BEA WebLogic Server

Programming WebLogic Security

BEA WebLogic Server 6.1
Document Date: November 17, 2003

Programming WebLogic Security

Document Edition	Document Date	Software Version
N/A	November 17, 2003	BEA WebLogic Server Version 6.1

Contents

About This Document

This document introduces concepts associated with the BEA WebLogic Server™ security features, explains how to use those features to make your WebLogic Server deployment secure, and provides a guide to application programming interfaces (APIs) in the WebLogic Security Service Provider Interface (SPI).

This document is organized as follows:

- Chapter 1, "Introduction to WebLogic Security," presents an overview of the features in WebLogic Security.
- Chapter 2, "Security Fundamentals," presents a detailed discussion of the concepts associated with the security features in WebLogic Server.
- Chapter 3, "Securing a WebLogic Server Deployment," describes how to secure your WebLogic Server deployment and how to thwart common security attacks.
- Chapter 4, "Programming with the WebLogic Security SPI," describes how to define a security policy for WebLogic Server and how to connect to WebLogic Server in a secure manner.

Audience

This document is intended for programmers who want to incorporate security into their WebLogic Server deployment.

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

The BEA corporate Web site provides all documentation for 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 to WebLogic Security

The following sections provide an introduction to WebLogic Security including:

- WebLogic Security Features
- WebLogic Security Architecture
- Using WebLogic Server as a Client to BEA Tuxedo

WebLogic Security Features

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 there is an increased probability that security attacks can be perpetrated. 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.

The security features provided in WebLogic Server let you establish secure connections from Web browsers, Java clients, and other WebLogic Servers to WebLogic Server. In addition, WebLogic Server can be used as a client to BEA Tuxedo over a secure connection.

Specifically, WebLogic Server provides the following security features:

- Security realms which represent a logical grouping of Users, Groups, ACLs, and permissions for the purpose of protecting WebLogic Server resources. You can use the default security realm or one of a set of alternative security realms that allow you to use Windows NT, UNIX, and LDAP security stores. In addition, custom developed security realms are supported.
- Authentication of clients requesting access to WebLogic Server resources. Authentication can be accomplished using a username/password combination or digital certificates where a client is authenticated using the identity inside of the X.509 digital certificate provided to WebLogic Server as part of a Secure Sockets Layer (SSL) connection.
- Authorization of Users and Groups through access control lists (ACLs).
- The Java Authentication and Authorization Service (JAAS) application programming interface (API) for authentication. The JAAS implementation in WebLogic Server provides LoginContext authentication and Subject authorization. Support for JAAS authorization is not provided.
- Data integrity and confidentiality through the SSL protocol. Clients can establish SSL sessions with WebLogic Server using the Hypertext Transfer Protocol (HTTP), the BEA proprietary T3 protocol, or the Remote Method Invocation (RMI) over Internet Inter-ORB (IIOP) protocol.
- Auditing of events such as failed login attempts, authentication requests, rejected digital certificates and invalid ACLs.
- Filtering of client connections for the purpose of accepting or rejecting the client request based the origin (host name or network address) or protocol of the client.
- Propagation of the security context from WebLogic Server security realms to BEA Tuxedo domains using WebLogic Enterprise Connectivity (WLEC). This feature allows the propagation of security information about the requesting WebLogic Server User to the BEA Tuxedo domain over network connections that are part of a trusted WLEC connection pool.

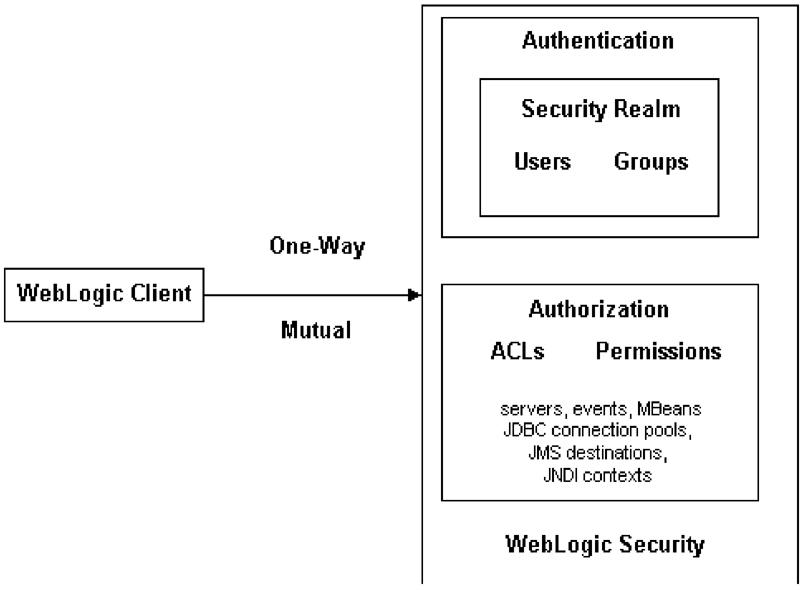
For information about security in WebLogic EJBs, see Programming WebLogic Enterprise JavaBeans.

For information about security in Web applications, see Assembling and Configuring Web Applications.

WebLogic Security Architecture

The security architecture in WebLogic Server is based on the authorization and authentication of Users. Figure 1-1 illustrates the security architecture in WebLogic Server.

Figure 1-1 WebLogic Server Security Architecture



Authentication is the first layer of security in the WebLogic Server environment. Authentication is the process of verifying an entity's identity before completing a connection. Authentication protects who gets access to the WebLogic Server environment.

The default authentication scheme for WebLogic Server is one-way authentication. One-way authentication is common on the Internet where customers want to create secure connections before they share personal data. When WebLogic Server receives a client request, WebLogic Server authenticates the client by comparing the supplied username and password against the usernames and passwords defined in the WebLogic Server security realm. If the username and password can be validated, the client is granted access to the WebLogic Server environment.

If the SSL protocol is used, the establishment of the secure connection requires WebLogic Server to present its digital certificate chain to the client to prove its identity. The client uses a set of digital certificates for certificate authorities it trusts to verify the authenticity of the digital certificate presented by WebLogic Server. When the SSL protocol on a WebLogic Server is configured for two-way SSL, the client is also required to pass a chain of digital certificates that validates its identity.

Once in the WebLogic Server environment, authorization protects who has access to the available resources. Authorization is based on the definition of Users and Groups and the permissions assigned to the resources in WebLogic Server. A resource can be an event, a servlet, JDBC connection pools, passwords, JMS destinations, and JNDI contexts. WebLogic Server uses security realms to logically organize Users, Groups, ACLs, and the permissions for the resources. A WebLogic Server resource is protected under only one security realm by an ACL in that security realm. A User must be defined in a security realm in order to access any resources belonging to that security realm.

When a User attempts to execute a method on a resource, the following steps are taken to determine whether access is permitted:

1. If the resource is protected and the User has not previously been authenticated, the User is requested to authenticate. If the authentication fails, the request is rejected.
2. WebLogic Server identifies the User invoking the method. If the user cannot be determined, the request is rejected.
3. WebLogic Server determines the set of required permissions to invoke the method on the WebLogic Server resource.
4. If the invoking User has at least one of the required permission, WebLogic Server allows the method to be invoked.

The following sections describe how WebLogic Server provides security for different types of connections.

Connections with Web Browsers

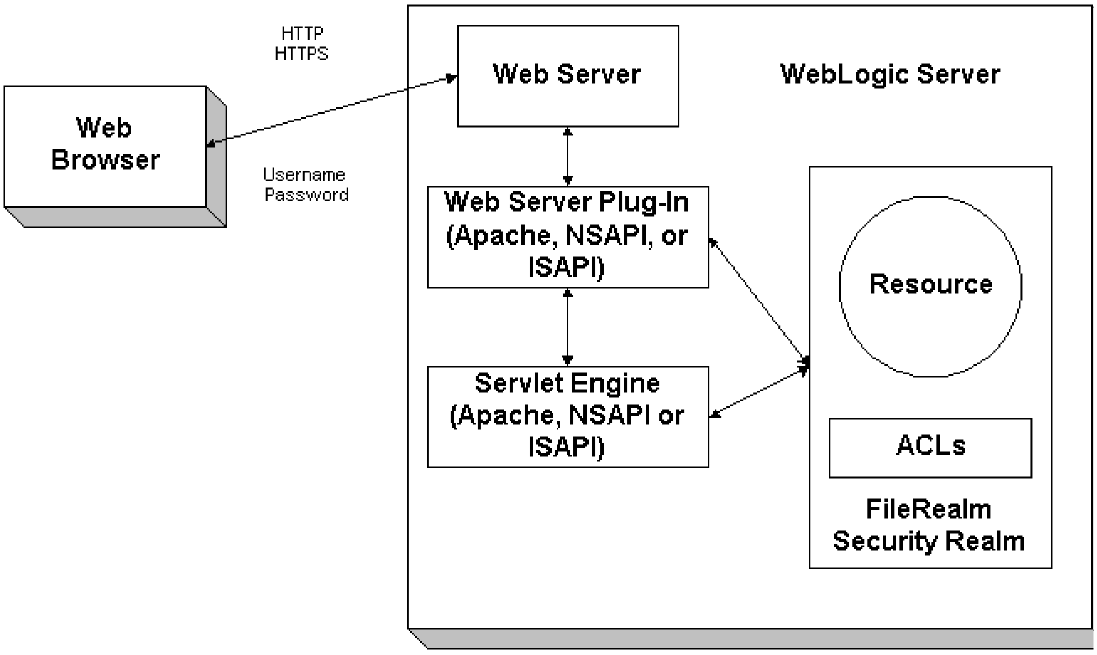
Web browsers interact securely with WebLogic Server in the following manner:

1. A user invokes a resource in WebLogic Server by entering the URL for that resource in a Web browser.
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 ACL. If the WebLogic Server resource is protected, the Web server uses the established HTTP connection to request a user ID and password from the user.
4. When the user's Web browser receives the request from WebLogic Server, it prompts the user for a user ID and password.
5. The Web browser sends the request again, along with the user ID 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 (user ID 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. If authorization succeeds, the servlet engine fulfills the request. The servlet engine resides within WebLogic Server.
9. Before invoking a method on the servlet, the servlet engine performs a security check. During this check, the servlet engine extracts the User's credentials from the security context and verifies that the User is authorized to invoke the method on the servlet.

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

Figure 1-2 Secure Login for Web Browsers



The HTTPS protocol provides an additional level of security to this usage scenario. Because the SSL protocol encrypts the data transmitted between the Web browser and WebLogic Server, the user ID and password do not flow in clear text. Therefore, WebLogic Server can authenticate a user while protecting that User's password by using the SSL protocol.

For more information, see the following sections:

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

Connections with Servlets, JSPs, EJBs, RMI Objects and Java Applications

Servlets, JSPs, EJBs, RMI objects, and Java applications use the Java Authentication and Authorization Service (JAAS) to authenticate WebLogic Server. JAAS is a standard extension to the Java 2 Software Development Kit. The authentication component of JAAS provides the ability to reliably and securely maintain client identity, regardless of whether the code is running as a Java application, a JSP, an EJB, an RMI object or a servlet. In WebLogic Server, JAAS is layered over the existing Security Service Provider Interface (SPI) allowing the continued use of realm-based authorization. This is necessary because WebLogic Server does not provide the authorization component of JAAS. All authorization checking is done through the underlying security realm.

When using JAAS authentication, Java clients enable the authentication process by instantiating a LoginContext object which in turns references a Configuration object. The Configuration object specifies the configured LoginModules to be used for client authentication. The LoginModule object prompts for and verifies the client credentials. It is important to understand that you need to write a LoginModule object for each type of authentication mechanism you want to use with WebLogic Server. For example, if you want to use mutual authentication, you need to provide a LoginModule object that both requests and provides credentials.WebLogic Server does not supply any LoginModule objects.

Clients (servlets, JSPs, EJBs, RMI objects, and Java applications) interact securely with WebLogic Server in the following manner:

1. The client creates a LoginContext which instantiates a LoginModule object and a CallbackHandler object.
   - The LoginContext references a Configuration object which specifies the LoginModules and their order of execution.
   - The CallbackHandler object gathers input from users (such as a password or the name of a digital certificate file) and passes it in turn to the LoginModules.
2. The client invokes the `login()` method of the LoginContext object. The `login()` method then invokes the specified LoginModules.
3. The LoginModule prompts for and verifies the client credentials.

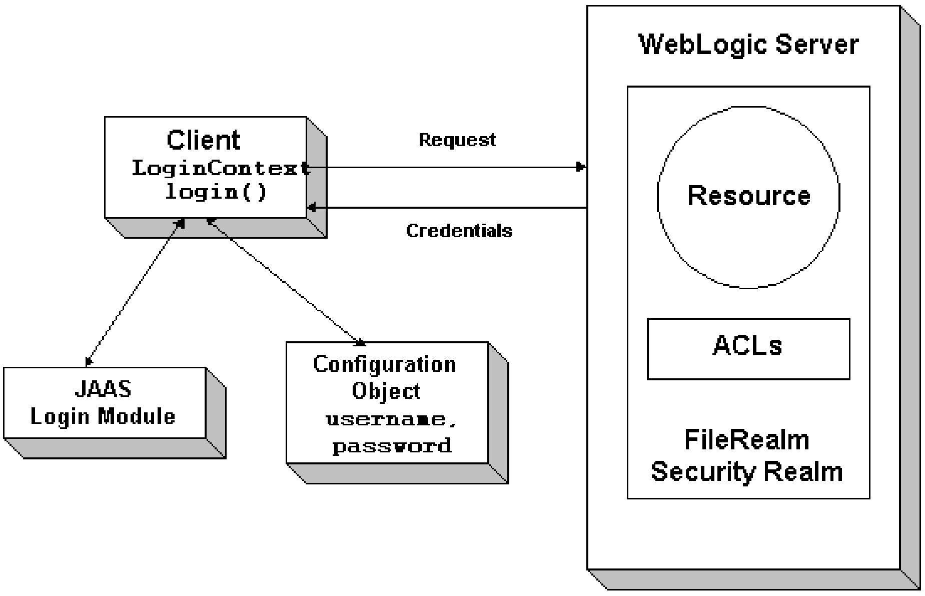
4. Upon successful authentication, WebLogic Server determines whether the client has the permission required to access the requested resource. The permission is determined by the ACL for the resource as defined in the WebLogic Server security realm.

5. Upon successful ACL authorization, the client request from the client is fulfilled by WebLogic Server.

When using a LoginModule that implements password authentication, you can configure WebLogic Server to use the SSL protocol. 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.

Figure 1-3 illustrates the secure login process for servlets, JSPs, EJBs, RMI objects, and Java applications.

Figure 1-3 Secure Login for Java Clients



WebLogic Server can function as a client to another WebLogic Server. In this scenario, WebLogic Server has the same authentication options as a client.

Note: WebLogic Server still supports the JNDI method of passing authentication. However, this functionality is being replaced with JAAS authentication.

For more information, see the following sections:

- Programming with the WebLogic Security SPI
- Managing Security

Connections with Administration Servers

In WebLogic Server, an Administration Server is a WebLogic Server that functions as the central source of all configuration information. An Administration Server may contain configuration information for one WebLogic Server or a cluster of WebLogic Servers. It is important to protect the connection between the Administration Server and the other WebLogic Servers in your environment from eavesdropping, tampering, replay, and impersonation attacks.

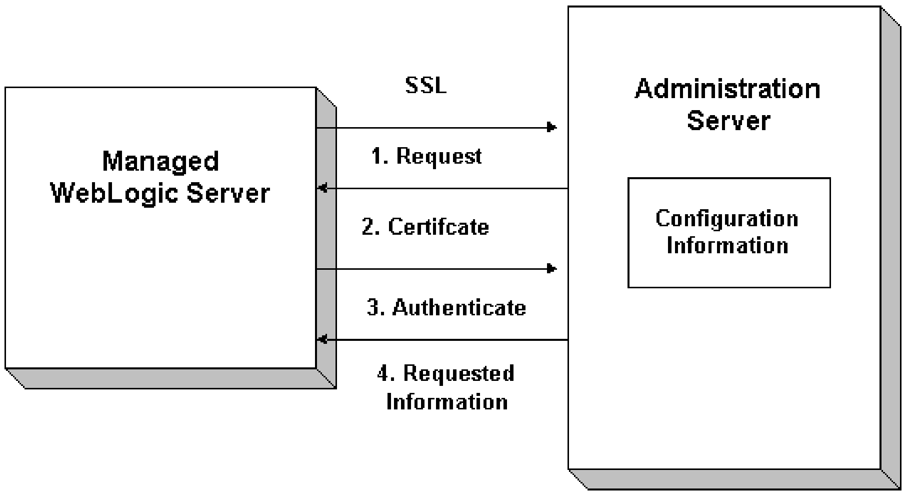
When the SSL protocol and certificate authentication are used, the Administration Server presents its digital certificate to the managed WebLogic Server whenever the managed WebLogic Server is started. The managed WebLogic Server then authenticates the Administration Server, using the information in the digital certificate.

Digital certificates for Administration Servers are provided by BEA. They are installed during the installation of WebLogic Server in `\wlserver6.1\config\mydomain`.

By default, the connection between the Administration Server and other WebLogic Servers is not secure. The file containing usernames and passwords is not encrypted. Usernames and passwords are sent in clear text over the connection, leaving configuration information unprotected. For this reason, we recommend using the SSL protocol and certificate authentication to protect the configuration information in the Administration Server.

Figure 1-4 illustrates the secure login process between Administration Servers and managed WebLogic Servers.

Figure 1-4 Secure Login for Managed and Administration Servers



For more information, see Managing Security.

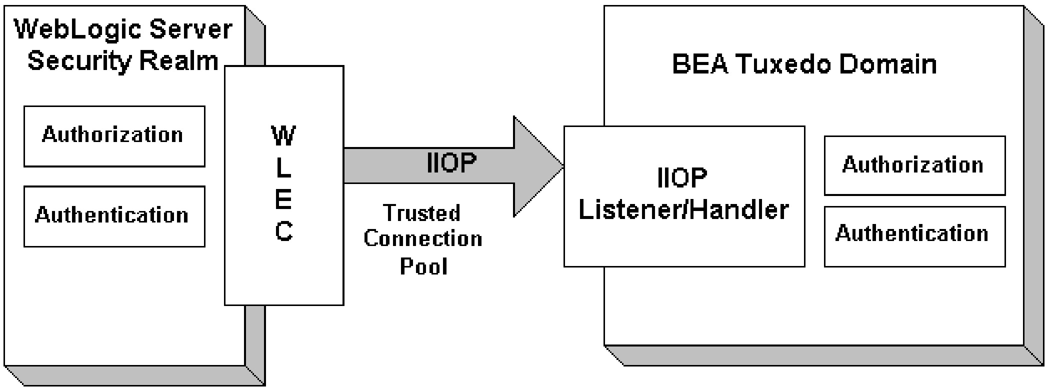
Using WebLogic Server as a Client to BEA Tuxedo

The scope of security in a WebLogic Server security realm differs from that in a BEA Tuxedo domain. Each contains it own security store of Users and its own access control. However, by using WebLogic Enterprise Connectivity, the identity of a User authenticated in a WebLogic Server security realm can be used to form the identity of an authenticated principal in a BEA Tuxedo domain over a connection that is part of a trusted WLEC connection pool. This functionality is referred to as *security context propagation*.

Note: The propagation of security context in the WebLogic Server product is unidirectional. It allows you to propagate a User's identity in only one direction: from a WebLogic Server security realm to a BEA Tuxedo domain.

Figure 1-5 illustrates how the propagation of the security context between the WebLogic Server and the BEA Tuxedo environments works.

Figure 1-5 Propagation of the Security Context Between WebLogic Server and BEA Tuxedo



When propagating the security context, the security identity of a WebLogic Server User is included as part of the service context of an IIOP request. This request is sent to a BEA Tuxedo domain over a network connection that is part of a pool of WLEC connections. Each network connection in a WLEC connection pool is authenticated using a defined User identity. Both password and certificate authentication can be used to establish a WLEC connection pool.

The propagated security identity is used by the IIOP Listener/Handler to impersonate a User identity in the BEA Tuxedo domain. The impersonated identity is represented as a pair of tokens: one for authorization and one for auditing. These tokens are propagated to the target CORBA object in the BEA Tuxedo domain, where they are used for authorization and auditing.

To facilitate the mapping of User identities, the IIOP Listener/Handler in BEA Tuxedo uses an authentication plug-in. This plug-in is responsible for mapping the User identity into the authorization and auditing tokens. These tokens are propagated, in turn, as part of the request being forwarded to the target CORBA object. The target CORBA object can then use the tokens to determine information about the initiator of the request, including the identity of the User and the Role or Group name with which the User is associated.

The SSL protocol is used to protect the confidentiality and integrity of the request sent from the WebLogic Server security realm. SSL encryption is provided for IIOP requests sent to CORBA objects in the BEA Tuxedo domain. To protect a request, both WebLogic Connectivity and the BEA Tuxedo CORBA application must be configured to use the SSL protocol.

For more information, see the following sections:

- "Configuring Security Context Propagation" in Managing Security
- Using WebLogic Connectivity

CHAPTER

2 Security Fundamentals

This section describes the following concepts behind WebLogic Server security:

- Resources
- Security Realms
- Users
- Groups
- ACLs and Permissions
- SSL Protocol
- Authentication Mechanisms
- Digital Certificates
- Certificate Authority
- Supported Public Key Algorithms
- Supported Symmetric Key Algorithms
- Supported Message Digest Algorithms
- Supported Cipher Suites

Resources

Resources are entities that are accessible from WebLogic Server, such as events, servlets, JDBC connection pools, JMS destinations, JNDI contexts, connections, sockets, files, and enterprise applications and resources, such as databases.

For each resource you want to protect in WebLogic Server, you must specify the following:

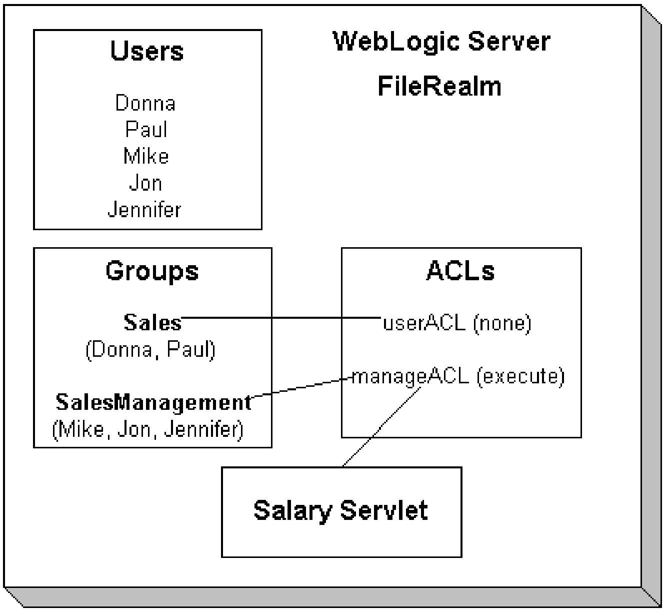
- An ACL defining who may access the resource
- The security realm to which the resource belongs
- An authentication mechanism that can verify users who request access to the resource.

For more information, see Managing Security.

Security Realms

A security realm is a logical grouping of Users, Groups, and ACLs. A WebLogic Server resource is protected under only one security realm and by a single ACL in that security realm. A User must be defined in a security realm in order to access any resources belonging to that realm. When a User attempts to access a particular WebLogic Server resource, WebLogic Server tries to authenticate and authorize the User by checking the ACL and permissions assigned the User in the relevant realm. Figure 2-1 illustrates how realms work in WebLogic Server.

Figure 2-1 Users, Groups, and ACLs in a WebLogic Server Realm



The default security realm in WebLogic Server is the File realm. When WebLogic Server is started, the File realm creates User, Group, and ACL objects from properties defined through the Administration Console in WebLogic Server and stored in the `fileRealm.properties` file.

Note: The File realm is designed for use with 10,000 or fewer users. If you have more that 10,000 users, we recommend using an alternate security realm.

WebLogic Server also provides support for developers who want or must accommodate special security situations. WebLogic Server allows you to use a security realm other than the File realm by installing an alternate security realm or by writing your own customized security realm. Alternate security realms can support some or all of the authentication and authorization operations WebLogic Server requires of a realm.

There are two possible realm configurations:

- Only a File realm.
- A Caching realm with an alternate security realm or a custom security realm.

In this configuration, the alternate security realm or the custom security realm functions as the primary realm. The File realm is the backup realm. The primary and backup realms in WebLogic Server work together to fulfill client requests with the proper authentication and authorization. For example, if you choose to use an alternate security realm such as the LDAP security realm, WebLogic Server first looks in that realm for a User. If the Users is not in the primary realm (in this case the LDAP security realm), WebLogic Server then looks for the User in the backup realm.

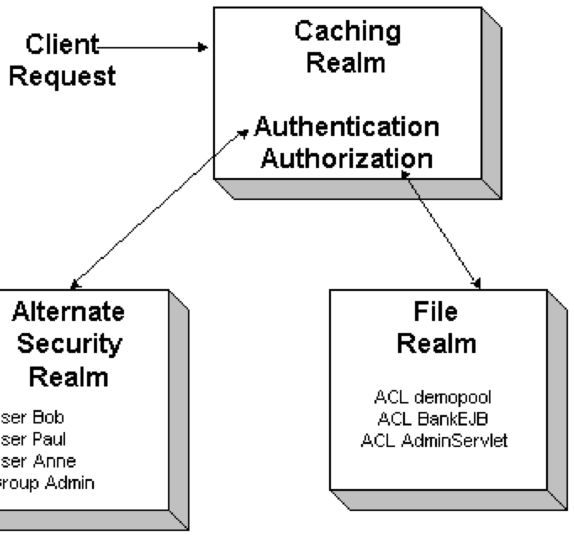
Note: If you choose to use an alternate security realm or a custom security realm, you must configure the Caching realm.

In the default scenario, a client request arrives at the WebLogic Server through the Caching realm. The Caching realm forwards the request to the File realm for authorization and authentication. When it receives the results of the realm lookups (whether successful or unsuccessful), the Caching realm stores these results. It maintains separate caches for User, Group, permission, ACL, and authentication look ups. You can selectively enable each type of cache, set the number of objects cached, and specify the number of seconds a cached object is valid. Effectively, the Caching realm makes the authentication and authorization process faster and more efficient.

If you use an alternate security realm or a custom security realm, the Caching realm evaluates the client request, delegates it to the appropriate security realm, and caches the results to make the next lookup faster. For example, you may be using an alternate security realm that supports only authentication operations. When a client request is received by WebLogic Server, the Caching realm contacts the alternate security realm for authentication, and then the File realm for authorization.

Figure 2-2 illustrates how alternate security realms, the Caching realm, and the File realm work together to authenticate and authorize users in a WebLogic Server environment.

Figure 2-2 Alternate Security Realms, the Caching Realm, and the File Realm in WebLogic Server



WebLogic Server provides the following alternate security realms:

- LDAP Security Realm

  Provides authentication through a Lightweight Directory Access Protocol (LDAP) server which allows you to manage Users in one place, the LDAP directory. When the LDAP security realm is used, the LDAP server authenticates Users and Groups. If you are using the SSL protocol with WebLogic Server, the LDAP Security Realm retrieves a User's common name from its digital certificate and searches the LDAP directory for that name. The LDAP Security Realm does not verify the digital certificate, that verification is performed by the SSL protocol.

  The LDAP Security Realm currently supports Open LDAP, Netscape iPlanet, Microsoft Site Server, and Novell NDS.

  Note: An updated LDAP security realm is provided in this release of WebLogic Server. This LDAP security realm provides improved performance and configurability. BEA recommends upgrading to this updated LDAP security realm to take advantage of this functionality. However, you can still use the old LDAP security realm. For more information, see Managing Security.

- Windows NT Security Realm

  Uses Windows NT account information to authenticate Users. Users and Groups defined through Windows NT can be used by WebLogic Server. You can use the Administration Console to view this realm, but you must use the facilities provided by Windows NT to defines Users and Groups.

- UNIX Security Realm

  Executes a native program, `wlauth`, to authenticate Users and Groups using UNIX login IDs and passwords. The UNIX Security realm is supported only on the Solaris and Linux platforms. The `wlauth` program uses a pluggable authentication module (PAM) that allows you to configure authentication services in a UNIX platform without altering applications that use those services. You can use the Administration Console to view this realm, but you must use the facilities provided by the UNIX platform to define Users and Groups.

- RDBMS Security Realm

  Reads Users, Groups, and ACLs from a database. The RDBMS Security realm is provided as an example of a custom realm implementation that provides authentication and authorization services for WebLogic Server.

For more information, see the following sections in Managing Security:

- “Specifying a Security Realm”
- “Configuring the Caching Realm”
- “Configuring the LDAP Security Realm”
- “Configuring the Windows NT Security Realm”
- “Configuring the UNIX Security Realm”
- “Configuring the RDBMS Security Realm”

Users

Users are entities that use WebLogic Server, such as application end users, client applications, and even other WebLogic Servers.

When a user wants to access WebLogic Server, it presents a username and a credential (either a password or a digital certificate) through programmatic means to WebLogic Server. If WebLogic Server can prove the identity of the User based on that username and credential, WebLogic Server associates the User with a thread that executes code on behalf of the User. Before the thread begins executing code, however, WebLogic Server checks pertinent ACLs to make sure the User has the required permissions to continue.

When defining a User in a WebLogic Server realm, you also define a password for that User. In the past, usernames and passwords were stored in clear text in a WebLogic Server security realm. Now WebLogic Server hashes all passwords. When WebLogic Server receives a client request, the password presented by the client is hashed and WebLogic Server compares it to the already hashed password for matching.

For more information, see the following sections in Managing Security:

- “Defining Users”
- “Protecting Passwords”

Groups

Groups are logically ordered sets of Users. Managing Groups is more efficient than managing large numbers of Users individually. For example, an administrator may specify permissions for 50 users at one time by specifying a Group. Usually, Group members have something in common. For example, a company may separate their sales staff into two Groups, Sales Representatives and Sales Managers, because staff members have different levels of access to WebLogic Server resources depending on their job descriptions.

WebLogic Server can be configured to assign Users to Groups. Each Group shares a common set of permissions that govern its member users' access to resources. You can mix Group names and User names whenever a list of Users is permitted.

A person can be defined as both an individual User and a Group member. Individual access permissions override any Group member access permissions. WebLogic Server evaluates each user by first looking for a Group, and testing whether the user is a member of the Group, and then looking for the User in the list of defined Users.

For more information, see the "Defining Groups" topic in Managing Security.

ACLs and Permissions

Permissions represent privileges required to access resources. A system administrator protects resources by creating lists of Users and Groups that have the permissions required to access those resources. Such lists are called access control lists (ACLs). ACLs and the functions for which they grant permission are resource-specific. For example, Users with the proper permissions may read, write, send, and/or receive files, load servlets, and link to libraries.

An ACL file is composed of AclEntries, each of which contains a set of permissions for a particular User or Group.

WebLogic Server uses the JavaSoft ACL standard, distributed with the Java Development Kit (JDK) 1.1, to extend the security framework of Java and make it practical for use at the enterprise level. The WebLogic Server implementation of ACLs is based on the `java.security.acl` package. Because ACLs in WebLogic Server are based on an open-standard, you are not tied to a proprietary solution.

You can set permissions on the following WebLogic Server resources:

- WebLogic Servers
- WebLogic events
- WebLogic JDBC connection pools
- WebLogic JMS destinations
- WebLogic JNDI contexts

- Weblogic MBeans

Note: Access to EJBs and Web applications is not controlled through ACLs. Rather, security elements in deployment descriptors are used to define access to a particular EJB or Web application.

For more information, see the “Defining ACLs” section in Managing Security.

For information about EJB security, see Programming WebLogic Enterprise JavaBeans.

For more information about Web application security, see Assembling and Configuring Web Applications.

SSL Protocol

The SSL protocol offers security to applications connected through a network. Specifically, the SSL protocol provides the following:

- A mechanism that the applications can use to authenticate each other’s identity.
- Encryption of the data exchanged by the applications.

When the SSL protocol is used, the target always authenticates itself to the initiator. Optionally, if the target requests it, the initiator can authenticate itself to the target. Encryption makes data transmitted over the network intelligible only to the intended recipient. An SSL connection begins with a handshake during which the applications exchange digital certificates, agree on the encryption algorithms to be used, and generate the encryption keys to be used for the remainder of the session.

The SSL protocol provides the following security features:

- Server authentication—WebLogic Server uses its digital certificate, issued by a trusted certificate authority, to authenticate to clients.
- Client Identify Verification—optionally, clients might be required to present their digital certificates to WebLogic Server. WebLogic Server then verifies that the digital certificate was issued by a trusted certificate authority and establishes the SSL connection. An SSL connection is not established if the digital certificate is not presented and verified. This type of connection is called

two-way SSL. When you use two-way SSL, the certificate presented by the client is not equivalent to a WebLogic Server User so the client must also present a username and credential (either a password or a digital certificate) to use for authentication and authorization.

- Confidentiality—all client requests and server responses are encrypted to maintain the confidentiality of data exchanged over the network.
- Data Integrity—data that flows between a client and WebLogic Server is protected from tampering by a third party.

If you are using a Web browser to communicate with WebLogic Server, you can use the Hypertext Transfer Protocol with SSL (HTTPS) to secure network communications.

The SSL protocol is tunneled over an IP-based protocol. Tunneling means that each SSL record is encapsulated and packaged with the headers needed to send the record over another protocol. The use of SSL is signified in the protocol scheme of the URL used to specify the location of WebLogic Server.

- SSL communications between Web browsers and WebLogic Server are encapsulated in HTTPS packets for transport. For example:

  ```
  https://myserver.com/mypage.html
  ```

  WebLogic Server supports HTTPS with Web browsers that support SSL version 3. The Java Virtual Machine (JVM) in WebLogic Server does not currently support the HTTPS protocol adapter. Consequently, WebLogic Server depends on the implementation of the SSL protocol in the Web browser.

- Java clients connecting to WebLogic Server with the SSL protocol tunnel over BEA's multiplexed T3 protocol. For example:

  ```
  t3s://myserver.com:7002/mypage.html
  ```

  Java clients running in WebLogic Server can establish either T3S connections to other WebLogic Servers, or HTTPS connections to other servers that support the SSL protocol, such as Web servers or secure proxy servers.

WebLogic Server is available with exportable- or domestic-strength SSL.

- Exportable SSL supports 512-bit certificates and 40-bit bulk data encryption.
- Domestic SSL also supports 768-bit and 1024-bit certificates, and 56-bit and 128-bit bulk data encryption.

The standard WebLogic Server distribution supports exportable-strength SSL only. The domestic version is available, by request only from your BEA sales representative. Since the United States Government relaxed restrictions on exporting encryption software in early 2000, the domestic version of WebLogic Server can be used in most countries.

During installation, you are prompted to choose which strength of the SSL protocol you want to use.We recommend the domestic WebLogic Server distribution because it allows stronger encryption. If you generate a Certificate Signature Request (CSR) using the exportable WebLogic Server distribution, you cannot support high-strength connections and you cannot authenticate clients that present domestic-strength certificates.

The implementation of the SSL protocol provided by WebLogic Server is a pure-Java implementation. A native library provides faster performance for some SSL operations on the Solaris, Windows NT, and IBM AIX platforms. You can request the use of these native Java libraries for the SSL protocol through the Administration Console.

For more information, see the "Configuring the SSL Protocol" section in Managing Security.

Authentication Mechanisms

WebLogic Server users must be authenticated whenever they request access to a protected WebLogic Server resource. For this reason, each user is required to provide a credential (a username/password pair or a digital certificate) to WebLogic Server. The following types of authentication mechanisms are supported by WebLogic Server:

- Password authentication—a user ID and password are requested from the user and sent to WebLogic Server in clear text. WebLogic Server checks the information and if it is trustworthy, grants access to the protected resource.

  The SSL (or HTTPS) protocol can be used to provide an additional level of security to password authentication. Because the SSL protocol encrypts the data transferred between the client and WebLogic Server, the user ID and password of the user do not flow in the clear. Therefore, WebLogic Server can authenticate the user without compromising the confidentiality of the user's ID and password.

- Certificate authentication—when an SSL or HTTPS client request is initiated, WebLogic Server responds by presenting its digital certificate to the client. The client then verifies the digital certificate and an SSL connection is established. The CertAuthenticator class then extracts data from the client's digital certificate to determine which WebLogic Server User owns the certificate and then retrieves the authenticated User from the WebLogic Server security realm.

  You can also use mutual authentication. In this case, WebLogic Server not only authenticates itself, it also requires authentication from the requesting client. Clients are required to submit digital certificates issued by a trusted certificate authority. Mutual authentication is useful when you must restrict access to trusted clients only. For example, you might restrict access by accepting only clients with digital certificates provided by you.

For more information, see the following sections in Managing Security:

- "Configuring the SSL Protocol"
- "Configuring Mutual Authentication"

Digital Certificates

Digital certificates are electronic documents used to verify the unique identities of principals and entities over networks such as the Internet. A digital certificate securely binds the identity of a user or entity, as verified by a trusted third party known as a certificate authority, to a particular public key. The combination of the public key and the private key provides a unique identity to the owner of the digital certificate.

Digital certificates allow verification of the claim that a specific public key does in fact belong to a specific user or entity. A recipient of a digital certificate can use the public key in a digital certificate to verify that a digital signature was created with the corresponding private key. If such verification is successful, this chain of reasoning provides assurance that the corresponding private key is held by the subject named in the digital certificate, and that the digital signature was created by that subject.

A digital certificate typically includes a variety of information, such as the following:

- The name of the subject (holder, owner) and other information required to confirm the unique identity the subject, such as the URL of the Web server using the digital certificate, or an individual'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 (defined by a start date and an end date)

The most widely accepted format for digital certificates is defined by the ITU-T X.509 international standard. Digital certificates can be read or written by any application complying with the X.509 standard. The public key infrastructure (PKI) in WebLogic Server recognizes digital certificates that comply with X.509 version 3, or X.509v3. We recommend obtaining digital certificates from a certificate authority such as Verisign or Entrust.

WebLogic Server supports the extensions provided by the X.509 standard, however, the `weblogic.security.X509` class does not provide accessors that provide information about which extensions are used in a particular digital certificate. To obtain that information, convert the `weblogic.security.X509` object to a `java.security.cert.X509Certificate` object. The following code example includes code to perform this conversion:

```
X509[] wlCerts=...
X509Certificate [] javaCerts = new X509Certificate[wlCerts.length];
try{
    CertificateFactory cf =
                     java.security.cert.CertificateFactory.getInstance(“X.509”);
    for(int i=0; i<wlCerts.length; i++){
       ByteArrayOutputStream bos = new ByteArrayOutputStream();
       wlcerts[i].output(bos);
       ByteArrayInputStream bis = new
    ByteArrayInputStream(bos.toByteArray());
       javaCerts[i] = (X509Certificate)cf.generateCertificate(bis);
       }
    }
```

For more information, see the “Configuring the SSL Protocol” section in Managing Security.

Certificate Authority

Digital certificates are issued by a certificate authority. Any trusted third-party organization or company that is willing to vouch for the identities of those to whom it issues digital certificates and public keys can be a certificate authority. When a certificate authority creates a digital certificate, the certificate authority signs it with its private key, to ensure that any tampering will be detected. The certificate authority then returns the signed digital certificate to the requesting subject.

The subject can verify the signature of the issuing certificate authority by using the public key of the certificate authority. The certificate authority makes its public key available by providing a digital certificate issued from a higher-level certificate authority attesting to the validity of the public key of the lower-level certificate authority. This scheme gives rise to hierarchies of certificate authorities. This hierarchy is terminated by a self-signed digital certificate known as the *root key*.

The recipient of an encrypted message can develop trust in the private key of a certificate authority recursively, if the recipient has a digital certificate containing the public key of the certificate authority signed by a superior certificate authority who the recipient already trusts. In this sense, a digital certificate is a stepping stone in digital trust. Ultimately, it is necessary to trust only the public keys of a small number of top-level certificate authorities. Through a chain of digital certificates, trust in a large number of users' digital signatures can be established.

Thus, digital signatures establish the identities of communicating entities, but a digital signature can be trusted only to the extent that the public key for verifying it can be trusted.

For more information, see the "Configuring the SSL Protocol" section in Managing Security.

Supported Public Key Algorithms

Public key (or asymmetric key) algorithms are implemented through a pair of different but mathematically related keys:

- A public key (which is distributed widely) for verifying a digital signature or transforming data into a seemingly unintelligible form
- A private key (which is always kept secret) for creating a digital signature or returning the data to its original form

The PKI in WebLogic Server also supports digital signature algorithms. Digital signature algorithms are simply public key algorithms used to generate digital signatures.

WebLogic Server supports the Rivest, Shamir, and Adelman (RSA) algorithm.

Supported Symmetric Key Algorithms

In symmetric key algorithms, the same key is used to encrypt and decrypt a message. The public key encryption system uses a symmetric key algorithm to encrypt a message sent between two communicating entities. Symmetric key encryption operates at least 1000 times faster than public key cryptography.

A block cipher is a type of symmetric key algorithm that transforms a fixed-length block of plain text (unencrypted text) data into a block of cipher text (encrypted text) data of the same length. This transformation takes place in accordance with the value of a randomly generated session key. The fixed length is called the block size.

The PKI in WebLogic Server supports the following symmetric key algorithms:

- DES-CBC (Data Encryption Standard for Cipher Block Chaining)

  DES-CBC is a 64-bit block cipher run in Cipher Block Chaining (CBC) mode. It provides 56-bit keys. (8 parity bits are stripped from the full 64-bit key.)

- Two-key triple-DES (Data Encryption Standard)

  Two-key triple-DES is a 128-bit block cipher run in Encrypt-Decrypt-Encrypt (EDE) mode. Two-key triple-DES provides two 56-bit keys (in effect, a 112-bit key).

  For some time it has been common practice to protect and transport a key for DES encryption with triple-DES, which means that the input data (in this case the single-DES key) is encrypted, decrypted, and then encrypted again (an encrypt-decrypt-encrypt process). The same key is used for the two encryption operations.

- RC4 (Rivest's Cipher 4)

  RC4 is a variable key-size block cipher with a key size range of 40 to 128 bits. It is faster than DES and can be exported with a key size of 40 bits. A 56-bit key size is allowed for foreign subsidiaries and overseas offices of United States companies. In the United States, RC4 can be used with keys of virtually unlimited length, although the WebLogic Server PKI restricts the key length to 128 bits.

WebLogic Server users cannot expand or modify this list of algorithms.

Supported Message Digest Algorithms

WebLogic Server supports the MD5 and SHA (Secure Hash Algorithm) message digest algorithms. Both MD5 and SHA are well known, one-way hash algorithms. A one-way hash algorithm takes a message and converts it into a fixed string of digits, which is referred to as a message digest or hash value.

MD5 is a high-speed, 128-bit hash; it is intended for use with 32-bit machines. SHA offers more security by using a 160-bit hash, but is slower than MD5.

Supported Cipher Suites

A cipher suite is an SSL encryption method that includes the key exchange algorithm, the symmetric encryption algorithm, and the secure hash algorithm used to protect the integrity of a communication. For example, the cipher suite called `RSA_WITH_RC4_128_MD5` uses RSA for key exchange, RC4 with a 128-bit key for bulk encryption, and MD5 for message digest.

The cyptographic modules used by WebLogic Server are FIPS 140-1 compliant.

WebLogic Server supports the cipher suites described in Table 2-1.

Table 2-1 SSL Cipher Suites Supported by WebLogic Server

Cipher Suite	Key Exchange Type	Symmetric Key Strength
`SSL_RSA_WITH_RC4_128_SHA`	RSA	128
`SSL_RSA_WITH_RC4_128_MD5`	RSA	128
`SSL_RSA_WITH_DES_CBC_SHA`	RSA	56
`SSL_RSA_EXPORT_WITH_RC4_40_MD5`	RSA	40
`SSL_RSA_EXPORT_WITH_DES_40_CBC_SHA`	RSA	40
`SSL_RSA_WITH_3DES_EDE_CBC_SHA`	RSA	112
`SSL_NULL_WITH_NULL_NULL`		
`SSL_RSA_WITH_NULL_SHA`	RSA	0
`SSL_RSA_WITH_NULL_MD5`	RSA	0

The license for WebLogic Server determines what strength (either domestic or export) of cipher suite is used to protect communications. If the cipher suite strength defined in the `config.xml` file exceeds the strength specified by the license, the server uses the strength specified by the license. For example, if you have an export strength license but you define the use of an domestic strength cipher suite in the `config.xml` file, the server rejects the domestic strength cipher suite and uses the export strength cipher suite.

CHAPTER

3 Securing a WebLogic Server Deployment

The following sections explain how to use the security features of WebLogic Server to protect your deployment:

- Why Is Security Important for WebLogic Server?
- Determine the Security Needs of Your WebLogic Server Deployment
- Secure the Machine on Which WebLogic Server Runs
- Accessing Protected Ports on UNIX
- Design Network Connections Carefully
- Manage the WebLogic Server Development and Production Environments
- Use Encryption
- Use the SSL Protocol
- Prevent Man-in-the-Middle Attacks
- Prevent Denial of Service Attacks
- Secure the HTTP Response Header
- Protect User Accounts
- Protect Application Content
- Replace HTML Special Characters in User-Supplied Data
- Use Protected EJBs to Limit Access to Business Logic

- Use ACLs
- Use the Appropriate Security Realm
- Secure Your Database
- Use Auditing
- Control Access to Multiple Domains

Why Is Security Important for WebLogic Server?

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.

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 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. As a policy, if there are any security-related issues with any BEA product, BEA will distribute an advisory and instructions with the appropriate course of action. If you are reponsible 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, there are partner products that can help you in your effort to secure the WebLogic Server production environment. For more information, see the BEA Partner's Page.

This topic makes some specific and general recommendations related to WebLogic Server.

Determine the Security Needs of Your WebLogic Server Deployment

Before securing your WebLogic Server deployment, it is important to understand the security needs of your WebLogic Server environment. To better understand the security needs, ask yourself the following questions:

- What WebLogic Server resources am I protecting?

  There are many resources in the WebLogic Server environment that can be protected including information in the database accessed by WebLogic Server, the availability of the Web site, the performance of the Web site, and the integrity of the Web site. Consider the resources you want to protect when deciding the level of security you must provide.

- From whom am I protecting the WebLogic Server resources?

  For most Web sites resources must be protected from everyone on the Internet. But should the Web site be protected from the employees on the intranet in your enterprise? Should your employees have access to all WebLogic Server resources? Should the system administrators have access to all WebLogic Server resources? Should the system administrators be able to access all data? You might consider giving access to highly confidential data or strategic resources to only a few well trusted system administrators. Perhaps it would be best to allow no system administrators access to the data or resources.

- What will happen if the protections on strategic resources fail?

  In some cases, a fault in your security scheme is easily detected and considered nothing more than an inconvenience. In other cases, a fault might cause great damage to companies or individual clients that use the Web site. Understanding the security ramification of each resource will help you to properly protect it.

As you read the following suggestions for securing your site, keep the answers to these questions in mind.

Secure the Machine on Which WebLogic Server Runs

A WebLogic Server deployment is only as secure as the security of the machine on which it is running. Therefore, it is important that you secure the physical machine, the operating system, and all other software that is installed on the host machine. The following are suggestions for securing the deployment machine, however, you should check with the manufacturer of the machine, operating system, and installed software for additional suggestions:

- Keep your hardware in a secured area to prevent unauthorized users from tampering with the deployment machine or its network connections.
- Have an expert review network services such as the e-mail program or directory service to ensure that there are no weaknesses that would permit a malicious attacker from accessing the operating system or system-level commands.
- Secure the file systems on the deployment machine, limiting directory and file access to a few, well-monitored user accounts. Some WebLogic Server configuration data (and some of your applications, including Java Server Pages (JSPs) and HTML pages guarded with access control lists (ACLs)) are stored in clear text on the file system. A user or intruder with read access to files and directories can easily defeat any security mechanisms you establish with WebLogic Server authentication and authorization schemes.
- Avoid creating multiple user accounts on deployment machines and avoid sharing file systems with other machines in the enterprise network.
- Create a Weblogic User in the operating system and use the security controls of the operating system to give this user ownership and exclusive access to all files and directories in the WebLogic Server deployment. No other user needs write-access to any files in the WebLogic Server deployment.

- Review active user accounts regularly and when personnel leave. Set a policy to expire passwords periodically. Never code passwords in client applications.

Accessing Protected Ports on UNIX

On UNIX systems, only processes that run under a privileged user account (in most cases, root) can bind to ports lower than 1024.

However, long-running processes like WebLogic Server should not run under these privileged accounts. Instead, you can do either of the following:

- For each WebLogic Server instance that needs access to privileged ports, configure the server to start under a privileged user account, bind to privileged ports, and change its user ID to a non-privileged account.

  If you use Node Manager to start the server instance, configure Node Manager to accept requests only on a secure port and only from a single, known host.

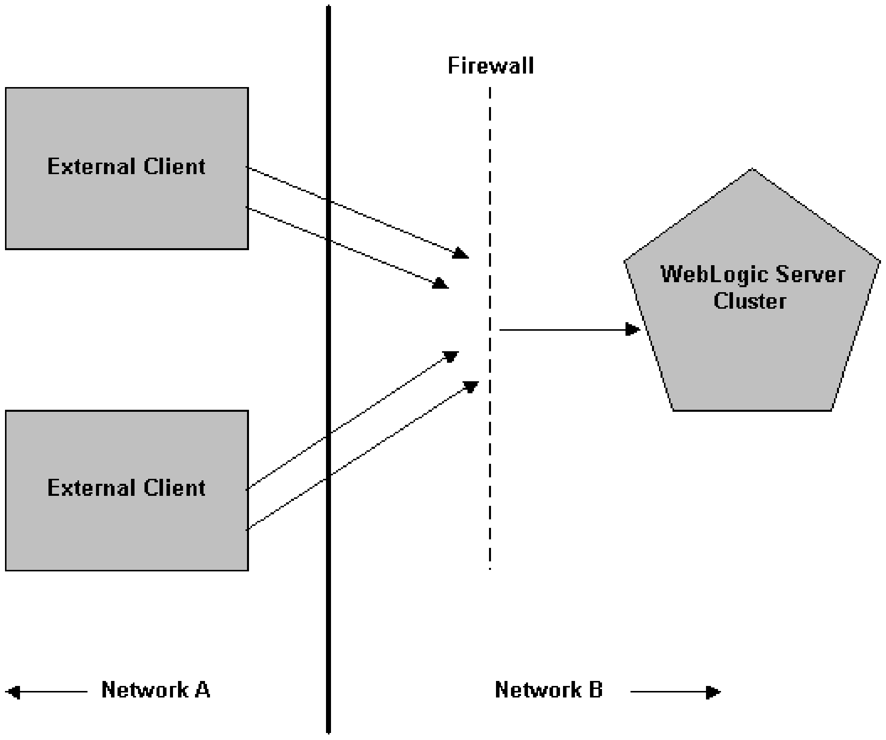
  See "Binding to Protected Ports on UNIX" in the *Administration Console Online Help*.
- Start WebLogic Server instances from a non-privileged account and configure your firewall to use Network Address Translation (NAT) software to map protected ports to unprotected ones. BEA does not provide NAT software.

Design Network Connections Carefully

When designing network connections, you want to use the easiest-to-manage solution. This choice must be weighed against the need to protect strategic WebLogic Server resources. Placing WebLogic Server resources further from potential intruders reduces the risk of a security breach. Inserting firewalls carefully in your enterprise increases security and can prevent a small security fault from turning into a security crisis. For example, it is a good idea to protect a database that contains critical data behind a firewall. In addition, protect the host machine for the database as well as the usernames and passwords for the database. Still, if someone acquires the username and passwords for the database, it is not nearly as damaging if the database is protected by a firewall and cannot receive connections from computers on the Internet.

There are many ways to combine firewalls, WebLogic Server, and other network servers. Figure 3-1 illustrates a typical setup with a firewall that filters traffic destined for a WebLogic Server cluster.

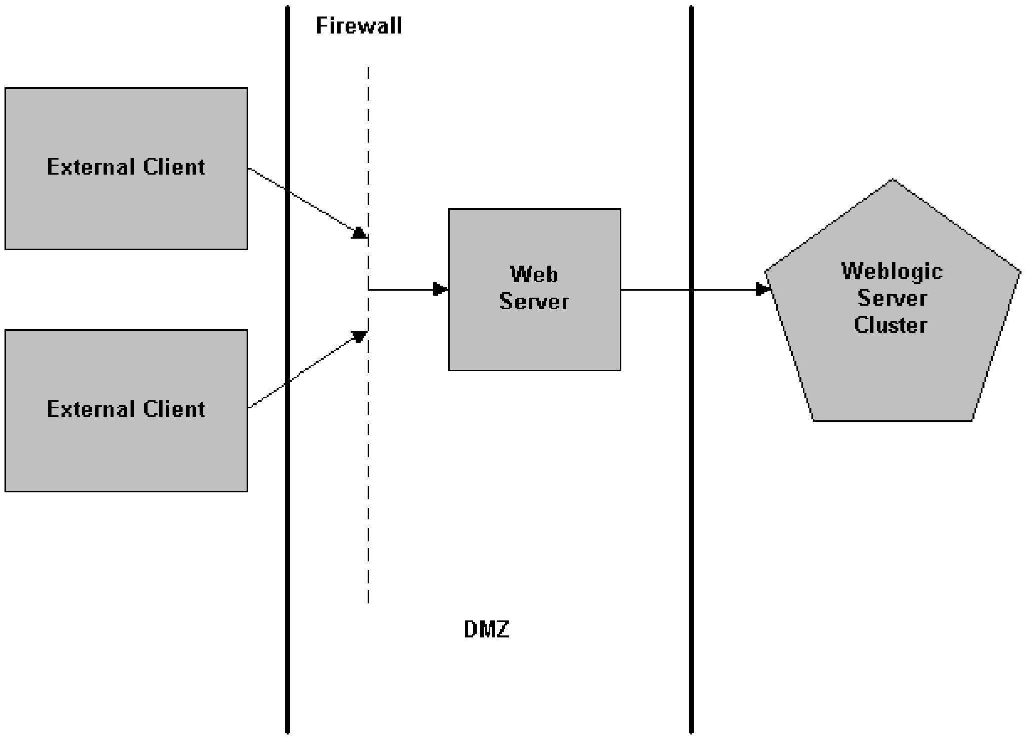
Figure 3-1 Typical Firewall Setup



Another common firewall configuration restricts access to only HTTP or HTTPS Web connections. The firewall permits clients to connect only to a Web server which usually runs at the standard HTTP port 80 or HTTPs port 443. The Web server may be a WebLogic Server or a third-party Web server set up to proxy requests to a WebLogic Server. For example, Netscape Enterprise Server, Microsoft Internet Server, and Apache Server can be set up to serve static Web pages and proxy servlet and JSP requests to WebLogic Server. Figure 3-2 illustrates this configuration.

In Figure 3-2, the Web server is a gateway operating in a demilitarized zone (DMZ). In computer networks, a DMZ is a computer host or small network inserted as a neutral zone between a company's private network and the outside public network. It prevents outside users from getting direct access to a server on which company data resides. A DMZ is an optional and more secure implementation of a firewall which can also act as a proxy server. WebLogic Server connections come only from proxied Web server requests, enhancing the security of your WebLogic Server applications and back-end resources. In the configuration shown in Figure 3-2, clients interact exclusively with the Web server and WebLogic Server connections are made only by proxied Web server requests. As a result, the security of WebLogic Server applications and back-end resources that are configured in this way are enhanced.

Figure 3-2 Firewall with Web Server Gateway



In addition to setting up a firewall, you can restrict who connects to your WebLogic Server deployment by implementing the `weblogic.security.net.ConnectionFilter` interface. This interface allows you to accept or reject a network connection based on the host name and network address of the originating machine as well as the protocol used.

Manage the WebLogic Server Development and Production Environments

For many reasons, development and production are easier when you develop on machines that closely mimic the production environment. However, security concerns suggest the following differences in the deployment and production environments:

- Do not develop on a production machine. Develop first on a development machine and then move code to the production machine when it is completed and tested. This process prevents bugs in the development environment from affecting the security of the production environment.
- Do not install the WebLogic Server sample applications on a production machine.
- The system password of a production machine should be unique within your domains and should be guarded carefully.
- Do not put the development tools on the production machine. These tools include development product components including the javac, rmic, and ejbc compilers as well as other development tools you may use. Keeping the development tools off the production machine, reduces the leverage an intruder has should they get partial access to a WebLogic Server production machine.
- Protect your source code. Getting access to your source code allows an intruder to find security holes. Always keep source code off of the production machine. Comments in JSP files that are not meant for the end user should use the JSP syntax of `<%/* ... */%>` rather than the HTML syntax of `<!-- ... -->` because the JSP comments are deleted when the JSP is compiled and therefore cannot be viewed. Also, disable the Case Sensitive Extensions field on the File tab of the Administration Console to further protect your JSP source.

- BEA does not recommend using the Servlet servlet in a production environment. You should remove all existing mappings between WebLogic servlets and the Servlet servlet from all web applications before using the applications in a production environment.
- Do not make the File servlet the default servlet in a production environment.

Use Encryption

Encryption is the process of taking text or other data and scrambling it so that it cannot be understood. Decryption reverses the process making the text or data understandable. The decryption process always requires knowledge of a secret key or password. The secret key is a long string of bits that is required as an argument to the decryption algorithm to make it work correctly. The strength of an encryption algorithm is measured by the number of bits in its key.

There are many types of encryption and each type of encryption comes in many strengths. The biggest differences between the algorithms is how much CPU time it takes to decrypt the data and how many keys there are (symmetric key algorithms have just one key that is used to both encryption and decrypt while public key algorithms have two keys, one to encryption and one to decrypt).

Encryption is typically used in places where sensitive information is stored or communicated. These places can include but are not limited to information on network machines, on disk, in a database, in memory, and in legacy systems.

There are drawbacks to using encryption:

- Encryption and decryption are computationally expensive algorithms that take CPU time to perform.
- Encryption can make debugging harder as you cannot review encrypted data to verify that it is correct.
- The loss of a secret key can render all encrypted data useless. Even the temporary loss of a secret key (for example, all the people who know the secret key are on vacation) can render a Web site useless until the secret key can be retrieved.

- Key management is an awkward problem.Who should know the secret key, where the secret key is stored, and whether the secret key itself should be encrypted are just some of the issues that must be addressed.

The questions to ask when designing the encryption for a WebLogic Server deployment are:

- What needs to be encrypted?
- What algorithm and strength should be used to encrypt data?
- Where will the keys be stored?

Use the SSL Protocol

Data that is sent over the network (either the Internet or an intranet) can be viewed by other parties on the network. This is unavoidable because of the design of networks. To prevent sensitive data from being compromised, the data should be encrypted.

To send encrypted data over the Internet you should use the HTTPS protocol (HTTP over the Secure Sockets Layer (SSL)) rather than the HTTP protocol. To configure your Web application for the SSL protocol you must use the `user-data-constraint` tag in the `web.xml` file and set the transport-guarantee to `CONFIDENTIAL`.

The demonstration digital certificates provided with WebLogic Server are for testing only. Everyone who downloads WebLogic Server has the private keys for these digital certificates. Do not use these digital certificates in a deployed WebLogic Server. Check the `filerealm.properties` file to make sure that the demonstration digital certificates have been removed from the deployed WebLogic Server.

Use the strongest encryption WebLogic Server supports: 1024-bit keys, 128-bulk data encryption on your data. The WebLogic Server version you download allows just 512-bit keys and 40-bit bulk encryption. Contact your BEA sales representative to request the stronger version.

Prevent Man-in-the-Middle Attacks

When using the SSL protocol, the data sent between the communicating parties can be vulnerable to man-in-the-middle attacks. A man-in-the-middle attack occurs when a machine inserted into the network captures, modifies, and retransmits messages to the unsuspecting parties. One way to avoid man-in-the-middle attacks is to validate that the host to which a connection is made is the intended or authorized party. An SSL client can compare the host name of the SSL server with the digital certificate of the SSL server to validate the connection. WebLogic Server provides a HostName Verifier to protect SSL connections from man-in-the-middle attacks.

By default, the HostName Verifier is enabled. However, during the implementation of WebLogic Server at your site, this functionality may have been disabled. To ensure a HostName Verifier is being used with your WebLogic Server deployment, check that the `SSL.Ignore.HostName.Verification` attribute on the SSL tab of the Servers node of the Administration Console is not checked.

Prevent Denial of Service Attacks

A Denial of Service attack leaves a Web site running but unusable. Hackers accomplish this by depleting or deleting one or more critical resources of the Web site. While a denial of service attack can happen if a hacker gets administrative privileges to your Weblogic Server, it usually occurs when an unprivileged user removes a required resource from a WebLogic Server deployment.

To perpetrate a Denial of Service attack on a WebLogic Server, an intruder bombards with many requests that are either very large in size, are slow to complete, or never complete so that the client stops sending data before completing the request. To prevent Denial of Service attacks, WebLogic Server allows you to restrict the size of a message as well as the maximum time it takes a message to arrive. You can set this information individually for each of the three protocols used by WebLogic Server: T3, HTTP and IIOP. See the online help for the Administration Console for information on setting the MaxT3MessageSize, CompleteT3MessageTimeout, MaxHTTPMessageSize, CompleteHTTPMessageTimeout, MaxIIOPMessageSize, and CompleteIIOPMessageTimeout fields. These fields have a default of 2 gigabytes for the maximum message size and 480 seconds for the complete message timeout. A value of 0 for any of the fields disables that check.

Secure the HTTP Response Header

Consider preventing WebLogic Server from sending its name and version number in HTTP responses.

By default, when an instance of WebLogic Server responds to an HTTP request, its HTTP response header includes the server's name and WebLogic Server version number. This poses a potential security risk if an attacker knows about some vulnerability in the specific version of WebLogic Server.

To prevent a WebLogic Server instance from sending its name and version number, disable the Send Server Header attribute in the Administration Console. The attribute is located on the Server → *ServerName* → Configuration → Protocols → HTTP tab.

Protect User Accounts

In a dictionary attack, a hacker sets up a script to attempt logins using passwords out of a “dictionary”. WebLogic Server provides a set of configurable attributes which protect user accounts from dictionary attacks. These attributes are configurable in a number of ways (for example, you can disable all the attributes, increase the number of invalid login attempts required before locking the account, increase the time period in which invalid login attempts have to take place before locking the account, and change the amount of time the user account is locked). It is up to site administrators to determine how these attributes should be set. Use this feature to protect accounts with maximum security. WebLogic Server ships with the maximum security enforced.

Note: If during development you reduce security by changing these attributes, remember to reset the attributes before deploying.

For more information, see Protecting Passwords.

Protect Application Content

By default, WebLogic Server uses a single directory, known as the web document root directory, as the location that contains static application content (HTML files and images) and dynamic application content (JSP and jHTML files). A potential vulnerability may occur if an application is allowed to create or modify files containing dynamic content within the web document root directory.

If an application is capable of modifying existing files in the web document root directory, there is the potential that the application could insert executable code in the form of JSP or jHTML tags in an existing file. If the particular file provides dynamic content, the inserted code would be executed the next time the file was served to a client.

To prevent the scenario under which this vulnerability could occur, BEA recommends the following supplemental security measures:

- WebLogic Server should only be installed on disks that support the ability to control access to specific directories and files (e.g., secure file system) to one or more specific user accounts. The use of an encrypted file system can be used to heighten the level of security at the cost of performance.
- A special operating system-specific user account (for example, `wls_owner`) should be established specifically to run WebLogic Server. This user account should be granted only the minimum operating system rights and privileges that are essential for successful execution of an application.
- The operating system-specific user account (`wls_owner`) should be the only user account that can access, create, or modify files in the web document root directory. This protection limits the ability of other applications executing on the same machine as WebLogic Server to access the web root directory.
- Directories containing JSP or jHTML files should be protected so that they can only be accessed or modified by the operating system-specific user account (`wls_owner`) under which WebLogic Serve is executed. Read-only access can be granted for administrative accounts such as `root` or `Administrator` for the purpose of archiving.
- The operating system-specific user account (`wls_owner`) that is used to create JSP and jHTML files should be granted only read and execute permissions to the

JSP and jHTML files. This protective measure will prevent the operating system-specific user account from knowingly writing to these files.

- Remove any unnecessary applications from the machine(s) that are used to run WebLogic Server. If it is not possible to remove an application, review the security environment under which the application executes. You need to understand which directories applications that execute with privileges (for example, under a privileged user account or applications with the setuid privilege) can access. BEA advises that no other application use the operating system-specific user account (`wls_owner`) under which WebLogic Server runs.
- If the operating system on which WebLogic Server runs supports security auditing of file and directory access, BEA recommends using audit logging to track any denied directory or file access violations.
- Consider the use of an Intrusion Detection System (IDS) to detect attempts to modify the production environment.

Replace HTML Special Characters in User-Supplied Data

The ability to return user-supplied data can present a security vulnerability called **cross-site scripting**, which can be exploited to steal a user’s security authorization. For a detailed description of cross-site scripting, refer to “Understanding Malicious Content Mitigation for Web Developers” (a CERT security advisory) at http://www.cert.org/tech_tips/malicious_code_mitigation.html.

To remove the security vulnerability, before you return data that a user has supplied, scan the data for HTML special characters. If you find any such characters, replace them with their HTML entity or character reference. Replacing the characters prevents the browser from executing the user-supplied data as HTML.

For more information, see “Securing User-Supplied Data in JSPs” and “Securing Client Input in Servlets.”

Use Protected EJBs to Limit Access to Business Logic

Some parts of your Web application are more sensitive than other parts. For example, the part of your application that renders HTML is less sensitive than the part of the application that accesses a key database table. More effort should be placed on protecting the sensitive parts of your Web application. One way to protect the sensitive parts of your Web application is to wrap them in Enterprise JavaBeans (EJBs) and use ACLs to protect the EJBs. This security technique ensures that only properly authenticated and authorized users can execute the EJBs.

The following is an example of how to use EJBs and ACLs to protect sensitive parts of your Web application:

- Code that allows a user to place an order on your Web site might be in an ACL protected EJB that is only accessible to registered users of your Web site.
- Code that searches and displays the catalog of products on your Web site could be in an EJB that is accessible to all users.
- Code that authorizes a return of merchandise may be in an ACL protected EJB that is only accessible to customer service personnel.

The exact choice of what to protect and to whom to grant access to specific operations must be considered on a per-application basis.

Remember your Web application is going to evolve over time. This makes hard-to-understand schemes even harder to manage. One way to help prevent future mistakes is to organize security by package. For example, you could have one package where all methods on all classes are available to registered users and another package where all methods on all classes are available only to customer service personnel. The final decision as to whom has what access is up to the EJB deployer but a security scheme based on package is an easy mechanism for the deployer to implement.

Use ACLs

ACLs are data structures with multiple entries that guard access to WebLogic Server resources. WebLogic Server provides ACLs that protect the following WebLogic Server resources:

- WebLogic Servers
- WebLogic Events
- WebLogic HTTP servlets/JSPs/HTML pages
- WebLogic JDBC connection pools
- WebLogic JMS destinations
- WebLogic JNDI contexts
- WebLogic MBeans

Use the provided ACLs to protect the resources in your WebLogic Server deployment.

ACLs and permissions for WebLogic EJBs differ from ACLs and permissions for other kinds of WebLogic Server resources in the following ways:

- EJB ACLs are configured in the access control properties of the EJB’s deployment descriptor.
- Permissions are granted on individual methods of a bean; there are no predefined permissions.
- Permissions on EJBs are granted to Roles, which map to groups in WebLogic Server.

For more information, see the following sections:

- ACLs and Permissions
- “Defining ACLs” in Managing Security
- Programming WebLogic Enterprise JavaBeans for information about assigning ACLs to WebLogic EJBs.

- Configuring Security in Web Applications for information about assigning security roles in Weblogic Web applications.

Use the Appropriate Security Realm

Security in WebLogic Server is based on the concept of a security realm. A security realm in WebLogic Server is a collection of Users and Groups, data about the Users and Groups, and permissions for WebLogic Server resources assigned to the Users and Groups. WebLogic Server offers many different types of security realms.

The default security realm, the File realm, is the least secure of security realms. Do not use the File realm for deployed WebLogic Servers. WebLogic Server provides a set alternate security realms that offer authorization and authentication through the facilities of the security realm. For more information about the available security realms, see the Security Realms section of the Concepts chapter.

Secure Your Database

Most Web applications use a database to store their data. Common databases used with WebLogic Server are Oracle, Microsoft's SQL Server, and Informix. The databases frequently hold the Web application's sensitive data including customer lists, customer contact information, credit card information, and other proprietary data. When creating your Web application you must consider what data is going to be in the database and how secure you need to make that data. You also need to understand the security mechanisms provided by the manufacturer of the database and decide whether they are sufficient for your needs. If the mechanisms are not sufficient, you can use other security techniques to improve the security of the database. One common technique is to encrypt sensitive data before writing it to the database. For example, you might leave all customer data in the database in plain text except for the credit card information which is encrypted.

Use Auditing

Auditing is the process of recording key security events in your WebLogic Server environment. The audit record is usually kept separate from the WebLogic Server log file. Reviewing the auditing records can help you determine whether there has been a security breach or an attempted breach. Noting things such as repeated failed logon attempts or a surprising pattern of security events may be the key to preventing serious problems. To use auditing, implement the `weblogic.security.audit` package. For more information, see the “Auditing Security Events” section in Managing Security.

Control Access to Multiple Domains

If the WebLogic Server Administration Console is used to create multiple domains, the System user in one domain can access the other domains created by that instance of the Administration Console. This behavior can present a security risk in a production environment. If you do not want this behavior, make sure that each domain in your enviroment in created in a different location (meaning on a different host or through a different WebLogic Server instance). You can also use the file protections offered through the operating system to protect the `config.xml` file so that only the appropriate System user has access to the file.

CHAPTER

4 Programming with the WebLogic Security SPI

The following sections describe how to program with the WebLogic Security SPI including:

- Before You Begin
- WebLogic Security SPI
- Using JAAS Authentication
- Using JNDI Authentication
- Communicating Securely with SSL-Enabled Web Browsers
- Using Mutual Authentication
- Using a Custom Host Name Verifier
- Using a Trust Manager
- Using an SSL Context
- Using Custom ACLs
- Writing a Custom Security Realm
- Auditing Security Events
- Filtering Network Connections

Before You Begin

This section describes programming with application programming interfaces (APIs) in the Security service provider interface (SPI) supplied by WebLogic Server. Before you perform the programming tasks described in this section, the following configuration tasks must be completed:

1. Specify a security realm (the default, an alternate, or a custom security realm).
2. Add Users and Groups to the security realm.
3. Assign ACLs and permissions to the resources in the security realm.
4. Configure the SSL protocol (an optional step to provide additional protection for network connections or when using certificate authentication).
5. Configure two-way SSL (optional).
6. Configure certificate authentication (optional)

For more information about these configuration tasks, see Managing Security in the *Administration Guide*.

For information about security in WebLogic EJBs, see Programming WebLogic Enterprise JavaBeans.

WebLogic Security SPI

The WebLogic Security SPI builds upon the Java Developer's Kit (JDK) security SPI: it provides implementations and extensions where needed and a realm interface that collects the security APIs into an authentication and authorization service for WebLogic Server. The authentication scheme in WebLogic Server is based on the Java Authentication and Authorization Service (JAAS). This standard provides the support needed to submit a username and credential (password or digital certificate) when initiating a secure connection to WebLogic Server.

Table 4-1 lists the packages that are used when security is used in the WebLogic Server environment.

Table 4-1 WebLogic Security Packages

This Package. . .	**Is Used for. . .**
`javax.security.auth`	Performing JAAS-style LoginContext and Subject based authentication.
`weblogic.security`	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.
`weblogic.security.acl`	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.
`weblogic.security.audit`	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.
`weblogic.security.net`	Examining connections to WebLogic Server and allowing or denying the connections based on attributes such as the IP address, domain, or protocol of the initiator of the network connection.
`weblogic.net.http.HTTPsURLConnection`	Makes an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server.
`weblogic.security.SSL.HostNameVerifier`	Provides support for client authentication. it includes the Host Name Verifier, Trust Manager, and SSL Context classes.

Using 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 who runs the code. JAAS is provided in WebLogic Server as an alternative to the JNDI authentication mechanism. It is the preferred method of authentication. In order to use JAAS, you need to have the Java SDK version 1.3 installed.

Note: The authorization component of JAAS is not provided in WebLogic Server.

Table 4-2 lists the JAAS classes supported in WebLogic Server.

Table 4-2 JAAS Classes

Class	Description
`javax.security.auth.Subject`	Represents the source of the request and can be any entity (for example, a person or a client). A Subject object is created at the completion of a successful user authentication or login.
`javax.security.auth.login.LoginContext`	Through the LoginContext object, an application initiates login, logout, and acquires the authenticated Subject for the purpose of authorization checking.
`javax.security.auth.login.Configuration`	Provides the `getConfiguration()` method for the purpose of obtaining a list of LoginModules provided in a particular implementation of WebLogic Server.
`javax.security.auth.spi.LoginModule`	Provides the ability to implement different kinds of authentication technologies into WebLogic Server. For example, one LoginModule object may perform password authentication while another LoginModule object performs certificate authentication.
`javax.security.auth.callback.Callback`	Gathers input from users (such as a password or the name of a digital certificate file) and passes it to the Java client.

Table 4-2 JAAS Classes (Continued)

Class	Description
`javax.security.auth.callback.Callback.CallbackHandler`	Provides a way for the LoginModule to communicate with a Subject to obtain authentication information. Implements the CallbackHandler interface and passes it to the LoginContext which forwards it directly to the underlying LoginModules. The LoginModules use the CallbackHandler both to gather input from users (such as a password) or to supply information to users (such as status information). By using CallbackHandlers, LoginModules can remain independent of the different ways WebLogic Server communicates with users.

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

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

   WebLogic Server provides a helper class `weblogic.security.auth.Authenticate` that facilitates the writing of a LoginModule class. The `weblogic.security.auth.Authenticate` class uses a JNDI Environment object and returns an authenticated Subject. The JNDI Environment object should include the properties listed in Table 4-3.

2. Implement a Configuration class that specifies which LoginModule classes should be used for your WebLogic Server and in which order the LoginModule classes should be invoked.

3. In the Java client, instantiate a LoginContext.

   The LoginContext consults the Configuration to load all of the LoginModules configured for WebLogic Server.

4. 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.

5. Retrieve the authenticated Subject from the LoginContext.
6. Upon successful authentication of a Subject, access controls can be placed upon that Subject by invoking the `doAs()` method of the `javax.security.auth.Subject` class. The `doAs()` method associates the specified Subject with the current thread's ACL and then executes the action. If the Subject has the necessary access controls the action is completed; however, if the Subject does not have the necessary access controls, a security exception is raised.

The `examples.security.jaas` example in the `samples/examples/security` directory provided with WebLogic Server shows how to use JAAS authentication in a Java client.

Listing 4-1 contains an implementation of the `javax.security.auth.spi.LoginModule` class that performs password authentication. The code in Listing 4-1 is excerpted from the SampleLoginModule in the `examples.security.jaas` package.

Listing 4-1 Example of LoginModule for Password Authentication

```
...

//Import the relevant classes.//
import java.util.Map;
import java.io.IOException;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
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.jndi.Environment;
```

```
public class SampleLoginModule implements LoginModule
{
       private Subject subject = null;
       private CallbackHandler callbackHandler = null;
       private Map sharedState = null;
       private Map options = null;
       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

  public void initialize(Subject subject,
                     CallbackHandler callbackHandler,
                     Map sharedState,
                     Map options)

  {

              this.subject = subject;
              this.callbackHandler = callbackHandler;
              this.sharedState = sharedState;
              this.options = options;

// Retrieve WebLogic Server URL string

              url = System.getProperty
                     ("weblogic.security.jaas.ServerURL");

//Authenticate the user using the username and password passed in.
//Return true if successful.
//Raise FailedLoginException if the authentication fails.
//Raise 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)
       throw new LoginException("No CallbackHandler Specified");

       // Populate callback list//
       Callback[] callbacks = new Callback[2];
       callbacks[0] = new NameCallback("username: ");
       callbacks[1] = new PasswordCallback("password: ", false);
```

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

        // Retrieve username
        username = ((NameCallback) callbacks[0]).getName();

        // 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);
        }

// Populate weblogic environment and authenticate
        Environment env = new Environment();
        env.setProviderUrl(url);
        env.setSecurityPrincipal(username);
        env.setSecurityCredentials(password);

// Authenticate user credentials, populating Subject
        Authenticate.authenticate(env, subject);

// Successfully authenticated subject with supplied info
        succeeded = true;
        return succeeded;

...
```

WebLogic Server uses the default LoginModule (`weblogic.security.internal.ServerLoginModule`) to gather authentication information during server initialization. To replace the default Login module, edit the `Server.policy` file and replace the name of the default Login module with the name of a custom Login module.

The JAAS implementation in WebLogic Server allows the use of NameCallback, PasswordCallback, and TextInputCallback callbacks in the provided LoginModules.

Optionally, custom Login modules can be specified in the `server.policy` file ahead of the default LoginModule. The JAAS implementation in WebLogic Server uses Login modules in the order in which they are defined in the `server.policy` file. The default Login module checks for existing system user authentication definitions prior to execution and does nothing if they are already defined.

The default Login Module is required to define JVM properties for both the system username and password. These properties are specified as `weblogic.management.username` and `weblogic.management.password` respectively. In order to use a custom Login modules, these properties must be set accordingly.

Listing 4-2 contains an implementation of the `javax.security.auth.login.Configuration` class. The code in Listing 4-2 is excerpted from the SampleConfig in the `examples.security.jaas` package.

Listing 4-2 Example of a Configuration Implementation

```
...
import java.util.Hashtable;
import javax.security.auth.login.Configuration;
import javax.security.auth.login.AppConfigurationEntry;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class SampleConfig extends Configuration

{
       String configFileName = null;

//Create a new Configuration object.
       public SampleConfig()

//Retrieve an entry from the Configuration using an application name
//as an index. This code specified a single Login Module.

       public AppConfigurationEntry[]
              getAppConfigurationEntry(String applicationName)
       {
              AppConfigurationEntry[] list =
                            new AppConfigurationEntry[1];
              AppConfigurationEntry entry = null;

// Get the specified configuration file
       configFileName =
              System.getProperty("weblogic.security.jaas.Policy");
       System.out.println("Using Configuration File: " +
              configFileName);

       try
       {
              FileReader fr = new FileReader(configFileName);
              BufferedReader reader = new BufferedReader(fr);
              String line;

       line = reader.readLine();
       while(line != null)
       {

// Skip lines until the line starting with a '{'
       if(line.length() == 0 || line.charAt(0) != '{')
              {
                     line = reader.readLine();
                     continue;
       }
// Read following line which contains the LoginModule configured
       line = reader.readLine();

       int i;
       for(i = 0; i < line.length(); i++)
       {
              char c = line.charAt(i);
              if(c != ' ')
       break;
       }
              int sep = line.indexOf(' ', i);

       String LMName = line.substring(0, sep).trim();
       String LMFlag = line.substring(sep + 1)line.indexOf
                                          (' ', sep + 1));

       System.out.println("Login Module Name: " + LMName);
       System.out.println("Login Module Flag: " + LMFlag);

       if(LMFlag.equalsIgnoreCase("OPTIONAL"))
       {
              entry = new AppConfigurationEntry(LMName,
                     AppConfigurationEntry.LoginModuleControlFlag.
                     OPTIONAL,new Hashtable());
       list[0] = entry;
       }


       else if(LMFlag.equalsIgnoreCase("REQUIRED"))
       {
              entry = new AppConfigurationEntry(LMName,
                     AppConfigurationEntry.LoginModuleControlFlag.
                     REQUIRED, new Hashtable());
       list[0] = entry;
       }


       else if(LMFlag.equalsIgnoreCase("REQUISITE"))
       {
              entry = new AppConfigurationEntry(LMName,
                     AppConfigurationEntry.LoginModuleControlFlag.
                     REQUISITE, new Hashtable());
       list[0] = entry;
       }

       else if(LMFlag.equalsIgnoreCase("SUFFICIENT"))

        {
              entry = new AppConfigurationEntry(LMName,
                     AppConfigurationEntry.LoginModuleControlFlag.
                     SUFFICIENT,new Hashtable());
       list[0] = entry;

...
//Refresh and reload all of the Login configurations.
       public void refresh()

...
```

Listing 4-3 contains an example of a Java client that uses JAAS authentication. The code in Listing 4-3 is excerpted from the SampleClient in the `examples.security.jaas` package. Note that the Java client includes an implementation of the `javax.security.auth.callback.Callback.CallbackHandler` class.

Listing 4-3 Example of Java Client That Uses JAAS Authentication

```
//Import the required classes.
import java.io.*;
import java.util.*;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.TextOutputCallback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
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;

public class SampleClient
{

//Attempt to authenticate the user.
       LoginContext loginContext = null;

//Set JAAS server url system property and create a LoginContext.
{

//Set Server url for SampleLoginModule, the LoginModule for
//the JAAS code example
       Properties property = new Properties();
       property = System.getProperties();
       property.put("weblogic.security.jaas.ServerURL", args[0]);
       System.setProperties(property);

// Set configuration class name to load SampleConfiguration, the
//Configuration for the JAAS code example
       Properties property = new Properties();
       property = System.getProperties();
       property.put("weblogic.security.jaas.Configuration",
                      "examples.security.jaas.SampleConfig");
       System.setProperties(property);

// Set Configuration file name to load sample configuration policy
//file.
       Properties property = new Properties();
       property = System.getProperties();
       property.put("weblogic.security.jaas.Policy",
                        "Sample.policy");
       System.setProperties(property);
```

```
// Create LoginContext
       loginContext = new LoginContext("SampleLoginModule", new
       MyCallbackHandler());
}

//Attempt authentication
       loginContext.login();

//Retrieve authenticated Subject and perform SampleAction as
//Subject.
       Subject subject = loginContext.getSubject();
       SampleAction sampleAction = new SampleAction();
       Subject.doAs(subject, sampleAction);

//Implementation of the CallbackHandler Interface
class MyCallbackHandler implements CallbackHandler

{
       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:
              System.out.println(toc.getMessage());
       break;
              case TextOutputCallback.ERROR:
              System.out.println("ERROR: " + toc.getMessage());
       break;
              case TextOutputCallback.WARNING:
              System.out.println("WARNING: " + toc.getMessage());
       break;
              default:
              throw new IOException("Unsupported message type: " +
              toc.getMessageType());
              }
       }
       else if(callbacks[i] instanceof NameCallback)

      {
// Prompt the user for the username
       NameCallback nc = (NameCallback) callbacks[i];
       System.err.print(nc.getPrompt());
       System.err.flush();
```

```
        nc.setName((new BufferedReader(new
        InputStreamReader(System.in))).readLine());
        }else if(callbacks[i] instanceof PasswordCallback)

       {
// Prompt the user for the password
        PasswordCallback pc = (PasswordCallback) callbacks[i];
        System.err.print(pc.getPrompt());
        System.err.flush();

//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[] password = new char[passLen];
               for(int passIdx = 0; passIdx < passLen; passIdx++)
               password[passIdx] = tmpPassword.charAt(passIdx);
               pc.setPassword(password);
        }
        else
        {
        throw new UnsupportedCallbackException(callbacks[i],
        "Unrecognized Callback");
        }
...
```

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

Using JNDI Authentication

Java clients can also use JNDI to pass credentials. 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.

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

Table 4-3 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`.
`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.

Table 4-3 JNDI Properties Used for Authentication (Continued)

Property	Meaning
`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 4-4 demonstrates how to use password authentication in a Java client. The code in Listing 4-4 is excerpted from the Client in the `examples.security.acl` example provided with WebLogic Server in the `samples/examples/security` directory.

Listing 4-4 Example of Password Authentication

```
...
Hashtable env = new Hashtable();
       env.put(Context.INITIAL_CONTEXT_FACTORY,
              "weblogic.jndi.WLInitialContextFactory");
       env.put(WLContext.PROVIDER_URL, “t3://weblogic:7001”);
       env.put(WLContext.SECURITY_AUTHENTICATION “simple”);
       env.put(Context.SECURITY_PRINCIPAL, “javaclient”);
       env.put(Context.SECURITY_CREDENTIALS, “password”);

      ctx = new InitialContext(env);
```

Listing 4-5 demonstrates how to use certificate authentication in a Java client. Notice the use of the T3S protocol, which is a WebLogic Server proprietary protocol over the SSL protocol. The SSL protocol protects the connection and communication between WebLogic Server and the Java client.

Listing 4-5 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 4-4 and Listing 4-5 generates a call to `weblogic.security.acl.Security.getUser()` which returns a User object if the username and password are correct or 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 ACLs.

Note: For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see “JNDI Contexts and Threads” and “How to Avoid JNDI Context Problems” in the *Programming WebLogic JNDI*.

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/examples/security/sslclient` directory.

Using Mutual 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 that presented it.

With mutual authentication, the requesting client also presents a digital certificate to WebLogic Server. By setting fields in the Administration Console, you can configure WebLogic Server to require requesting clients 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 more information, see the “Configuring the SSL Protocol” section in Managing Security.

The following sections describes the different ways mutual authentication can be implemented in WebLogic Server.

Note: When using JAAS for authentication in a Java client, you write a LoginModule class that performs mutual authentication.

Mutual Authentication with JNDI

When using JNDI for 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 `fileRealm.properties` 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 4-6 demonstrates how to use mutual authentication in a Java client. The code in Listing 4-6 is excerpted from the AltClient in the `examples.security.acl` example in the `samples/examples/security` directory provided with WebLogic Server.

Listing 4-6 Example of Mutual Authentication

```
package examples.security.acl;

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


public class AltClient
{

  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 ACL-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.

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 implement the CertAuthenticator interface, WebLogic Server passes the digital certificate for the Java client to the implementation of the CertAuthenticator class. The CertAuthenticator class maps the digital certificate to a WebLogic Server User. Because the digital certificate is processed only at the time of the first connection request from the JVM, it is not possible to set a new user identity when you use the CertAuthenticator class.

Caution: **Restriction:** Multiple, concurrent, user logins to WebLogic Server from a single client JVM when using mutual authentication and JNDI is not supported. If multiple logins are executed on different threads, the results are undeterminable and might result in one user's requests being executed on another user's login, thereby allowing one user to access another user's data. WebLogic Server does not support multiple, concurrent, certificate-based logins from a single client JVM. For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see "JNDI Contexts and Threads" and "How to Avoid JNDI Context Problems" in *Programming WebLogic JNDI*.

Mapping a Digital Certificate to a WebLogic Server User

When you perform mutual authentication, WebLogic Server authenticates the digital certificate of the Web browser or Java client in order to establish an SSL connection. However, the digital certificate does not identify the Web browser or Java client as a User in the WebLogic Server security realm. If the Web browser or Java client requests a WebLogic Server resource protected by an ACL, WebLogic Server requires the Web browser or Java client to provide a username and password.

To map a Web browser or Java client to a User in the WebLogic Server security realm, implement the `weblogic.security.acl.CertAuthenticator` interface. The CertAuthenticator class is called after an SSL connection has been established. The class can extract data from a digital certificate to determine which User owns the digital certificate. The CertAuthenticator class then calls the `weblogic.security.acl.getUser()` method to retrieve the authenticated User object from the WebLogic Server security realm.

When the CertAuthenticator class is installed, it is unnecessary for Web browsers to prompt for WebLogic Server usernames and for Java applications to set a password in the JNDI `SECURITY_CREDENTIALS` property. For more information, see the "Configuring the SSL Protocol" section in Managing Security.

If you use the CertAuthenticator class with a Java client application, note that the Java client cannot change the User identity once the SSL connection is established. To supply a new digital certificate, you must stop the JVM for the Java client and restart the client in a new JVM instance so that a new SSL connection can be negotiated.

You can use any of the several methods to map a digital certificate to a User. One technique is to set the password of a User to the fingerprint of the User's digital certificate. Then you can extract the username from the digital certificate, calculate the fingerprint**,** and call the `weblogic.security.acl.getUser()` method in the same way WebLogic Server does when a User submits a username and a password to access a resource.

Note: The fingerprint of a digital certificate is not part of the certificate but it can be computed from the certificate. A fingerprint is the MD5 digest of the DER-encoded `CertificateInfo` which is an ANS.1 type included in the X.509 specification.

The CertAuthenticator class has a public no-arg constructor and invokes the `authenticate()` method. WebLogic Server calls the `authenticate()` method with a username, which may be null, a `Certificate` array containing the digital certificate of the Java client or a certificate chain, and a Boolean that is `true` if an SSL handshake succeeds. You can call methods on the `Certificate` array to retrieve data from the digital certificate.

Listing 4-7 shows how to implement the CertAuthenticator interface. It extracts the username from the e-mail address in the digital certificate and calls the `weblogic.security.acl.getUser()` method to retrieve an authenticated User object from the WebLogic Server security realm. Because the code example examines only a portion of the e-mail address, this example is not very secure. Digital certificates with the same e-mail address in different domains can be mapped to the same User and no additional authentication is performed. If you want to implement this feature, you may want to add code that fully establishes the identity of the client.

Listing 4-7 also shows how to map a digital certificate to a User in a WebLogic Server security realm. The code in Listing 4-7 is excerpted from SimpleCertAuthenticator in the `examples.security.cert` example in the `samples/examples/security` directory provided with WebLogic Server.

Listing 4-7 Example of Mapping a Digital Certificate to a WebLogic Server User

```
package examples.security.cert;


import weblogic.security.Certificate;
import weblogic.security.Entity;
import weblogic.security.X500Name;
import weblogic.security.acl.CertAuthenticator;
import weblogic.security.acl.BasicRealm;
import weblogic.security.acl.Realm;
import weblogic.security.acl.User;

public class SimpleCertAuthenticator
  implements CertAuthenticator
{
  private BasicRealm realm;

  public SimpleCertAuthenticator()
  {
    realm = Realm.getRealm("weblogic");
  }


  /**
   * Attempt to authenticate a remote user.
   *
   * @param userName ignored by this example
   * @param certs used to attempt to map from email address to
   * @a WebLogic user.
   * @param ssl if false, this example returns null
   * @return authenticated user, or null if authentication failed
   */
  public User authenticate(String userName, Certificate[] certs,
boolean ssl)
  {
    // This implementation only trusts certificates that originate
    // from a successful two-way SSL handshake.
    if (ssl == false)
    {
      return null;
    }

    User result = null;
    Certificate cert = certs[0];
    Entity holder = cert.getHolder();

    if (holder instanceof X500Name)
    {
      X500Name x500holder = (X500Name) holder;
      String email = x500holder.getEmail();

      if (email != null)
      {
int at = email.indexOf("@");

if (at > 0)
{
  String name = email.substring(0, at);

  // Make sure that the user we've pulled out of the email
  // address really exists.
  result = realm.getUser(name);
}
      }
    }

    return result;
  }
}
```

The constructor in Listing 4-7 shows how to get access to the WebLogic Server security realm in a server-side class. The `getRealm("weblogic")` method of the `weblogic.security.acl.realm` class returns the realm being used by WebLogic Server, whether it is the File realm or an alternative security realm, such as the LDAP Security realm.

The `weblogic.security.X500Name` class includes accessor methods to retrieve fields from the `weblogic.security.Certificate` class. Listing 4-7 casts the Certificate object to an X500Name object, calls the `getEmail()` method of the X500Name object, and takes the initial substring of the e-mail address. The `getUser(String)` method in the `weblogic.security.acl.AbstractableRealm` class retrieves the WebLogic Server user with the computed username. If the user does not exist, the `authenticate()` method of the `weblogic.security.acl.AbstractableRealm` class returns null.

Using Mutual Authentication with Other WebLogic Servers

You can use mutual authentication in server-to-server communication in which one WebLogic Server is acting as the client of another WebLogic Server. Using mutual authentication in server-to-server communication allows you to depend on high-security connections, even without the more familiar client/server environment.

Listing 4-8 establishes a secure connection to a second WebLogic Server called `server2.weblogic.com` from a servlet running in WebLogic Server.

Listing 4-8 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");

T3Client t3c = e.createProviderClient();
t3c.connect();

e.setInitialContextFactory
      (“weblogic.jndi.WLInitialContextFactory”);
Context ctx = new InitialContext(e.getProperties())

...
```

In Listing 4-8, the WebLogic JNDI Environment class creates a hash table to store the following parameters:

- `setProviderURL`—specifies the URL of the WebLogic Server instance acting as the SSL server. The WebLogic Server instance acting as SSL client calls this method. 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 the two WebLogic Servers instances.
- `setSSLClientCertificate`—specifies a certificate chain to use for the SSL connection. You use this method to specify an input stream array that consists of a private key (which is the first input stream in the array) and a chain of X.509 certificates (which make up the remaining input streams in the array). Each certificate in the chain of certificates is the issuer of the certificate preceding it in the chain.

- `setSSLServerName`—specifies the name of the WebLogic Server instance acting as the SSL server. When the SSL server presents its digital certificate to the server acting as the SSL client, the name specified using the `setSSLServerName` method is compared to the common name field in the digital certificate. In order for hostname verification to succeed, the names must match. This parameter is used to prevent man-in-the-middle attacks.
- `setSSLRootCAFingerprint`—specifies digital codes that represent a set of trusted certificate authorities. The root certificate in the certificate chain received from the WebLogic Server instance acting as the SSL server has to match one of the fingerprints specified with this method to be trusted. This parameter is used to prevent man-in-the-middle attacks.

Note: For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see "JNDI Contexts and Threads" and "How to Avoid JNDI Context Problems" in the *Programming WebLogic JNDI*.

Using Mutual 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.net.ssl.cipher_suite`—returns the cipher suite in use as a string.
- `javax.net.ssl.session`—returns the SSL Session object that contains the cipher suite and the dates on which the object was created and last used.
- `javax.net.ssl.rootCA`—returns the digital certificate from the specified certificate authority as a string.
- `javax.net.ssl.rootCADigest`—returns the certificate digest from the specified certificate authority as a string.

- `javax.net.ssl.peer_certificates`—obtains information about the client's certificate chains and returns an array of digital certificate as objects. You can then cast the array to `weblogic.security.X509`.

You have access to the user information defined in the digital 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)

You can see the available information in each digital certificate by calling the `to_string()`method on the X509 object that represents the digital certificate.

You can create a `weblogic.security.JDK11Certificate` object for a digital certificate by passing an X509 object to its constructor. The `weblogic.security.JDK11Certificate` class implements the `java.security.Certificate` interface.

Listing 4-9 converts the first X509 object in an array of X509 objects to a JDK111Certificate object.

Listing 4-9 Converting an X509 Object to a JDK111Certificate Object

```
weblogic.security.JDK11Certificate jdk11cert =
       new weblogic.security.JDK111Certificate(X509certs [0]);

print(out, “jdk11cert.getPrincipal().getName() -”,
            jdk11cert.getPrincipal().getName() );
print(out, “jdk11cert.getGuarantor().getName() -”,
            jdk11cert.getGuarantor().getName() );
```

The `weblogic.security.JDK11Certificate` class has the following member functions that provide additional information about the digital certificate:

- `getIssuerCertificate()`—returns the digital certificate of the issuer as a `java.security.Certificate` object.
- `getFingerprint()`—returns the fingerprint of the digital certificate. The fingerprint is the MD5 of the DER-encoded digital certificate. It is difficult to construct a different digital certificate with the identical fingerprint.
- `getSubjectOrgUnit()`—returns a string containing the Organizational unit of the distinguished name in the digital certificate.

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 Server or a WebLogic client is acting as an SSL client to another application server. It prevents man-in-the-middle attacks.

The default behavior of 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.HostnameVerifier` 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.HostnameVerifier` interface and define the methods that capture information about the server's security identity.

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.HostnameVerifier=hostnameverifier
  ```

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

To make a connection that uses a non-default JDK protocol handler, make sure to initialize the handler by calling these two functions:

```
weblogic.net.http.Handler.init();
weblogic.management.application.Handler.init();
```

An example of a custom Host Name Verifier is available in the SSLclient code example in the `examples.security.sslclient` package in the `/samples/examples/security/sslclient` directory. This code example contains a NullHostnameVerifier 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.

You can associate an instance of a Host Name verifier with an SSL Context through the `setHostnameVerifier` method. For example:

```
public void setHostnameVerifier (HostnameVerifier hv)
```

See the following information on Using an SSL Context.

Using a Trust Manager

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

When an SSL client connects to an SSL server, the SSL server presents its digital certificate chain to the client for authentication. That chain can sometimes contain an invalid digital certificate. The SSL specification says that the client should drop the SSL connection upon discovery of an invalid certificate. Web browers, however, attempt 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.

Use the `weblogic.security.SSL.TrustManager` class to create a Trust Manager. The class 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.TrustManager` class uses a callback function to override the result of verifying the digital certificate. You can associate an instance of a Trust Manager wtih an SSL Context through the `setTrustManager()` method. The `weblogic.security.SSL.TrustManager` class conforms to the JSSE specification. Note that the use of a Trust Manager does not impact performance. You can only set up a Trust Manger programmatically; its use is not defined through the Administration Console or on the command-line.

Examples of using Trust Manager are available in the `/samples/examples/security/sslclient` directory:

- This example shows how to set up a new SSL connection by using an SSL Context with the Trust Manager:

  `/samples/examples/security/sslclient/SSLSocketClient`

- This example contains a custom Trust Manager that always returns true:

  `/samples/examples/security/sslclient/NulledTrustManager`

Using an SSL Context

SSL Context 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 SSL Context class is used as a factory for SSL sockets. For example, all sockets that are created by socket factories provided by the SSL Context can agree on session state by using the handshake protocol associated with the SSL Context. 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 SSL Context can resuse 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 SSL Context, use the setTrustManager method.

You can only set up SSL Context programatically; not by using the Administration Console or the command line. A Java new expression or the getInstance () factory method of the SSL Context class can create an SSL Context object. The getInstance() factory method is static and it returns an instance to implement a secure socket protocol. An example of using SSL Context is available in the `/samples/examples/security/sslclient` directory:

- This example shows how to create a new SSL Socket Factory that will create a new SSL Socket using SSL Context:

  `/samples/examples/security/sslclient/SSLSocketClient`

SSL Context conforms to Sun Microsystems's Java Secure Socket Extension (JSEE), so is forward-compatible code.

Using Custom ACLs

WebLogic Server defines a standard set of access control lists (ACL)s to protect resources. If you create an ACL for a resource, WebLogic Server automatically checks the permissions for that resource before allowing anyone to access it. Most resources in WebLogic Server can be fully protected with these standard ACLs.

Some resources, however, require more protection than is offered by the standard set of ACLs. WebLogic Server allows you to augment the security for such resources. You may, for example, create a servlet that checks user permissions before writing certain data to a Web page. The `weblogic.security.acl.Security` class provides access to realm operations, such as checking an ACL. This class is available only to server-side code.

The `Security.hasPermission()` and `Security.checkPermission()` methods in the `weblogic.security.acl.Security` class test whether a user has the required permission necessary to access a resource. The two methods are similar except that the `Security.hasPermission()` method returns a Boolean (`true` if the user has the relevant permission) and the `Security.checkPermission()`method throws `java.lang.SecurityException` if the user does not have the permission.

The `examples.security.acl` example provided with WebLogic Server (in the `samples/examples/security` directory) shows how to create your own ACLs and test them in a server-side class. The custom ACL protects an RMI class, `FrobImpl`, with a custom ACL named `aclexample` that has the permission of `frob`.

Note: Before a custom ACL can be used, the ACL must be installed in the security realm being used by WebLogic Server.

To use the custom ACL, the Java client application must:

1. Get a JNDI InitialContext from WebLogic Server.
2. Look up the protected resource in the WebLogic Server JNDI tree using the permission name. For example, the Java client looks up `FrobImpl` using the permission name, `frob`.
3. Execute the `frob()` method on the RMI stub.

The `FrobImpl` class contains the server-side code that tests the ACL. Listing 4-10 shows how to use the static `checkPermission()` method in the `weblogic.security.acl.realm` class to test the custom ACL.

Listing 4-10 Testing Custom ACLs in the Default Security Realm

```
Security.checkPermission(Security.getCurrentUser(),
          "aclexample",
          Security.getRealm().getPermission("frob"),
          '.');
```

You can also test custom ACLs in alternate security realms:

1. Use the getRealm(realm_name) method of the weblogic.security.acl.realm class to get the alternate security realm.
2. Retrieve the custom ACL and permission using the getACL()and the getPermission()methods of the weblogic.security.acl.realm class.
3. Test the permission by calling the acl.checkPermission()method.

Listing 4-11 illustrates this technique for testing custom ACLs.

Listing 4-11 Testing Custom ACLs in an Alternate Security Realm

```
User p = Security.getCurrentUser();
BasicRealm realm = Realm.getRealm(realm_name);
Acl acl = realm.getAcl(acl_name);
Permission perm = realm.getPermission(permission_name);
boolean result = acl == null || !acl.checkPermission(p, perm);
```

The last line in Listing 4-11 tests whether the custom ACL was found and whether the user p has the frob permission. The sense of the test is reversed; if the ACL exists and the user has frob permission, the result is false.

If a security realm does not implement the getACL() functionality, it should throw a java.lang.UnsupportedOperationException exception. ACL lookups will then fall back to the secondary realm. If no secondary realm is configured, a runtime error will occur.

If you audit security events and you use the technique in Listing 4-12 for testing permissions, you must explicitly call the static Audit class of the weblogic.security.audit package if you want to audit your permission tests. The call to the Audit class generates a notification of the permission-checking event to the AuditProvider class in WebLogic Server.

Listing 4-12 Testing Permission

```
    Audit.checkPermission("Frob", acl, p, perm, !result);
```

For a complete code example that uses custom ACLs, see the `examples.security.acl` package in the `samples/examples/security` directory provided by WebLogic Server.

Writing a Custom Security Realm

You may need to create your own security realm to draw from an existing security store in your environment such as a directory server on the network. To write a custom security realm that supports authentication you need to write code that:

1. Defines a User class for the custom security realm.
2. Defines a Group class for the custom security realm.
3. Defines an enumeration class that return all Users and Groups in a security store and releases the resources of the security store when finished.
4. Defines a class for the custom security realm.
5. Obtains configuration data about the security store.
6. Authenticates a User.
7. Returns the members of a Group and creates a hash table that contains the members of a Group.
8. Returns a User object given a User name.
9. Returns a Group object given a Group name.
10. Uses an enumeration for Users to return User objects for all the Users in the security store.

11. Uses an enumeration for Groups to return Group objects for all the Groups in the security store.

You can also write a custom security realm that supports authorization. For more information, see “Using Authorization in a Custom Security Realm.”

Note: WebLogic Server also provides the capability to create a custom security realm that can be managed through the WebLogic Server Administration Console. For more information, see the Javadoc for the `weblogic.security.acl` package or contact BEA Professional Services.

Do not execute RMI calls to another server from a custom security realm. Using RMI calls may cause the server to run out of socket reader threads.

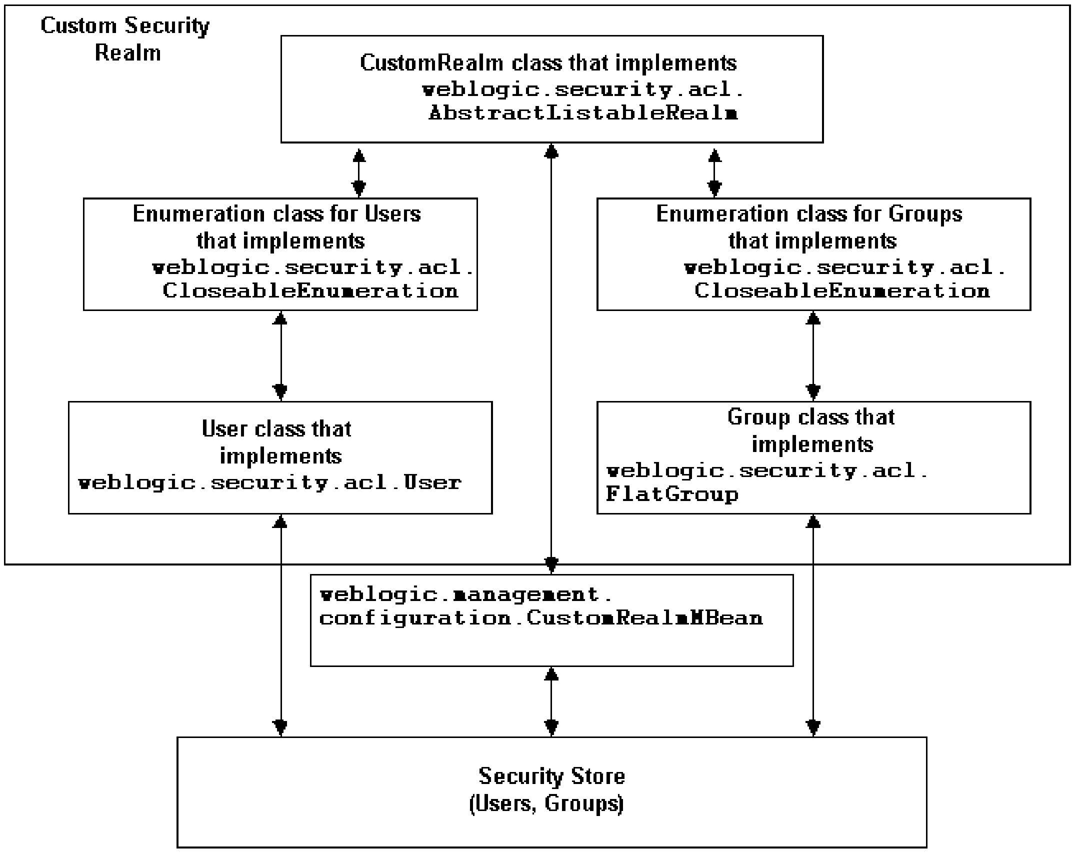
Table 4-4 lists the WebLogic classes used to create a custom security realm.

Table 4-4 WebLogic Classes Used to Create Custom Security Realms

Class	Definition
`weblogic.security.acl.User`	Defines a User that is retrieved from a local security store.
`weblogic.security.acl.FlatGroup`	Defines a Group whose membership is updated when the local security store is updated.
`weblogic.security.acl.CloseableEnumeration`	Defines an enumeration that can be closed thus releasing resources.
`weblogic.security.acl.AbstractListableRealm`	Allows you to create a security realm whose Users, Groups, ACLs, and permissions can be viewed through the Administration Console. However, you need to use the facilities provided by the security store you are using to add and delete Users, Groups, and ACLs and assign permissions to Users and Group.
`weblogic.security.acl.RefreshableRealm`	Synchronizes the information about Users, Groups, ACLs, and permissions displayed in the Administration Console with the information in the local security store.
`weblogic.management.configuration.CustomRealmMbean`	Obtains configuration information about the security store accessed from the custom security realm.

Figure 4-1 illustrates how these classes work together to create a custom security realm.

Figure 4-1 WebLogic Classes Used to Create Custom Security Realms


Custom Security Realm
CustomRealm class that implements weblogic.security.acl. AbstractListableRealm
Enumeration class for Users that implements weblogic.security.acl. CloseableEnumeration
Enumeration class for Groups that implements weblogic.security.acl. CloseableEnumeration
User class that implements weblogic.security.acl.User
Group class that implements weblogic.security.acl. FlatGroup
weblogic.management. configuration.CustomRealmMBean
Security Store (Users, Groups)

The following sections describe the programming tasks required to write and custom security realm.

Define a Class for Users

Extend the `weblogic.security.acl.User` class to create a User class for the custom security realm.

Listing 4-13 contains code that defines a User class.

Listing 4-13 Defining a User Class

```
// Import the required classes
import weblogic.security.acl.user;
...

// Create a custom User class for the custom realm
   /*package */class CustomRealmUser
     extends User
  {
    // Keep track of the User’s custom realm
    CustomRealm realm = null;

    // Implement a constructor
    /*package*/ CustomRealmUser(String name, CustomRealm realm);

      {
        // Call base constructor class passing in the name of the
        // User
        super(name);

        // Keep track of this User’s realm
        this.realm = realm;
      }
        // Return the User’s custom realm
        public BasicRealm getRealm()
         {
            return realm;
          }
    }
...
```

Define a Class for Groups

Groups make it easier to manage security. Internally, a custom security realm represents a Group as a hash table containing a list of members which can be Users or Groups.

To implement a Group in a custom security realm, extend the `weblogic.security.acl.Flatgroup` class to create a new Group with no membership information. The constructor for the class takes as input the name of the desired Group and the realm object corresponding to the custom security realm.

The Flatgroup class is especially designed to work with custom security realms. A custom security realm needs to periodically update Group membership. The FlatGroup maintains its group membership in a cache instead of in a static set. When the cache expires, the Group implementation queries the security store to obtain the most recent membership information. The default time for the Group cache is five minutes which means any changes you make in the underlying store will be recognized in the custom security realm within five minutes. You can tune this value by setting the GroupMembershipCacheTTL field in the Administration Console to the number of seconds a cached group remains valid.

Listing 4-14 contains code that defines a Group class.

Listing 4-14 Defining a Group Class

```
// Import the required classes
import weblogic.security.acl.FlatGroup;
...

/*package*/ class CustomRealmGroup
extends FlatGroup
{
    // Implement a constructor
    /*package*/ CustomRealmGroup(String name,
        CustomRealm realm);
      {
        // Call the base class constructor passing in the name
        // of the Group and custom security realm
        super(name, source);
      }
        // Implement a method that returns the user class for
        // custom security realm
        protected Class getUserClass()
      {
        return CustomRealmUser.class;
      }
}
...
```

Define Enumeration Classes for Users and Groups

Write enumeration classes for Users and Groups. If the enumeration holds resources that must be released when the enumeration is done (for example, release a database cursor), then implement the `weblogic.security.acl.CloseableEnumeration` class. Otherwise, implement the `java.util.Enumeration` interface. In the enumerator constructor, pass the arguments needed to access the security store.

Do not create a User or Group object for every User or Group in the custom security realm and put them in a hash table that enumerates over them. A custom security realm in a deployed WebLogic Server can have more Users and Groups than can fit into memory. Instead, use a database cursor which can be used to incrementally create User or Groups objects as they are needed.

Listing 4-15 contains code that defines enumeration classes for users and groups.

Listing 4-15 Defining Enumeration Classes for Users and Groups

```
// Import the required classes
import weblogic.security.acl.FlatGroup;
import weblogic.security.acl.ClosableEnumeration;

...
// Define an enumeration class for users.
/*package*/ class CustomRealmUsersEnumeration
 implements CloseableEnumeration
{
    // Keep data members here (for example, a database cursor)

      // Keep track of the enumeration’s security realm (for
      // use with the User constructor)
      private CustomRealm realm      = null;

      // Implement a constructor
      /*package */ CustomRealmUsersEnumeration(...,
                            CustomRealm realm)
      {
        this.realm     = realm;
      }

      // Implement a method to determine if there are more Users.
      public boolean hasMoreElements()
      {
        //For example, use a database cursor to see if there are
        // more users
        return (there are more users ...) ? true : false;
      }

        // Implement a method to return the next user.
        // The method must return users objects that use the
        // User class for the custom security realm
        public Object nextElement()
        {
          // For example, use the database cursor to get the name
          // of the next user.
          return new CustomRealmUser(next user name ..., realm);
        }

        // Implement a method to terminate the enumeration.
        // This step is optional
        public void close()

       {
          // If this enumeration is delegating to an iterator that
          // needs to be closed (e.g., a database cursor), release
          // the resources here.
       }
}

// Create a Group class for the custom security realm
/*package*/class CustomRealmGroupsEnumeration
   implements CloseableEnumeration
{
    // Keep data members here (for example, a database cursor

      // Keep track of the enumeration’s security realm (for
      // use with the Groups constructor)
      private CustomRealm realm      = null;

        // Implement a constructor
        /*package */ CustomRealmUsersEnumeration(...,
                            CustomRealm realm)
        {
          this.realm     = realm;
        }

        // Implement a method to determine if there are more Groups.
        public boolean hasMoreElements()
        {
          // For example, use a database cursor to see if there are
          // more groups
          return (there are more groups ...) ? true : false;
        }

        // Implement a method to return the next group.
        // The method must return group objects that use the
        // Group class for the custom security realm
        public Object nextElement()
        {
          // For example, use the database cursor to get the name
          // of the next group.
          return new CustomRealmGroup(next group name ..., realm);
        }

       // Implement a method to terminate the enumeration.
        // This step is optional
        public void close()

       {
          // If this enumeration is delegating to an iterator that
          // needs to be closed (e.g., a database cursor), release
          // the resources here.
       }
}
...
```

Define a Class for the Custom Security Realm

Extend the `weblogic.security.acl.AbstractListableRealm` class to define a new class for the custom security realm and implement a constructor that creates the custom security realm. This class needs to:

1. Obtain configuration data for the security store.
2. Authenticate Users.
3. Determine the members of a Group

4. Get Users and Groups from the security store.

Weblogic Server caches User and Group information in memory. Therefore, BEA recommends having a custom security realm go to disk to whenever it needs User or Group information. The WebLogic Server system administrator can only change Users or Groups for a custom security realm by editing the information in the security store with tools provided for the security store. After the system administrator changes User or Group information in the security store, the system administrator must update the information in the Administration Console by clicking on the Reset button. This action automatically flushes the User and Group information kept in memory.

If the custom security realm caches User or Group information in memory, implement the `weblogic.security.acl.RefreshableRealm` class and the `refresh()` method of the class. When the system administrator updates resets the realm, the `refresh()` method is called. Use the `refresh()` method to discard any cached User or Group information.

Listing 4-16 contains code that defines a class for a custom security realm.

Listing 4-16 Defining a Class for a Custom Security Realm

```
...
// Import the necessary classes
import weblogic.security.acl.AbstractListableRealm;
import weblogic.security.acl.BasicRealm;
import weblogic.security.acl.RefreshableRealm;
import weblogic.server.Server;

// Create a class for the custom security realm
public class CustomRealm
   extends AbstractListableRealm // Required
   implements Refreshable Realm // Optional

// Implement a constructor that creates the custom security realm
public CustomRealm()
{
       super(“Custom Realm”);
...

public void refresh()
{
       // Discard User and Group information in-memory
}
```

Obtain Configuration Data for the Security Store

In order to connect to the security store, configuration properties (for example, a property that contains a URL or a property that specifies a directory path) that specify how to access the security store used by the custom security realm must be defined. Once these properties are defined, the system administrator for WebLogic Server must set the properties in the Configuration Data section of the Custom Security Realm Create window in the Administration Console. The properties for the security store must be defined in the WebLogic Server administration environment before the custom security realm can be used.

For example, define two properties `userInfoFileName` and `groupInforFileName` that specify directory paths to files that contain User and Group information. The system administrator enters those properties in the Configuration Data section of the Custom Security Realm Create window in the Administration Console.

In the code for the custom security realm, use the `weblogic.management.Helper` and `weblogic.management.configuration.DomainMbean` classes to retrieve configuration properties for the custom security and use those properties to connect to the security store.

Listing 4-17 contains code that retrieves configuration properties for the security store and then connects to the security store.

Listing 4-17 Accessing the Security Store

```
...
// Import the necessary classes
import weblogic.management.Helper;
import weblogic.management.configuration.DomainMBean;

MBeanHome mHome = Helper.getMBeanHome(user, password, url,
                                        servername);
DomainMBean domainMBean = mHome.getActiveDomain();
SecurityMBean secMBean = domainMBean.getSecurity();
BasicRealmMBean basicRealmMBean =
              secMBean.getRealm().getCachingRealm.getBasicRealm();

CustomRealmMBean customRealmMBean =
  (CustomRealmMBean)basicRealmMBean;

// Get configuration data from the CustomRealmMBean
Properties configData = customRealmMBean.getConfigurationData();
```

```
// Get the properties for the custom security store.
String userInfoFileName =
       configData.getProperty(“UserInfoFileName);
String groupInfoFileName =
       configData.getProperty(“GroupInfoFileName);
```

Authenticate Users

If the custom security realm uses passwords to authenticate Users, you must explicitly implement the `authUserPassword()` method to authenticate Users. Write code that checks the security store to ensure that the user specified by the `authUserPassword()` method exists with the supplied password.

- If the user exists and the password is correct, return a User object. Use the User class for the custom security realm.
- If the user does not exist or if the password is incorrect, the authentication fails. In this case, return null to indicate the authentication failed.

Listing 4-18 contains code that authenticates a User.

Listing 4-18 Authenticating Users

```
...
//Implement a method which authenticates a user
protected User authUserPassword(String name, String password)
{
    // Check the security store to see if there is a user
    // named name with password password

    if (...) {
              return new CustomRealmUser(name, this);
    }  else   {
           return null;
   }
}
```

Determine the Members of a Group

Use the `getGroupMembersInternal()` method of the `weblogic.security.acl.AbstractListableRealm` class to get the members of a group and build a hash table with the members of a group.

Listing 4-19 contains code that obtains the members of a Group.

Listing 4-19 Obtaining the Members of a Group

```
...
// Implement a method to return members of a group.
protected Hashtable getGroupMembersInternal(String name)
{
       // Check the security store to see if there is a group with
       // the specified name
       if (!...)  {

       // If the group does not exist, throw an exception
       throw new CustomRealmException(“No such group : “ + name);
    }

       // Create a hash table which is filled with group members
       // and is then returned.

       Hashtable members = new Hashtable();

       // Get the group members from the security store
       for (...) {// loop over the members
          if (...) {
          // If this member is a user, create a user object
          // for this user and add it to the list of members.
          // Use the User class created for the custom
          // security realm.
          members.put(memberName, new CustomRealmUser
                    (memberName, this));
          }  else if
          // If this member is a group, create a group object
          // for this group and add it to the list of members.
          // Use the Group class created for the custom
          // security realm.
          members.put(memberName, new CustomRealmGroup
                    (memberName, this));
          }
    }
    // Return the hash table containing the members of the group.
    return members;
  }
...
```

Get Users and Groups from Security Store

Implement the `getUser()` and `getGroup()` methods of the `weblogic.security.acl.AbstractListableRealm` class to retrieve Users and Groups from the security store into the custom security realm. Use the `getUsers()` and `getGroups()` methods of the `weblogic.security.acl.AbstractListableRealm` class to return enumeration objects that return User objects and Group objects for the User and Groups in the security store. The Users and Groups in the security store can now be viewed through the Administration Console.

Listing 4-20 contains code that retrieves Users and Groups from the security store.

Listing 4-20 Retrieving Users and Groups from the Security Store

```
...
// Implement a method to return a User object given the name of
// the user.
public User getUser(String name)
{
  // Check the security store to see if there is a user with the
  // the specified name.
  if (...) {
  // If the user exists, return a User object for the
  // the user. Use the User class created for the custom
  // security realm.
  return new CustomRealmUser(name, this);
  } else {
  // Return a null if the user does not exist
  return null;
  }
}

// Implement a method to return a Group object given the name of
// the Group.
public Group getGroup(String name)
{
  // Check the security store to see if there is a group with the
  // the specified name.
  if (...) {
  // If the group exists, return a Group object for the
  // the Group. Use the Group class created for the
  // custom security realm
  return new CustomRealmGroup(name, this
  } else {
  // Return a null if the group does not exist
  return null;
  }
}

// Implement a method to return an enumeration object that can
// can be used to iterate over all the users in the custom security
// realm.
public Enumeration getUsers()
{
  // Return an enumeration object that returns User object for
  // the users in the security store. Use the user enumeration
  // class created for the custom security realm. Remember to
  // give the enumeration access to the security store so that
  // it can iterate over the users.
  return new CustomRealmUsersEnumeration(..., this);
}

// Implement a method to return an enumeration object that can
// can be used to iterate over all the groups in the custom security
// realm.
public Enumeration getGroups()
{
  // Return an enumeration object that returns Group object for
  // the Groups in the security store. Use the group
  // enumeration class created for the custom security realm.
  // Remember to give the enumeration access to the security
  // store so that it can iterate over the Groups.
  return new CustomRealmGroupsEnumeration(..., this);
}
...
```

Note: To improve performance, use the `isMember()` method of the `weblogic.security.acl.FlatGroup` class. This method checks to see if an individual user is a member of a group rather than fetching all group members.

Using Authorization in a Custom Security Realm

To construct an ACL in a custom security realm, create a new AclImpl object, set its name, and then add an AclEntryImpl object for each User or Group. Each AclEntry object contains a set of permissions associated with a particular principal which represents a User or a Group.

The AclImpl interface imposes an ordering constraint on constructing ACLs. You must add all permissions to an AclEntryImpl object before you add the AclEntryImpl object to the AclImpl object.

In the class for the custom security realm, use the `getAcl()` method to retrieve ACLs from a security store. The `getAclInternal()` method steps through the result set creating an AclImpl object, creating an AclEntryImpl object for each User or Group, and then adding permissions to the AclEntryImpl object. When an AclEntryImpl object is finished, it is added to the AclImpl object. When the AclImpl object is finished, the `getAclInternal()` method returns the finished ACL.

Auditing Security Events

The `weblogic.security.audit` package allows you to use an audit SPI for events that occur in the WebLogic Server security realm. The package includes an interface, AuditProvider, and a static class, Audit, to which WebLogic Server sends auditable security events.

To enable auditing, you create a class that implements the AuditProvider interface and the methods that represent the security events you want to audit. WebLogic Server calls the methods on your class when a user attempts to authenticate, when a permission is tested, or when an invalid digital certificate or root digital certificate is presented. Your AuditProvider class receives the information for each event type and processes the event in whatever way you choose. For example, it could log only unsuccessful authentication requests in the WebLogic Server log file, or record all auditable events in a database table.

Before you can use an AuditProvider class, you need to install the class through the Administration Console. For more information, see the “Installing an Audit Provider” section in Managing Security.

The LogAuditProvider example is available in the `examples.security.audit` package in the `/samples/examples/security/audit` directory provided by WebLogic Server. The example writes all events it receives in the WebLogic Server log file. It also defines filter methods for each event type, and calls those filters to decide whether to log a particular event. In the example code, the filter methods always return `true` so that all events are logged. If you extend this example, you can override the filter methods with methods that select the events you want to log. If you want to take some action other than logging, you can use the LogAuditProvider example as a starting point for creating your own provider.

Filtering Network Connections

Passwords, ACLs, and digital certificates allow you to secure WebLogic Server resources using some characteristic of a user. You can add an additional layer of security by filtering network connections. For example, you can deny any non-SSL connections originating outside of your corporate network.

To filter network connections, create a class that implements the `weblogic.security.net.ConnectionFilter` interface and install the class in WebLogic Server so that you can examine requests as they occur and then accept or deny them.

Before you can use a Connection Filter, you need to install the class through the Administration Console. For more information, see the “Installing a Connection Filter” section in Managing Security.

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 ConnectionFilter class. The `ConnectionEvent` object includes the remote IP address (in the form of `java.net.InetAddress`), the remote port number, the port number of the local WebLogic Server, and a string specifying the protocol (HTTP, HTTPS, T3, T3S, or IIOP).

Your `ConnectionFilter` class can examine the `ConnectionEvent` object and accept the connection by returning, or deny the connection by throwing a `FilterException`.

The `examples.security.net.SimpleConnectionFilter` example included in the `samples/examples/security/net` directory provided by WebLogic Server filters connections using a rules file. The `SimpleConnectionFilter` example parses the rules file and sets up a rule-matching algorithm so that connection filtering adds minimal overhead to a WebLogic Server connection. The `SimpleConnectionFilter` example is an efficient, generalized connection filter. If necessary, you can modify this code. 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.

In Listing 4-21, 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.

Listing 4-21 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 RMI over IIOP over SSL

The SSL protocol can be used to protect IIOP connections to RMI or EJB remote objects. The SSL protocol secures connections through authentication and encrypts the data exchanged between objects. You can use RMI over IIOP over SSL in WebLogic Server in the following ways:

- With a CORBA client Object Request Broker (ORB)
- With a Java client

In either case, you need to configure WebLogic Server to use the SSL protocol. For more information, see Configuring the SSL Protocol.

To use RMI over IIOP over SSL with a CORBA client ORB, do the following:

1. Configure the CORBA client ORB to use the SSL protocol. Refer to the product documentation for your client ORB for information about configuring the SSL protocol.
2. Use the `host2ior` utility to print the WebLogic Server IOR to the console. The `host2ior` utility prints two versions of the IOR, one for SSL connections and one for non-SSL connections.
3. Use the SSL IOR when obtaining the initial reference to the CosNaming service that accesses the WebLogic Server JNDI tree.

For more information about using RMI over IIOP, see Programming WebLogic RMI Over IIOP.

To use RMI over IIOP over SSL with a Java client, do the following:

1. If you want to use callbacks, obtain a private key and digital certificate for the Java client.
2. Extend the `java.rmi.server.RMISocketFactory` class to handle SSL socket connections. Be sure to specify the port on which WebLogic Server listens for SSL connections. For an example of a class that extends the `java.rmi.server.RMISocketFactory` class, see Listing 4-22.
3. Run the ejbc compiler with the `-d` option.
4. Add your extension of the `java.rmi.server.RMISocketFactory` class to the `CLASSPATH` of the Java client.
5. Use the following command options when starting the Java client:

```
-xbootclasspath/a:%CLASSPATH%
-Dorg.omg.CORBA.ORBSocketFactoryClass=implementation of
java.rmi.server.RMISocketFactory
-Dssl.certs=directory location of digital certificate for Java
client
-Dssl.key=directory location of private key for Java client
```

The Java client needs to have the classes that WebLogic Server uses for the SSL protocol included in its `CLASSPATH`.

For incoming connections (from WebLogic Server to the Java client for the purpose of callbacks), you need to specify a digital certificate and private key for the Java client on the command line. Use the `ssl.certs` and `ssl.key` command-line options to provide this information. The Java client in Listing 4-22 uses the SSL libraries in WebLogic Server to provide the SSL socket. Alternatively, you can use SSL provider such as Sun Microsystem Inc.’s JSSE as the SSL socket.

Listing 4-22 Example of java.rmi.server.RMISocketFactory

```
package examples.rmi_iiop.ejb.rmi_iiop;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.rmi.server.RMISocketFactory;
import java.util.StringTokenizer;
import java.util.Vector;
import weblogic.security.PEMInputStream;
import weblogic.security.RSAPrivateKey;
import weblogic.security.SSL.SSLCertificate;
import weblogic.security.SSL.SSLParams;
import weblogic.security.SSL.SSLServerSocket;
import weblogic.security.SSL.SSLSocket;
import weblogic.security.X509;

/**
*To use the SSL protocol , set the
*org.omg.CORBA.ORBSocketFactoryClass system property to
*examples.rmi_iiop.ejb.rmi_iiop.SSLSocketFactory.
*Since WebLogic Server may need to talk to the Java client
*(for example, when the Java client exports remote objects that
*WebLogic Server must call), it may be necessary to provide an SSL
*private key and digital certificate so that WebLogic Server can
*establish an SSL connection with the Java client*/

public class SSLSocketFactory extends RMISocketFactory

{
    static int sslPort = 7002;

    SSLCertificate cert;
    RSAPrivateKey key;

    private static InputStream getDERStream(String fileName)
    throws IOException
    {
      InputStream is = new FileInputStream(fileName);

      if (fileName.toLowerCase().endsWith(“.pem”)) {
         is = new PEMInputStream(is);
    }

    return is;
    }

    public SSLSocketFactory()
    {
      String certFiles = System.getProperty(“ssl.certs”);
      String keyFile = System.getProperty(“ssl.key”);

      if (certFiles == null) {
      System.err.println(“Warning: no server certs (ssl.certs)
      provided!”);
      System.err.println(“Warning: incoming server connections
      may fail!”);
      return;
    }
```

```
  if (keyFile == null) {
  System.err.println(“Warning: no server private key (ssl.key)
  provided!”);
  System.err.println(“Warning: incoming server connections
  may fail!”);
  }

  StringTokenizer toks = new StringTokenizer(certFiles,
                       System.getProperty(“path.separator”,
                        “,”));
  cert = new SSLCertificate();
  cert.certificateList = new Vector();

  try {
    if (keyFile != null) {
    cert.privateKey = new
                      RSAPrivateKey(getDERStream(keyFile));
  }

     while (toks.hasMoreTokens()) {
     InputStream is = getDERStream(toks.nextToken());
     cert.certificateList.addElement(new X509(is));
     is.close();
     }
  }

  catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
  }
}

public Socket createSocket(String host, int port)
  throws IOException
{
  Socket sock = null;

  System.out.println(“*** connecting to “ + host + “:” + port);

  if (port == sslPort) {
     try {
     SSLParams p = new SSLParams();

     sock = new SSLSocket(host, port, p);
     }
     catch (Exception e) {
     e.printStackTrace();
     }
  }
  else {
     sock = new Socket(host, port);
  }

   return sock;
}

  public ServerSocket createServerSocket(int i)
     throws IOException
  {
    ServerSocket sock = null;

     if (true) {
       try {
       SSLParams p = new SSLParams();

     if (cert != null) {
       p.setServerCert(cert);
      }
      else {
      System.err.println
      ("**** Listening for SSL connections without server
       private key or certs!");

      System.err.println
      ("**** THIS MAY CAUSE FAILURES IF THE SERVER
      CONNECTS TO US!");
      }

     sock = new SSLServerSocket(i, p);
     }
     catch (Exception e) {
     e.printStackTrace();
     }
  }
  else {
    sock = new ServerSocket(i);
  }

  int lp = sock.getLocalPort();
  if (i != lp) {
     System.out.println("*** listening on any port -
     got " + lp);
  }
  else {
    System.out.println("*** listening on port " + lp);
  }

  return sock;
}
```

```
}
```