for handling
 * the case where the host that's being connected to and the server
 * name from the certificate SubjectDN must match.
 *
 * This is a null version of that class to show the WebLogic SSL
* client classes without major problems. For example, in this case,
 * the client code connects to a server at 'localhost' but the
 * democertificate's SubjectDN CommonName is 'bea.com' and the
 * default WebLogic HostnameVerifierJSSE does a String.equals() on
```

```
 * those two hostnames.
 *
 * @see HostnameVerifier#verify
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights
 * Reserved.
 */

public class NulledHostnameVerifier implements
                    weblogic.security.SSL.HostnameVerifierJSSE {
  public boolean verify(String urlHostname, String certHostname) {
    return true;
  }
}
```

Using a Trust Manager

The `weblogic.security.SSL.TrustManagerJSSE` interface allows you to override validation errors in a peer's digital certificate and continue the SSL handshake. You can also use the interface to discontinue an SSL handshake by performing additional validation on a server's digital certificate chain.

Note: This interface takes new style certificates and replaces the `weblogic.security.SSL.TrustManager` interface, which is being deprecated in this release of WebLogic Server.

When an SSL client connects to an SSL server, the SSL server presents its digital certificate chain to the client for authentication. That chain could contain an invalid digital certificate. The SSL specification says that the client should drop the SSL connection upon discovery of an invalid certificate. Web browsers, however, ask the user whether to ignore the invalid certificate and continue up the chain to determine if it is possible to authenticate the SSL server with any of the remaining certificates in the certificate chain. The Trust Manager eliminates this inconsistent practice by enabling you to control when to continue or discontinue an SSL connection. Using a Trust Manager you can perform custom checks before continuing an SSL connection. For example, you can use the Trust Manager to specify that only users from specific localities, such as towns, states, or countries, or users with other special attributes, to gain access via the SSL connection.

Use the `weblogic.security.SSL.TrustManagerJSSE` interface to create a Trust Manager. The interface contains a set of error codes for certificate verification. You can also perform additional validation on the peer certificate and interrupt the SSL

handshake if need be. After a digital certificate has been verified, the `weblogic.security.SSL.TrustManagerJSSE` interface uses a callback function to override the result of verifying the digital certificate. You can associate an instance of a Trust Manager with an SSL Context through the `setTrustManager()` method. The `weblogic.security.SSL.TrustManagerJSSE` interface conforms to the JSSE specification. Note that the use of a Trust Manager may potentially impact performance depending on the checks performed. You can only set up a Trust Manger programmatically; its use cannot be defined through the Administration Console or on the command-line.

An example of a NulledTrustManager (See Listing 3-16) is available in the SSLclient code example in the `examples.security.sslclient` package in the `SAMPLES_HOME\server\src\examples\security\sslclient` directory.

Listing 3-16 TrustManager Code Example

```
package examples.security.sslclient;

import weblogic.security.SSL.TrustManagerJSSE;
import javax.security.cert.X509Certificate;

public class NulledTrustManagerJSSE implements TrustManagerJSSE{

  public boolean certificateCallback(X509Certificate[] o, int validateErr) {
   System.out.println(" --- Do Not Use In Production ---\n" + "  By using this " +
                      "NulledTrustManager, the trust in the server's identity "+
                      "is completely lost.\n --------------------------------");
    for (int i=0; i<o.length; i++)
      System.out.println(" certificate " + i + " -- " + o[i].toString());
    return true;
  }
}
```

The SSLSocketClient example uses the custom Trust Manager show above. The SSLSocketClient shows how to set up a new SSL connection by using an SSL Context with the Trust Manager. This example is at this location on WebLogic Server:

`SAMPLES_HOME\server\src\examples\security\sslclient`

Using an SSLContext

SSLContext is used to implement a secure socket protocol that holds information such as Host Name Verifier and Trust Manager for a given set of SSL connections. An instance of the SSLContext class is used as a factory for SSL sockets. For example, all sockets that are created by socket factories provided by the SSLContext can agree on session state by using the handshake protocol associated with the SSLContext. Each instance can be configured with the keys, certificate chains, and trusted root CAs that it needs to perform authentication. These sessions are cached so that other sockets created under the same SSLContext can reuse them later. See Modifying Parameters for Session Caching in the *Administration Guide* for more information on session caching. To associate an instance of a Trust Manager class with its SSLContext, use the setTrustManagerJSSE method.

You can only set up an SSL context programmatically; not by using the Administration Console or the command line. A Java new expression or the `getInstance()` method of the SSLContext class can create an SSLContext object. The `getInstance()` method is static and it generates a new SSLContext object that implements the specified secure socket protocol. An example of using SSLContext is provided in the SSLSocketClient sample in the `SAMPLES_HOME\server\src\examples\security\sslclient` directory. This example shows how to create a new SSL Socket Factory that will create a new SSL Socket using SSLContext: SSLContext conforms to Sun Microsystems's Java Secure Socket Extension (JSSE), so it is forward-compatible code.

Listing 3-17 shows a sample instantiation.

Listing 3-17 SSL Context Code Example

```
import weblogic.security.SSL.SSLContext;

SSLcontext sslctx = SSLContext.getInstance ("https")
```

Using an SSLServerSocketFactory

Instances of this class create and return SSL sockets. This class extends `javax.net.SocketFactory`.

Listing 3-18 shows a sample instantiation.

Listing 3-18 SSLServerSocketFactory Code Example

```
import weblogic.security.SSL.SSLSocketFactory;

  SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();
```

Using URLs to Make Outbound SSL Connections

You can use a URL object to make an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server. The `weblogic.net.http.HttpsURLConnection` class provides a way to specify the security context information for a client including the digital certificate and private key of the client. Instances of this class represent an HTTPS connection to a remote object.

The SSLClient code example demonstrates using the WebLogic URL object to make an outbound SSL connection (see Listing 3-19). The code example shown in Listing 3-19 is excerpted from the `SSLClient.java` file in the *`SAMPLES_HOME`*`\server\src\examples\security\sslclient` directory.

Listing 3-19 WebLogic URL Outbound SSL Connection Code Example

```
wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(),
                 query);
weblogic.net.http.HttpsURLConnection sconnection =
         new weblogic.net.http.HttpsURLConnection(wlsUrl);
```

In addition, the SSLClient code example demonstrates using the Java Secure Socket Extension (JSSE) application programming interface (API) to make an outbound SSL connection (see Listing 3-20).

Listing 3-20 JSSE URL Outbound SSL Connection Code Example

```
URL jsseUrl = new URL( "https", host,
                       Integer.valueOf(sport).intValue(), query,
      new com.sun.net.ssl.internal.www.protocol.https.Handler() );
java.net.HttpURLConnection sconnection =
        (java.net.HttpURLConnection) jsseUrl.openConnection();
```

CHAPTER

4 Securing EJB Applications

You can use deployment descriptors and the Administration Console to secure EJBs just as you can with Web applications. See "Securing Web Applications (Thin Clients)" on page 1 for a information on securing Web applications.

This section presents the following topics:

- Adding Declarative Security to EJBs
- Using the <global-role/> Tag With EJBs
- Adding Programmatic Security to EJBs

Adding Declarative Security to EJBs

To implement declarative security in EJBs you use deployment descriptors (`ejb-jar.xml` and `weblogic-ejb-jar.xml`) to define the security requirements. Listing 4-1 shows examples of how to use the `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptors to map security role names to a security realm. The deployment descriptors map the application's logical security requirements to its runtime definitions. And at runtime, the EJB container uses the security definitions to enforce the requirements.

To configure a security in the EJB deployment descriptors, perform the following steps (see Listing 4-1):

1. Use a text editor to create `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptor files.
2. In the `ejb-jar.xml` file, define the role name, ejb name, and method name.
3. In the WebLogic-specific EJB deployment descriptor file, `weblogic-ejb-jar.xml`, define the role name and link it to one or more principals in a security realm.

For more information on configuring security in the ejb-jar.xml file, see the *Sun Microsystems Enterprise JavaBeans Specification, Version 2.0* which is at this location on the Internet: `http://java.sun.com/products/ejb/docs.html`.

Listing 4-1 Using the ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm

ejb-jar.xml entries:

```
          ...
<assembly-descriptor>
  <security-role>
    <role-name>manger</role-name>
    <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
          ...
</assembly-descriptor>
           ...
```

weblogic-ejb-jar.xml entries:

```
  <security-role-assignment>
      <role-name>manager</role-name>
      <principal-name>al</principal-name>
      <principal-name>george</principal-name>
      <principal-name>ralph</principal-name>
  </security-role-assignment>
     ...
```

Using the <global-role/> Tag With EJBs

With WebLogic Server versions 7.0 SP1 and later, there are four different options, or approaches, that you can use to configure security in EJBs:

- **Approach #1: Always use the deployment descriptors.** In this case, you simply elect to never use the Administration Console to configure security and, instead, use the deployment descriptors. The security configuration settings are read from the deployment descriptors every time the EJB is deployed or redeployed. To use this approach, the Administration Console's Ignore Security Data in Deployment Descriptors attribute on the General tab of a security realm must be left in its default state—unchecked.
- **Approach #2: Always use the Administration Console.** In this case, before you deploy the application for the first time, you must check the Administration Console's Ignore Security Data in Deployment Descriptors attribute so that the deployment descriptors are ignored on initial deployment and subsequent redeployments. Then you use the Administration Console to configure Global Roles after the application is deployed.
- **Approach #3: Use the deployment descriptors to configure security at initial application deployment and then use the Administration Console to configure security from that point forward.** In this case, the security configuration settings are read from the deployment descriptors upon initial deployment. Then you check the Ignore Security Data in Deployment Descriptors attribute so that deployment descriptors are ignored on subsequent redeployments and you use the Administration Console to make all subsequent changes to the application's security configuration.
- **Approach #4: Use the deployment descriptors to configure policies and use the Administration Console to configure roles.** In WebLogic Server 7.0 SP1 and later, support was added for the `<global-role/>` tag for use in the `weblogic-ejb-jar.xml` deployment descriptors. You can use this tag, instead of the `<principal-name>` tag, to explicitly indicate that you want the roles defined in the deployment descriptors by the `<role-name>` tag to use the mappings that you specify in the Administration Console.

  Thus, the `<global-role/>` tag gives you the flexibility of not having to specify a specific role mapping for each role defined in the deployment descriptors for a particular EJB. Rather, you can use the Administration Console to specify and

modify a specific role mapping for each defined role at anytime. Additionally, because you may elect to use this tag on some EJBs and not others, it is not necessary to check the Ignore Security Data In Deployment Descriptors attribute on the General tab of the security realm. Thus, within the same security realm, deployment descriptors can be used to specify and modify security for some EJBs, while the Administration Console can be used to specify and modify security for others.

Listing 4-2 shows how to use the <global-role/> tag with the `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptors.

Listing 4-2 Using the <global-role> tag in EJB Deployment Descriptors for Role Mapping

`ejb-jar.xml entries:`

```
          ...
<assembly-descriptor>
  <security-role>
    <role-name>manger</role-name>
    <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
          ...
</assembly-descriptor>
           ...
```

`weblogic-ejb-jar.xml entries:`

```
  <security-role-assignment>
      <role-name>manager</role-name>
      <global-role/>
      ...
  </security-role-assignment>
   ...
```

For information about how to use the Administration Console to configure security for EJBs, See *Managing WebLogic Security*.

Adding Programmatic Security to EJBs

To implement programmatic security in EJBs you use the `javax.ejb.EJBContext.getCallerPrincipal()` and the `javax.ejb.EJBContext.isCallerInRole()` methods.

getCallerPrincipal

You use the `getCallerPrincipal()` method to determine the caller of the enterprise java bean. The `javax.ejb.EJBContext.getCallerPrincipal()` method obtains the `java.security.principal` and returns the Principal object that identifies the caller. You can use the `java.lang.Class.getName()` method to retrieve the current user's name and then do a lookup to determine whether the user has the privileges needed to access the resource.

For more information about how to use the `getCallerPrincipal()` method, see http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Security5.html.

isCallerInRole

The `isCallerInRole()` method is used to determine if the caller (the current user) has been assigned a Role that is authorized to perform actions on the WebLogic Server resources in that thread of execution. For example, the method `javax.ejb.EJBContext.isCallerInRole("admin")`will return `true` if the current user has `admin` privileges.

For more information about how to use the `isCallerInRole()` method, see http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Security5.html.

For Javadoc for the isCallerInRole() method, see http://java.sun.com/products/ejb/javadoc-1.1/javax/ejb/EJBContext.html#isCallerInRole(java.lang.String).

CHAPTER

5 Protecting Application Server Resources

There are primarily two ways to protect application server resources: network connection filters and J2EE sandbox security. This section discussed these topics:

- Using Network Connection Filters to Protect Application Server Resources
- Using J2EE Sandbox Security to Protect Application Server Resources

Using Network Connection Filters to Protect Application Server Resources

Security policies allow you to secure WebLogic Server resources using some characteristic of a user. However, you can add an additional layer of security by using connection filters to filter network connections. For example, you can deny any non-SSL connections originating outside of your corporate network.

Network connection filters are a type of firewall in that they can be configured to filter on protocols, IP addresses, and DNS node names.

This section covers the following topics:

- Connection Filter Interfaces
- Connection Filter Classes
- Guidelines for Writing Connection Filter Rules

- Configuring the Default Connection Filter
- Developing Custom Connection Filters
- Connection Filter Examples

Connection Filter Interfaces

Two *`weblogic.security.net`* interfaces are provided for implementing connection filters:

- ConnectionFilter interface
- ConnectionFilterRulesListener interface

ConnectionFilter interface

This interface defines the `accept()` method which is used to implement connection filtering. To program the server to perform connection filtering, you must instantiate a class that implements this interface and then define that class in the Administration Console. This is the minimum implementation requirement for connection filtering. However, implementing this interface alone does not permit the use of the Administration Console to enter filtering rules to restrict client connections, rather some other form (such as a flat file, which is defined in the console) must be used for that purpose.

For a code example of how to use this interface, see "Connection Filter Examples" on page 5-8.

ConnectionFilterRulesListener interface

This interface defines two methods which are used to implement connection filtering: `setRules()` and `checkRules()`. Implementing this interface in addition to the ConnectionFilter interface allows the use of the Administration Console to enter filtering rules to restrict client connections.

Note: To be able to enter and edit connection filtering rules on the Administration Console, you must implement the ConnectionFilterRulesListener interface; otherwise some other means must be used. For example, you could use a flat file.

For a code example of how to use this interface, see "Connection Filter Examples" on page 5-8.

Connection Filter Classes

Two *weblogic.security.net* classes are provided for implementing connection filters:

- ConnectionFilterImpl Class
- ConnectionEvent Class

ConnectionFilterImpl Class

This class is a default connection filter implementation. This class implements the ConnectionFilter and ConnectionFilterRulesListener interfaces. It accepts all incoming connections and also provides static factory methods that allow the server to obtain the current connection filter.

This class is provided ready to use as part of the WebLogic Server product. To configure this class for use, see "Configuring the Default Connection Filter" on page 5-6.

ConnectionEvent Class

This is the class from which all event state objects are derived. All events are constructed with a reference to the object, that is, the source that is logically deemed to be the object upon which a specific event initially occurred.

For a code example of how to use this class, see Listing 5-1.

Guidelines for Writing Connection Filter Rules

This section describes how connection filter rules are written and evaluated. If no connection rules are specified, all connections are accepted.

Connection filter rules can be written in a flat file or input directly on the Administration Console, depending on how you implement connection filtering. The Network Connection Filter code examples, which are located in the *SAMPLES_HOME*`\server\src\examples\security\net` directory, demonstrate both methods.

The following sections provide information and guidelines for writing connection filter rules:

- Connection Filter Rules Syntax
- Types of Connection Filter Rules
- How Connection Filter Rules are Evaluated

Connection Filter Rules Syntax

The syntax of connection filter rules is as follows:

- Each rule must be written on a single line in the source code.
- Tokens in a rule are separated by white space.
- A pound sign (#) is the comment character. Everything after a pound sign on a line is ignored.
- Whitespace before or after a rule is ignored.
- Lines consisting only of whitespace or comments are skipped.

All rules have the following format:

```
target localAddress localPort action protocols
```

where

- `target` specifies one or more servers to filter.
- `localAddress` defines the host address of the server. (If you specify an asterisk (`*`), the match returns all local IP addresses.)
- `localPort` defines the port on which the server is listening. (If you specify an asterisk, the match returns all available ports on the server).
- `action` specifies the action to perform. This value must be `allow` or `deny`.

- `protocols` is the list of protocol names to match. The following protocols may be specified: `http, https, t3, t3s, giop, giops, dcom, ftp`. If no protocol is defined, all protocols will match a rule.

Types of Connection Filter Rules

Two types of filter rules are recognized:

- Fast rules

  A fast rule applies to a hostname or IP address with an optional netmask. If a hostname corresponds to multiple IP addresses, multiple rules are generated (in no particular order). Netmasks can be specified either in numeric or dotted-quad form. For example:

```
dialup-555-1212.pa.example.net  127.0.0.1 7001 deny t3 t3s  # http(s) OK

192.168.81.0/255.255.254.0   127.0.0.1 8001 allow   # 23-bit netmask

192.168.0.0/16   127.0.0.1 8002 deny   # like /255.255.0.0
```

  Hostnames for fast rules are looked up once at server startup. While this design greatly reduces overhead at connect time, it can result in the filter obtaining out of date information about what addresses correspond to a host name. BEA Systems recommends using numeric IP addresses instead.

- Slow rules

  A slow rule applies to part of a domain name. Since a rule requires a connect-time DNS lookup on the client-side in order to perform a match, a slow rule may be much slower than the fast rule. Slow rules are also subject to DNS spoofing. Slow rules are specified as follows:

```
*.script-kiddiez.org 127.0.0.1 7001 deny
```

  An asterisk only matches at the head of a pattern. If you specify an asterisk anywhere else in a rule, it is treated as part of the pattern. Note that the pattern will never match a domain name since an asterisk is not a legal part of a domain name.

How Connection Filter Rules are Evaluated

When a client connects to WebLogic Server, the rules are evaluated in the order in which they were written. The first rule to match determines how the connection is treated. If no rules match, the connection is permitted.

If you want to further protect your server and only allow connections from certain addresses, specify the last rule as:

```
0.0.0.0/0  *  *  deny
```

With this as the last rule, only connections that are allowed by preceding rules are allowed, all others are denied.

Configuring the Default Connection Filter

To configure the default connection filter, connectionFilterImpl, perform the following steps on the Administration Console:

1. Bring up the Administration Console in a browser.
2. To configure the default connection filter class, perform the following steps:
   a. Select the domain node in the left pane. The domain node is the top-level node. It is the name of your server's domain.
   b. Click on the Security Tab.
   c. Click on the Filter tab.
   d. Enter `weblogic.security.net.ConnectionFilterImpl` in the Connection Filter field.
   e. Enter a connection filter rule in the Connection Rules field. For example, to deny access to the node with IP address of `192.168.81.0`, enter:

      ```
      192.168.81.0  127.0.0.1 7001 deny
      ```

      Where the `192.168.81.0` is the IP address of the system to be denied access, `127.0.0.1` is the IP address is system running WebLogic Server, and `7001` is the port on which access is to be denied. For more information on connection filter rules syntax, see "Connection Filter Rules Syntax" on page 5-4.
3. Click Apply.
4. Restart your server, otherwise, the connection filter does not take effect.

   Note: Subsequent changes to the connection filter rules **do not** require a server restart.

5. To test whether the connection filter is working, perform the following steps:

   a. Open a browser on the system that has been denied access and enter the URL for Administration Console on the system running WebLogic Server. The URL format is as follows:

      `http://hostname:port/console`

      For example, `http://pcwiz:7001/console`

   b. If you are using Internet Explorer as your browser, the message **"The Page cannot be displayed"** is displayed in your browser.

Developing Custom Connection Filters

To develop custom connection filters with WebLogic Server, perform the following steps:

1. Write a class that implements the ConnectionFilter interface (minimum requirement).

   Or, optionally, if you want to use the Administration Console to enter and modify the connection filtering rules directly, write a class the implements both the ConnectionFilter interface and the ConnectionFilterRulesListener interface.

2. If you choose the minimum requirement in step 1 (only implementing the ConnectionFilter interface), enter the connection filtering rules in a flat file and define the location of the flat file in the class that implements the ConnectionFilter interface. For an example of how to use a flat file to implement connection filtering, see the `SimpleConnectionFilter.java` file in the *`SAMPLES_HOME`*`\server\src\examples\security\net` directory.

3. If you choose to implement both interfaces in step 1, use the Administration Console to configure the class in WebLogic Server and enter the connection filtering rules in the Administration Console. For instructions for configuring the class in the Administration Console, see the "Configuring Connection Filtering" section in *Managing WebLogic Security*. For an example of how to use the ConnectionFilterRulesListener interface to implement connection filtering, see the `SimpleConnectionFilter2.java` file in the *`SAMPLES_HOME`*`\server\src\examples\security\net` directory.

If connection filtering is implemented when a Java client or Web browser client tries to connect to WebLogic Server, WebLogic Server constructs a `ConnectionEvent` object and passes it to the `accept()` method of your connection filter class. The connection filter class examines the `ConnectionEvent` object and accepts the connection by returning or denies the connection by throwing a `FilterException`.

Both implemented classes call the `accept()` method after gathering information about the client connection. However, if you only implement the ConnectionFilter interface, the information gathered includes the remote IP address and the connection protocol (HTTP, HTTPS, T3, T3S, IIOP, or IIOPS). If you implement both interfaces, the information gathered includes the remote IP address, remote port number, local IP address, local port number and the connection protocol.

Connection Filter Examples

Two connection filter examples are provided with the WebLogic Server software. Both examples provide an efficient, generalized connection filter. Both examples parse the rules and sets up a rule-matching algorithm so that connection filtering adds minimal overhead to a WebLogic Server connection. If necessary, you can modify this code and reuse it. You may, for example, want to accommodate the local or remote port number in your filter or a more site-specific algorithm that will reduce filtering overhead. For instructions on how to build, configure, and run these samples, see the `package.html` file in the *`SAMPLES_HOME`*`\server\src\examples\security\net` directory provided by WebLogic Server.

SimpleConnectionFilter Example

The `examples.security.net.SimpleConnectionFilter` example is included in the *`SAMPLES_HOME`*`\server\src\examples\security\net` directory provided by WebLogic Server. This example implements the `ConnectionFilter` interface and filters connections using rules that you define in the `filter` file.

SimpleConnectionFilter2 Example

The `examples.security.net.SimpleConnectionFilter2` example is included in the *`SAMPLES_HOME`*`\server\src\examples\security\net` directory provided by WebLogic Server. This example implements the `ConnectionFilter` and `ConnectionFilterRulesListener` interfaces and filters connections using the rules that you define on the Administration Console.

Example of the Accept Method Used in Filtering Network Connections

In Listing 5-1, WebLogic Server calls the `SimpleConnectionFilter.accept()` method with a `ConnectionEvent`. The `SimpleConnectionFilter.accept()` method gets the remote address and protocol and converts the protocol to a bitmask to avoid string comparisons in rule-matching. Then the `SimpleConnectionFilter.accept()` method compares the remote address and protocol against each rule until it finds a match.

This code fragment is taken from the `SimpleConnectionFilter.java` file in the *`SAMPLES_HOME`*`\server\src\examples\security\net` directory.

Listing 5-1 Example of Filtering Network Connections

```
public void accept(ConnectionEvent evt)
  throws FilterException
{
  InetAddress remoteAddress = evt.getRemoteAddress();
  String protocol = evt.getProtocol().toLowerCase();
  int bit = protocolToMaskBit(protocol);

  // this special bitmask indicates that the
  // connection does not use one of the recognized
  // protocols
  if (bit == 0xdeadbeef)
  {
    bit = 0;
  }

  // Check rules in the order in which they were written.

  for (int i = 0; i < rules.length; i++)
  {
    switch (rules[i].check(remoteAddress, bit))
    {
```

```
      case FilterEntry.ALLOW:
return;
      case FilterEntry.DENY:
throw new FilterException("rule " + (i + 1));
      case FilterEntry.IGNORE:
break;
      default:
throw new RuntimeException("connection filter internal error!");
      }
    }

    // If no rule matched, we allow the connection to succeed.

    return;
  }
```

Using J2EE Sandbox Security to Protect Application Server Resources

WebLogic Server supports the J2EE requirements for sandbox security for Web, EJB and connector components. Furthermore, WebLogic Server extends the connector model of specifying additional policies in the deployment descriptor to the Web and EJB components.

J2EE has requirements for the Java 2 sandbox of different application types (see the J2EE 1.3 spec, section 6.2.2) as does the Connector 1.0 spec (see section 11.2).

Furthermore, the J2EE specification suggests that the deployer be able to add to these policies. For J2EE, this is done through comments in the deployment descriptor but the specification states: “A future version of this specification will allow these security requirements to be specified in the deployment descriptor for the application components.” The connector specification already provides for deployment descriptors to specify additional policies using the security-permission tag as in the following example:

```
<security-permission >
<description> Optional explanation goes here </description>
<security-permission-spec>
<!—
```

A single grant statement following the syntax of `http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSyntax` without the "codebase" and "signedBy" clauses goes here. For example:

```
-->
grant {
permission java.net.SocketPermission "*", "resolve";
};
</security-permission-spec>
</security-permission>
```

WebLogic Server meets these requirements and in addition adds the `security-permission` tag to the `weblogic.xml` and `weblogic-ejb-jar.xml` files. This both extends the connector model to the two other application types providing a uniform interface to policies across all component types as well as anticipates future J2EE specification changes.

Each of the containers (web, EJB, connector) must register applications as they are deployed and undeployed. Upon a server reboot, each deployed application must be re-registered. To register them, the containers should call one of the following methods from `weblogic.security.service.SupplementPolicyObject`:

```
/**
* Set extra permissions for the specified URL. If there are
* already extra permissions set (by a previous setPolicies*()),
* calling this will override them.
*
* @param url URL to set permissions for
* @param grantStatement text of grant statement for permissions
* to add to the URL
*/
static public void setPolicies (URL url,
                                String grantStatement,
                                String defaultType);
/**
* Set extra permissions for the specified file. If there are
* already extra permissions set (by a previous setPolicies*()),
* calling this will override them.
*
* @param filename file to set permissions for
* @param grantStatement text of grant statement for permissions
* to add to the file
*/
static public void setPolicies (String filename,
                                String grantStatement,
                                 String defaultType);
```

The grantStatement should be all of the text within the security-permission-spec tags. The defaultType should be one of EJB_COMPONENT, WEB_COMPONENT, or CONNECTOR_COMPONENT defined in the same class.

When an application is undeployed, the containers should call the `removePolicies()` method in the same class:

```
static public void removePolicies(String filename);
static public void removePolicies(URL url);
```

CHAPTER

A Deprecated Security APIs

Some or all of the Security interfaces, classes, and exceptions in the following WebLogic security packages were deprecated in this release of WebLogic Server:

- weblogic.security
- weblogic.security.acl
- weblogic.security.audit
- weblogic.security.auth
- weblogic.security.net
- weblogic.security.SSL

For specific information on the interfaces, classes, and exceptions deprecated in each package, see the WebLogic Javadoc for each package in the WebLogic Server 8.1 online documentation at edocs.bea.com.

Index

A

API methods
- accept() 5-8
- doAs() 3-4
- getAttribute() 3-45
- getCallerPrincipal() 4-5
- getSubject() 3-4, 3-19
- isCallerInRole() 4-5
- isUserInRole() 2-20
- login() 3-6
- removePolicies() 5-12
- runAs() 3-4, 3-6, 3-19
- setPolicies() 5-11
- setTrustManager() 3-49

C

CallbackHandler 3-5
classes
- HttpsURLConnection 3-51
- javax.security.PrivilegedAction 3-6
- JNDI Environment 3-44
- PrivilegedAction 3-19
- SSLContext 3-50
- weblogic.security.Security 3-4

code examples
- accept() method 5-9
- CallbackHandler interface 3-14
- ConnectionFilter interface 5-7
- ConnectionFilterRulesListener interface 5-7
- getSubject() method 3-19
- Host Name Verifier custom version 3-47
- Java Client That Uses JAAS Authentication 3-22
- Login() method 3-18
- LoginContext 3-17
- network connection filter 5-4, 5-8
- PrivilegedAction 3-20
- runAs() method 3-19
- Sample_jaas.config 3-17
- SimpleConnectionFilter 5-8
- SimpleConnectionFilter2 5-9
- SSLClient 3-29
- SSLClientServlet 3-39
- SSLServerSocketFactory 3-50
- SSLSocketClient 3-35, 3-49
- Trust Manager 3-49
- URL Outbound SSL Connection 3-51

configuration file 3-5

D

DCOM 5-5
declarative security
- implementing in EJBs 4-1
- implementing in Web applications 2-19

deployment descriptor files
- ejb-jar.xml 4-1
- web.xml 2-19
- weblogic.xml 2-19, 4-1

deployment descriptors

using to implement security in EJBs 4-1
using to implement security in Web applications 2-19
digital certificates 3-45, 3-46
chain of 3-48
contents of 3-45
demonstration version 3-46
invalid 3-48
overriding validation errors 3-48

E

e-commerce 1-4
EJBContext
isCallerInRole() 4-5
EJBContext.getCallerPrincipal() 4-5
ejb-jar.xml
using to implement security in EJBs 4-1
EJBs
implementing declarative security 4-1
encryption
services 1-4
exceptions
LoginException 3-6

F

FTP 5-5

G

GIOP 5-5
GIOPS 5-5
global-role tag 2-17, 2-18, 4-3, 4-4

H

host name verification
impact of turning off 3-46
ways to turn off 3-46
when to turn off 3-46
Host Name Verifier
code example 3-47
custom version 3-46
how to use a custom version 3-47
requirements for using a custom version 3-47
HTTP 5-5, 5-8
HTTPS 5-5, 5-8
HttpServletRequest.isUserInRole(String role) 2-20
HttpsURLConnection 3-51

I

IIOP 5-8
IIOPS 5-8
implementing programmatic security
in EJBs 4-5
in Web applications 2-20
interface
ConnectionFilterRulesListener 5-2
interfaces
CallbackHandler 3-4, 3-13
ConnectionFilter 5-2
ConnectionFilterRulesListener 5-7
HostnameVerifier 3-47
HostnameVerifierJSSE 3-47
java.security.Principal 3-4
TrustManager 3-48
TrustManagerJSSE 3-48
IP address 5-8

J

J2EE 5-10
J2SE 3-4
JAAS 3-3
Authentication Tutorial 3-5
CallbackHandler interface 3-5
compiling applications
additional system classpaths required when 3-3, 3-6

getSubject() method 3-4
LoginContext 3-3
LoginModule 3-4
Principal 3-4
programming steps 3-4
Subject 3-4
version supported 3-3
jar files
wlManagement.jar 3-3, 3-6
wlSecurityProviders.jar 3-3, 3-6
Java 2 Platform, Standard Edition
See J2SE
Java Secure Socket Extension
See JSSE
Java Security Packages and Classes 1-5
java.security.Principal interface 3-4
javax.security.auth.Subject
doAs() method 3-4
javax.security.PrivilegedAction class 3-19
javax.servlet.request.cipher_suite 3-45
javax.servlet.request.key_size 3-45
javax.servlet.request.X509Certificate 3-45
JNDI 3-3
JNDI parameters
setProviderURL 3-44
setSSLClientCertificate 3-44
setSSLRootCAFingerprint 3-45
setSSLServerName 3-44
JSSE 3-50, 3-51

L

login() method 3-6
LoginContext 3-5
login() method 3-6
LoginException 3-6

M

man-in-the-middle attacks 3-46
message URL http
//java.sun.com/products/ejb/docs.html 4-2
mutual authentication 3-40

N

network connection filter
evaluation of rules 5-5
fast rules 5-5
flat file example 5-7
rules syntax 5-4
slow rules 5-5
network connection filters 5-1
guidelines for writing 5-3
NulledTrustManager 3-49

P

principal-name tag 2-17, 4-3
PrivilegedAction 3-4
PrivilegedExceptionAction 3-4
programmatic security
implementing in EJBs 4-5
implementing in Web applications 2-20

R

response to invalid certificates
as specified by the SSL specification 3-48
response to invalid digital certificates
by Web browsers 3-48

S

sample_jaas.config 3-5
sandbox security 5-10
security
APIs 1-5
applying programmatically in servlet 2-21
packages

deprecation details A-1
list of A-1
Security policies 5-1
security-permission tag 5-11
security-permission-spec tags 5-12
Service Pack 1-4
servlet container 2-19
SSL
handshake 3-48
SSLContext 3-50
setting up 3-50
Subject 3-6
subject's public key 3-45
SubjectDN 3-46

T

T3 5-5, 5-8
T3S 5-5, 5-8
Trust Manager
associating with SSLContext 3-50
benefits of 3-48
creating 3-48
performing custom checks 3-48
trusted certificate authority 3-45
TrustManagerJSSE interface
uses of 3-48

U

UsernamePasswordLoginModule 3-13

V

validation errors
overriding 3-48

W

Web application 2-19
Web browsers
response to invalid digital certificate 3-48
web.xml
using to implement security in Web applications 2-19
WebLogic security
deprecated packages and classes 1-5
packages 1-8
packages that implement JSSE 3-2
WebLogic Server
container support for JAAS 3-3
SSL implementation 3-2
support for JAAS 3-3
weblogic.jar 3-7
weblogic.security.Security.runAs() method 3-6, 3-19
weblogic.security.service.SupplementPolicy Object 5-11
weblogic.servlet.request.SSLSession 3-45
weblogic.xml 5-11
using to implement security in EJBs 4-1
using to implement security in Web applications 2-19
WebLogic's JSSE implementation 3-2
weblogic-ejb-jar.xml 5-11
wlManagement.jar 3-3, 3-6
wlSecurityProviders.jar 3-3, 3-6

X

X509Certificate 3-45