



BEA WebLogic Enterprise

Using Security

WebLogic Enterprise 5.1
Document Edition 5.1
May 2000

Copyright

Copyright © 2000 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, BEA Builder, BEA Jolt, BEA Manager, BEA MessageQ, BEA Tuxedo, BEA TOP END, BEA WebLogic, and ObjectBroker are registered trademarks of BEA Systems, Inc. BEA elink, BEA eSolutions, BEA TAP, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Express, BEA WebLogic Personalization Server, BEA WebLogic Server, Java Enterprise Tuxedo and WebLogic Enterprise Connectivity are trademarks of BEA Systems, Inc.

All other company names may be trademarks of the respective companies with which they are associated.

Using Security

Document Edition	Date	Software Version
5.1	May 2000	BEA WebLogic Enterprise 5.1

Contents

About This Document

What You Need to Know	xiii
e-docs Web Site	xiii
How to Print the Document.....	xiii
Documentation Conventions	xv

Part I. Security Concepts

1. Overview of the WebLogic Enterprise Security Service

The WebLogic Enterprise Security Service	1-1
The WebLogic Enterprise Security Environment	1-5
Single Sign-on in the WebLogic Enterprise Security Environment.....	1-7
WebLogic Enterprise Security SPIs	1-9

2. Introduction to the SSL Technology

The SSL Protocol	2-2
Digital Certificates.....	2-4
Certificate Authority	2-5
Certificate Repositories	2-6
A Public Key Infrastructure.....	2-7
PKCS-5 and PKCS-8 Compliance	2-8
Supported Public Key Algorithms.....	2-9
Supported Symmetric Key Algorithms	2-9
Supported Message Digest Algorithms	2-10
Supported Cipher Suites	2-11
Standards for Digital Certificates	2-12

3. The WebLogic Enterprise Security Features

Authentication	3-1
Using Username/Password Authentication	3-2
How Username/Password Authentication Works	3-2
Development Process for Username/Password Authentication	3-4
Certificate-based Authentication	3-6
How Certificate-based Authentication Works.....	3-7
Requirements for Using Certificate-based Authentication.....	3-8
Development Process for Certificate-based Authentication.....	3-9
Using an Authentication Plug-in	3-12
Authorization	3-12
Auditing	3-14
Link-Level Encryption	3-16
How LLE Works	3-17
Encryption Key Size Negotiation.....	3-18
WSL/WSH Connection Timeout During Initialization.....	3-20
Development Process	3-20
The SSL Protocol.....	3-20
How the SSL Protocol Works	3-21
Requirements for Using the SSL Protocol	3-23
Development Process for the SSL Protocol	3-24
Single Sign-On	3-26
PKI Plug-Ins	3-27
Commonly Asked Questions About the WebLogic Enterprise	
Security Service Features	3-29
Do I Have to Change the Security in an Existing WebLogic Enterprise Application?.....	3-29
Can I Use the SSL Protocol in an Existing WebLogic Enterprise Application?	3-30
When Should I Use Mutual Certificate-based Authentication?	3-31

Part II. Security Administration

4. Managing Public Key Security

Requirements for Using Public Key Security.....	4-1
---	-----

Who Needs Digital Certificates and Private/Private Key Pairs.....	4-2
Requesting a Digital Certificate	4-2
Publishing Certificates in the LDAP Directory Service.....	4-3
Editing the LDAP Search Filter File	4-5
Storing the Private Keys in a Common Location.....	4-6
Defining the Trusted Certificate Authorities.....	4-8
Creating a Peer Rules File.....	4-10

5. Configuring Link-Level Encryption

Understanding min and max Values.....	5-1
Verifying the Installed Version of LLE.....	5-2
Configuring LLE on WebLogic Enterprise Application Links.....	5-3
Configuring LLE on BEA Tuxedo Workstation Client Links	5-3
Configuring LLE on Bridge Links	5-4
Configuring LLE on tlisten Links	5-6
Configuring LLE on Domain Gateway Links	5-6

6. Configuring the SSL Protocol

Setting Parameters for the SSL Protocol.....	6-2
Defining a Port for SSL Network Connections.....	6-2
Enabling Host Matching.....	6-3
Setting the Encryption Strength.....	6-4
Setting the Interval for Session Renegotiation	6-7
Defining Security Parameters for the IIOP Listener/Handler	6-7
Example of Setting Parameters on the ISL System Process.....	6-9
Example of Setting Command-Line Options on the CORBA C++ ORB	6-9
Example of Setting System Properties on the CORBA Java ORB	6-11

7. Configuring Authentication

Configuring the Authentication Server	7-1
Defining Authorized Users.....	7-3
Defining a Security Level.....	7-6
Configuring Application Password Security	7-7
Configuring Username/Password Authentication	7-8
Sample UBBCONFIG File for Username/Password Authentication.....	7-9
Configuring Certificate-based Authentication	7-11

Sample UBBCONFIG File for Certificate-based Authentication	7-13
Configuring Access Control	7-15
Configuring Optional ACL Security	7-15
Configuring Mandatory ACL Security.....	7-16

8. Configuring Single Sign-on

Single Sign-on with Username/Password Authentication	8-1
Single Sign-on with Username/Password Authentication and the SSL Protocol.....	8-2
Single Sign-on with the SSL Protocol and Certificate-Based Authentication	8-4

9. Configuring Security Plug-Ins

Registering the Security Plug-Ins (SPIs)	9-1
--	-----

Part III. Security Programming

10. Writing a WebLogic Enterprise CORBA Application that Implements Security

Understanding the Address Formats of the Bootstrap Object	10-2
Using the Host and Port Address Format	10-4
Using the corbaloc URL Address Format	10-5
Using the corbalocs URL Address Format.....	10-6
Using Username/Password Authentication	10-6
The Security Sample Application.....	10-7
Writing the Client Application	10-8
Using Certificate-based Authentication.....	10-16
The Secure Simpapp Sample Application.....	10-17
Writing the Client Application	10-18
Using the Invocations_Options_Required() Method.....	10-21

11. Building and Running the CORBA Sample Applications

Building and Running the Security Sample Application.....	11-2
Building and Running the Secure Simpapp Sample Application	11-2
Step 1: Copy the Files for the Secure Simpapp Sample Application into a Work Directory	11-3

Step 2: Change the Protection Attribute on the Files for the Secure Simpapp Sample Application.....	11-7
Step 3: Verify the Settings of the Environment Variables	11-7
Step 4: Execute the runme Command	11-10
Using the Secure Simpapp Sample Application	11-15

12. Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security

Before You Begin	12-2
How Authentication Works with WebLogic Enterprise EJBs	12-2
Development Steps	12-3
Step 1: Define Security Roles for the Methods of the WebLogic Enterprise EJB	12-3
Step 2: Specify Security Roles in the Deployment Descriptor of the EJB	12-5
Step 3: Define the JNDI Environment Properties	12-6
Step 4: Establish the InitialContext	12-10
Step 5: Use Home to Get a WebLogic Enterprise EJB	12-10
Step 6: Use the getCallerPrincipal Method to Authenticate a WebLogic Enterprise EJB	12-11
Limitations and Restrictions	12-11
Example of Using Security in a WebLogic Enterprise EJB	12-11

13. Troubleshooting

Using ULOGS and ORB Tracing	13-1
CORBA::ORB_init Problems	13-3
Username/Password Authentication Problems	13-4
Certificate-based Authentication Problems	13-5
Tobj::Bootstrap:: resolve_initial_references Problems	13-6
IIOP Listener/Handler Startup Problems	13-7
Configuration Problems	13-8
Problems with Using Callbacks Objects with the SSL Protocol	13-9
Troubleshooting Tips for Digital Certificates	13-9

Part IV. Security Reference

14. WebLogic Enterprise Security Service APIs

The WebLogic Enterprise Security Model	14-2
Authentication of Principals	14-2
Controlling Access to Objects	14-3
Administrative Control	14-3
Functional Components of the WebLogic Enterprise Security Service	14-3
The Principal Authenticator Object	14-5
Using the Principal Authenticator Object with Certificate-based Authentication	14-6
WLE Extensions to the Principal Authenticator Object	14-6
The Credentials Object	14-7
The SecurityCurrent Object	14-9

15. Security Modules

CORBA Module	15-2
TimeBase Module	15-2
Security Module	15-4
Security Level 1 Module	15-6
Security Level 2 Module	15-7
Tobj Module	15-8

16. C++ Security Reference

SecurityLevel1::Current::get_attributes	16-2
SecurityLevel2::Current::authenticate	16-3
SecurityLevel2::Current::set_credentials	16-6
SecurityLevel2::Current::get_credentials	16-7
SecurityLevel2::Current::principal_authenticator	16-8
SecurityLevel2::Credentials	16-9
SecurityLevel2::Credentials::get_attributes	16-11
SecurityLevel2::Credentials::invocation_options_supported	16-12
SecurityLevel2::Credentials::invocation_options_required	16-14
SecurityLevel2::Credentials::is_valid	16-16
SecurityLevel2::PrincipalAuthenticator	16-17
SecurityLevel2::PrincipalAuthenticator::continue_authentication ..	16-19
Tobj::PrincipalAuthenticator::get_auth_type	16-20

Tobj::PrincipalAuthenticator::logon	16-22
Tobj::PrincipalAuthenticator::logoff	16-24
Tobj::PrincipalAuthenticator::build_auth_data	16-25

17. Java Security Reference

18. Automation Security Reference

Method Descriptions	18-2
DISecurityLevel2_Current	18-2
DISecurityLevel2_Current.get_attributes	18-3
DISecurityLevel2_Current.set_credentials	18-4
DISecurityLevel2_Current.get_credentials.....	18-5
DISecurityLevel2_Current.principal_authenticator.....	18-6
DITobj_PrincipalAuthenticator.....	18-7
DITobj_PrincipalAuthenticator.authenticate	18-8
DITobj_PrincipalAuthenticator.build_auth_data.....	18-10
DITobj_PrincipalAuthenticator.continue_authentication	18-12
DITobj_PrincipalAuthenticator.get_auth_type	18-13
DITobj_PrincipalAuthenticator.logon	18-15
DITobj_PrincipalAuthenticator.logoff.....	18-17
DISecurityLevel2_Credentials	18-17
DISecurityLevel2_Credentials.get_attributes	18-18
DISecurityLevel2_Credentials.is_valid	18-19
Programming Example.....	18-20

Index



About This Document

This document provides an introduction to concepts associated with the BEA WebLogic Enterprise™ security features, a description of how to secure your WebLogic Enterprise applications using the WebLogic Enterprise security features, and a guide to the use of the application programming interfaces (APIs) in the WebLogic Enterprise Security Service.

This document covers the following topics:

- Chapter 1, “Overview of the WebLogic Enterprise Security Service,” presents an overview of the features in the WebLogic Enterprise Security Service.
- Chapter 2, “Introduction to the SSL Technology,” introduces the concepts associated with a Public Key Infrastructure (PKI).
- Chapter 3, “The WebLogic Enterprise Security Features,” presents an indepth discussion of the features in the WebLogic Enterprise Security Service and describes the development and administration processes needed to implement the features.
- Chapter 4, “Managing Public Key Security,” describes how to set up a public key infrastructure to interact with WebLogic Enterprise applications that use the Secure Sockets Layer (SSL) protocol and certificate-based authentication.
- Chapter 5, “Configuring Link-Level Encryption,” describes setting parameters in the UBBCONFIG file for Link Level Encryption (LLE).
- Chapter 6, “Configuring the SSL Protocol,” describes configuring the IIOP Listener/Handler, the CORBA C++ ORB, or the CORBA Java ORB so that it can be used with the Secure Sockets Layer (SSL) protocol and certificate-based authentication.
- Chapter 7, “Configuring Authentication,” explains the configuration tasks required when using authentication in a WebLogic Enterprise application.

-
- Chapter 8, “Configuring Single Sign-on,” explains the configuration tasks required when using trusted connection pools in a WebLogic Enterprise application.
 - Chapter 9, “Configuring Security Plug-Ins,” explains how to register Security Plug-Ins in the WebLogic Enterprise environment.
 - Chapter 10, “Writing a WebLogic Enterprise CORBA Application that Implements Security,” explains how the bootstrapping options work and describes implementing password-based authentication and certificate-based authentication in WebLogic Enterprise CORBA applications.
 - Chapter 11, “Building and Running the CORBA Sample Applications,” describes how to build and run the Security and Secure Simpapp sample applications.
 - Chapter 12, “Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security,” describes implementing password-based and certificate-based authentication in WebLogic Enterprise EJBs.
 - Chapter 13, “Troubleshooting,” provides troubleshooting tips that can be used when solving problems that occur with the security portion of a WebLogic Enterprise application.
 - Chapter 14, “WebLogic Enterprise Security Service APIs,” introduces the WebLogic Enterprise security model and the functional components of the security model.
 - Chapter 15, “Security Modules,” includes the Object Management Group (OMG) Interface Definition Language (IDL) for the modules used by the WebLogic Enterprise Security service.
 - Chapter 16, “C++ Security Reference,” includes the C++ method descriptions.
 - Chapter 17, “Java Security Reference,” includes the Java method descriptions.
 - Chapter 18, “Automation Security Reference,” includes the Automation method descriptions.

What You Need to Know

This document is intended for programmers who want to incorporate security into their WebLogic Enterprise applications and system administrators who are responsible for setting up and maintaining the security infrastructure in an enterprise.

e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA Systems, Inc. corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the “e-docs” Product Documentation page at <http://e-docs.beasys.com>.

How to Print the Document

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

A PDF version of this document is available on the WebLogic Enterprise 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 Enterprise documentation Home page, click the PDF Files button, and select the document you want to print.

If you do not have Adobe Acrobat Reader installed, you can download it for free from the Adobe Web site at <http://www.adobe.com/>.

Related Information

For more information about CORBA, Java 2 Enterprise Edition (J2EE), BEA Tuxedo®, distributed object computing, transaction processing, C++ programming, and Java programming, see the *WebLogic Enterprise Bibliography* in the WebLogic Enterprise online documentation.

Contact Us!

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

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.1 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSUPPORT at www.beasys.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	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
<i>italics</i>	Indicates emphasis or book titles.
monospace text	<p>Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames and their extensions. Monospace text also indicates text that you must enter from the keyboard.</p> <p><i>Examples:</i></p> <pre>#include <iostream.h> void main () the pointer psz chmod u+w * \tux\data\ap .doc tux.doc BITMAP float</pre>
monospace boldface text	<p>Identifies significant words in code.</p> <p><i>Example:</i></p> <pre>void commit ()</pre>
<i>monospace italic text</i>	<p>Identifies variables in code.</p> <p><i>Example:</i></p> <pre>String <i>expr</i></pre>
UPPERCASE TEXT	<p>Indicates device names, environment variables, and logical operators.</p> <p><i>Examples:</i></p> <pre>LPT1 SIGNON OR</pre>

Convention	Item
{ }	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
[]	Indicates optional items in a syntax line. The brackets themselves should never be typed. <i>Example:</i> buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...
	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
...	Indicates one of the following in a command line: <ul style="list-style-type: none">■ That an argument can be repeated several times in a command line■ That the statement omits additional optional arguments■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. <i>Example:</i> buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...
.	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Part I **Security Concepts**

Chapter 1. Overview of the WebLogic Enterprise
 Security Service

Chapter 2. Introduction to the SSL Technology

Chapter 3. The WebLogic Enterprise Security Features

1 Overview of the WebLogic Enterprise Security Service

This topic includes the following sections:

- The WebLogic Enterprise Security Service
- The WebLogic Enterprise Security Environment
- Single Sign-On in the WebLogic Enterprise Security Environment
- WebLogic Enterprise Security SPIs

The WebLogic Enterprise Security Service

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, where the proof material is a secret word or phrase that gives a user access to a particular program or system, and data encryption is the translation of data into a form that cannot be interpreted.

1 *Overview of the WebLogic Enterprise Security Service*

Distributed applications such as those used for electronic commerce (e-commerce) offer many access points for malicious people to intercept data, disrupt operations, or generate fraudulent input; the more distributed a business becomes, the more vulnerable it is to attack. Thus, the distributed computing software, or middleware, upon which such applications are built must provide security.

The Security Service of the WebLogic Enterprise product lets you establish secure connections between client and server applications. It has the following features:

- Authentication of CORBA C++ client applications, CORBA Java client applications, RMI client applications and EJBs to the WebLogic Enterprise domain. Authentication can be accomplished using a standard username/password combination or certificate-based authentication where an entity is authenticated using the identity inside of the X.509 digital certificate provided to the server applications.
- Data integrity and confidentiality through Link Level Encryption (LLE) or the Secure Sockets Layer (SSL) protocol. CORBA C++, CORBA Java, RMI client applications and EJBs can establish SSL sessions with a WebLogic Enterprise domain. Tuxedo client applications can use LLE to protect network traffic between bridges and domains.
- A single sign-on environment between the WebLogic Server™ and WebLogic Enterprise environments using WebLogic Enterprise Connectivity. This feature allows the propagation of security information about the requesting WLS principal to the WebLogic Enterprise domain over network connections that are part of a trusted connection pool.
- Security Service Provider Interfaces (SPIs) that can be used to integrate security mechanisms that provide authentication, authorization, auditing, and public key security features. Security vendors can use the SPIs to integrate third-party security offerings into the WebLogic Enterprise environment.
- A Public Key Infrastructure (PKI) that uses the SSL protocol and X.509 digital certificates to provide data privacy for messages sent over network links. In addition, a set of PKI SPIs are provided.

To access the full features of the WebLogic Enterprise Security Service, you need to install the WebLogic Enterprise Security Pack. The WebLogic Enterprise Security Pack contains the files necessary to enable the use of the SSL protocol, LLE, and PKI. For information about installing the WebLogic Enterprise Security Pack, see the [BEA WebLogic Enterprise Installation Guide](#).

Note: *Using Security* describes the security features of the T-Engine environment in the WebLogic Enterprise product. For a complete description of using the security features in the J-Engine environment in the WebLogic Enterprise product, see [Using WebLogic SSL](#) and [Using WebLogic ACLs](#) in the WebLogic Server portion of the online documentation. For a description of the T-Engine and J-Engine environments of the WebLogic Enterprise product, see [Getting Started](#).

Table 1-1 summarizes the features in the Security Service of the WebLogic Enterprise product.

Table 1-1 WebLogic Enterprise Security Service Features

Security Features	Description	SPI	Default Implementation
Authentication	Proves the stated identity of users or system processes; safely remembers and transports identity information; and makes identity information available when needed.	Implemented as a single interface	Provides security at three levels: no authentication, application password, and username/password authentication. Certificate-based authentication is available by installing the WebLogic Enterprise Security Pack.
Authorization	Controls access to resources based on identity or other information.	Implemented as a single interface	N/A
Auditing	Safely collects, stores, and distributes information about operating requests and their outcomes.	Implemented as a single interface	Default auditing security is implemented via the features of the user log (ULOG).
Link-Level Encryption	Uses symmetric key encryption to establish data privacy for messages moving over the network links that connect the machines in a WebLogic Enterprise application.	N/A	RC4 symmetric key encryption.

1 Overview of the WebLogic Enterprise Security Service

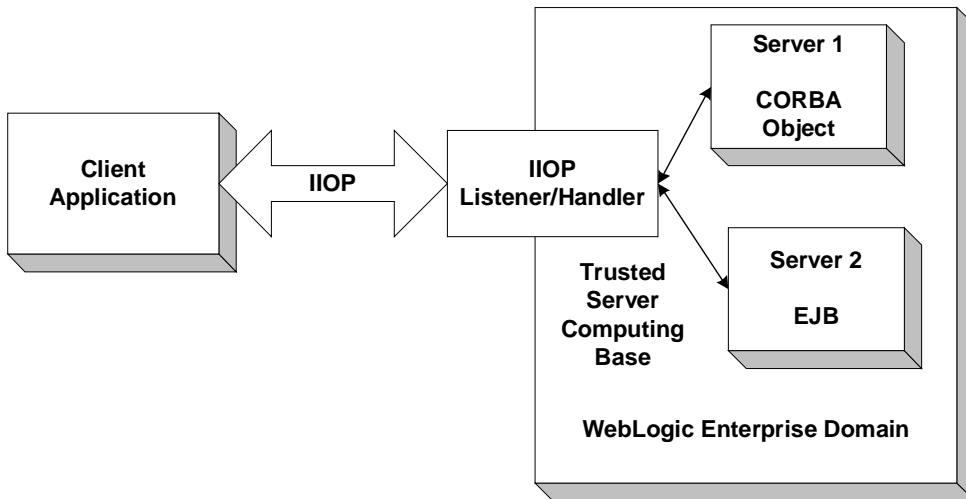
Table 1-1 WebLogic Enterprise Security Service Features (Continued)

Security Features	Description	SPI	Default Implementation
SSL protocol	Uses asymmetric encryption to establish data privacy for messages moving over network links between WebLogic Enterprise domains.	N/A	The SSL version 3.0 protocol is only available by installing the WebLogic Enterprise Security Pack.
Single sign-on	Propagates the security identity of a WebLogic Server client identity to a WebLogic Enterprise using WebLogic Connectivity.	N/A	N/A
Public key security	Uses public key (or asymmetric key) encryption to establish data privacy for messages moving over the network links between remote client applications and the IIOP Listener/Handler. Complies with SSL version 3 allowing mutual authentication based on X.509 digital certificates.	Implemented as the following interfaces: <ul style="list-style-type: none">■ Public key initialization■ Key management■ Certificate lookup■ Certificate parsing■ Certificate validation■ Proof material mapping	Default public key security supports the following algorithms: <ul style="list-style-type: none">■ RSA for key exchange.■ DES and its variants RC2 and RC4 for bulk encryption.■ MD5 and SHA for message digests.

The WebLogic Enterprise Security Environment

Direct end-to-end mutual authentication in a distributed enterprise middleware environment such as the WebLogic Enterprise environment can be prohibitively expensive, especially when accomplished through security mechanisms optimized for long duration connections. It is not efficient for principals to establish direct network connections with each server application, nor is it practical to exchange and verify multiple authentication messages as part of process each service request. Instead, the WebLogic Enterprise product implements a delegated trust authentication model as shown in Figure 1-1.

Figure 1-1 Delegated Trust Model



1 *Overview of the WebLogic Enterprise Security Service*

In a delegated trust model, principals (generally users of client applications) authenticate to a trusted system gateway process. In the case of the WebLogic Enterprise product, the trusted system gateway process is the IIOP Listener/Handler. As part of successful authentication, security tokens are assigned to the initiating principal. A security token is an opaque data structure suitable for transfer between processes.

When a request from an authenticated principal reaches the IIOP Listener/Handler, the IIOP Listener/Handler attaches the principal's security tokens to the request and delivers the request to the target server application for authorization and auditing purposes.

In a delegated trust authentication model, the IIOP Listener/Handler trusts that the authentication software in the WebLogic Enterprise domain will verify the identity of the principal and generates the appropriate security tokens. Server applications, in turn, trust that the IIOP Listener/Handler will attach the correct security tokens. Server applications also trust that any other server applications involved in the process of a request from a principal will safely deliver the security tokens.

A session is established between the initiating client application and the IIOP Listener/Handler in the following way:

1. When a client application wants to access an object within a WebLogic Enterprise domain, the client application uses either a username and password or a X.509 digital certificate to authenticate over the connection with the IIOP Listener/Handler.
2. A security association called a security context is established between a principal and the IIOP Listener/Handler. This security context is used to control access to objects in the WebLogic Enterprise domain.

The IIOP Listener/Handler retrieves the authorization and auditing tokens from the security context. Together, the authorization and auditing tokens represent the principal's identity associated with the security context.

3. Once the authentication process is complete, the principal invokes an object in the WebLogic Enterprise domain. The request is packaged into an IIOP request and forwarded to the IIOP Listener/Handler. The IIOP Listener/Handler associates the request with the previously established security context.
4. The IIOP Listener/Handler receives the request from the initiating principal.

The protection of messages between the client application and the IIOP Listener/Handler is dependent on the security technology used in the WebLogic

Enterprise application. The default behavior of the WebLogic Enterprise product is to encrypt the authentication information but not to protect the message sent between the client application and the WebLogic Enterprise domain. The message is sent in clear text. The SSL protocol can be used to protect the message. If the SSL protocol is configured to protect messages for integrity and confidentiality, the request is digitally signed and sealed (encrypted) before it is sent to the IIOP Listener/Handler.

5. The IIOP Listener/Handler forwards the request along with the authorization and auditing tokens of the initiating principal to the appropriate server application.
6. When the request is received by the server application, the WebLogic Enterprise system interrogates the forwarded tokens of the requesting principal to determine if the request should be processed or denied. The WebLogic Enterprise Security Service will, based on the decision of the authorization implementation integrated in the WebLogic Enterprise environment, deny the processing of any request on an object for which the requesting principal has no permission to access.

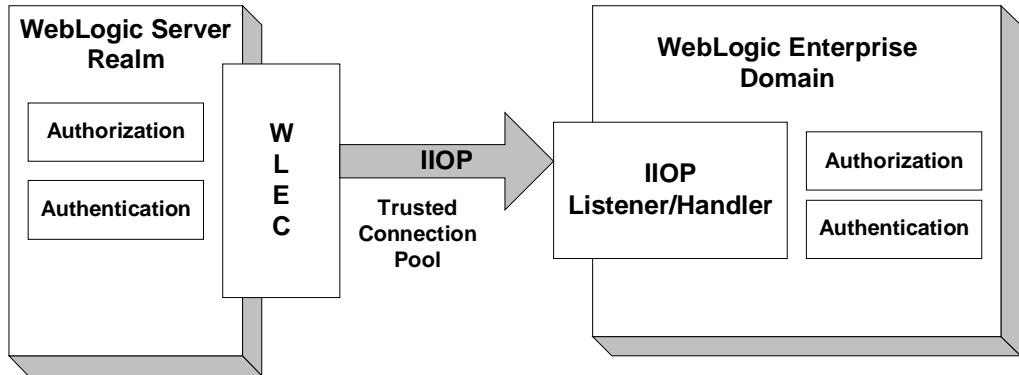
Single Sign-On in the WebLogic Enterprise Security Environment

A WebLogic Server realm and a WebLogic Enterprise domain are considered separate scopes of security definitions. Each contains its own security database of users and access control. However, by using WebLogic Enterprise Connectivity, the identity of a principal authenticated in a WebLogic Server realm can be presented and used to form the identity of an authenticated principal in a WebLogic Enterprise domain over a connection that is part of a trusted pool of connections.

Note: The single sign-on functionality in the WebLogic Enterprise product is unidirectional. You can only propagate a principal's identity from the WebLogic Server realm to the WebLogic Enterprise domain.

Figure 1-2 illustrates how single sign-on works in the WebLogic Enterprise Security Environment.

Figure 1-2 Single Sign-On in the WebLogic Enterprise Security Environment



When using single sign-on, the security identity of a WebLogic Server principal is propagated as part of the service context of a I I O P request sent to the WebLogic Enterprise domain over a network connection that is part of a trusted connection pool. Each network connection in a trusted connection pool has been authenticated using a defined principal identity. Both username/password and certificate-based authentication can be used to establish a trusted connection pool.

The propagated security identity is used by the I I O P Listener/Handler to impersonate a principal identity in the WebLogic Enterprise 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 server application in the WebLogic Enterprise domain where they are used for authorization and auditing purposes.

To facilitate the mapping of principal identities, the I I O P Listener/Handler uses an authentication plug-in. This plug-in is responsible for mapping the principal identity into the authorization and auditing tokens. These tokens are propagated as part of the request being forwarded to the target EJB or CORBA object. The target EJB or CORBA object can then use these tokens to determine information about the initiator of the request, including the identity of the principal and the role/group name with which the principal is associated.

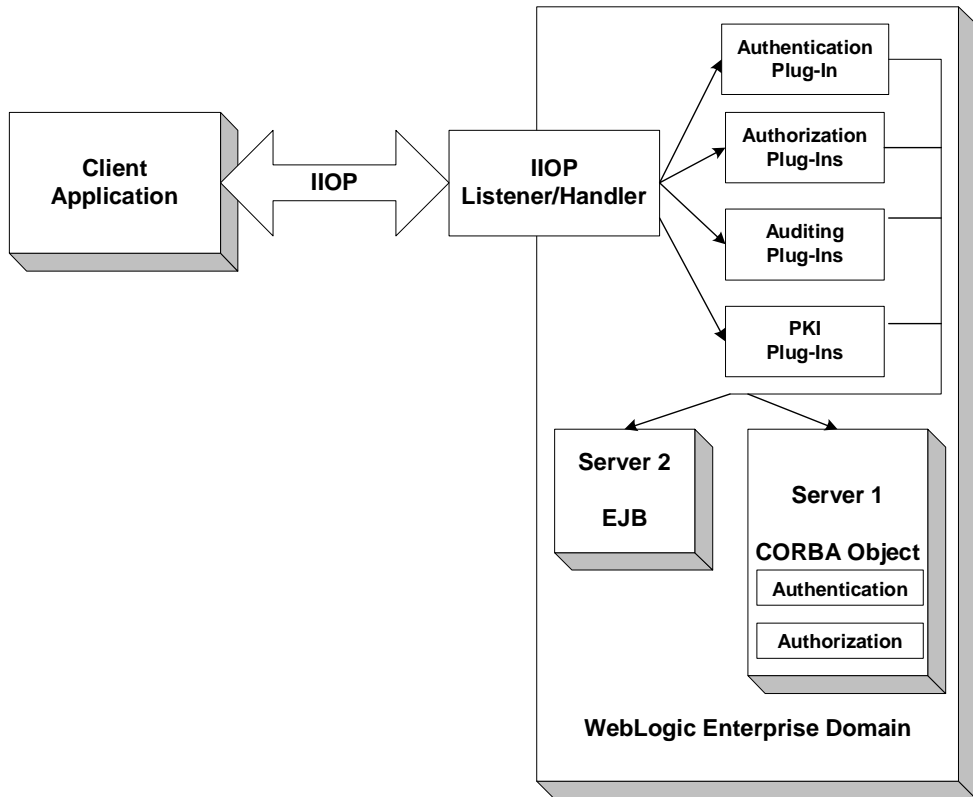
The SSL protocol can be used to protect the confidentiality and integrity of the request from the WebLogic Server realm. SSL encryption is provided for IIOP requests to CORBA objects in the WebLogic Enterprise domain and RMI on IIOP requests to EJBs and RMI objects. In order to protect the request, both WebLogic Connectivity and the WebLogic Enterprise application must be configured to use the SSL protocol.

For information about implementing single sign-on, see Chapter 8, “Configuring Single Sign-on.”

WebLogic Enterprise Security SPIs

As shown in Figure 1-3, the authentication, authorization, auditing, and public key security features available with the WebLogic Enterprise product are implemented through a plug-in interface, which allows security plug-ins to be integrated into the WebLogic Enterprise environment. A security plug-in is a code module that implements a particular security feature.

Figure 1-3 Architecture for the WebLogic Enterprise Security Service Provider Interfaces



The WebLogic Enterprise Security Service provides interfaces for the types of security plug-ins listed in Table 1-1.

Table 1-2 The WebLogic Enterprise Security Plug-Ins

Plug-In	Description
Authentication	Allows communicating processes to mutually prove identification.
Authorization	Allows system administrators to control access to WebLogic Enterprise applications. Specifically, an administrator can use authorization to allow or disallow principals to use resources or services provided by a WebLogic Enterprise application.
Auditing	Provides a means to collect, store, and distribute information about operating requests and their outcomes. Audit-trail records may be used to determine which principals performed, or attempted to perform, actions that violated the configured security policies of a WebLogic Enterprise application. They may also be used to determine which operations were attempted, which ones failed, and which ones successfully completed.
Public key initialization	Allows public key software to open public and private keys. For example, gateway processes may need to have access to a specific private key in order to decrypt messages before routing them.
Key management	Allows public key software to manage and use public and private keys. Note that message digests and session keys are encrypted and decrypted using this interface, but no bulk data encryption is performed using public key cryptography. Bulk data encryption is performed using symmetric key cryptography.
Certificate lookup	Allows public key software to retrieve X.509v3 digital certificates for a given principal. Digital certificates may be stored using any appropriate certificate repository, such as Lightweight Directory Access Protocol (LDAP).

Table 1-2 The WebLogic Enterprise Security Plug-Ins (Continued)

Plug-In	Description
Certificate parsing	Allows public key software to associate a simple principal name with an X.509v3 digital certificate. The parser analyzes a digital certificate to generate a principal name to be associated with the digital certificate.
Certificate validation	Allows public key software to validate an X.509v3 digital certificate in accordance with specific business logic.
Proof material mapping	Allows public key software to access the proof materials needed to open keys, provide authorization tokens, and provide auditing tokens.

The specifications for the SPIs are currently only available to third-party security vendors who have entered into a special agreement with BEA Systems, Inc. Customers who want to customize a security feature must contact one of these vendors or BEA Professional Services. For example, a BEA customer who wants a custom implementation of public key security must contact a third-party vendor who can provide the appropriate security plug-in or BEA Professional Services.

For more information about security plug-ins, including installation and configuration procedures, see your BEA account executive.

2 Introduction to the SSL Technology

This topic includes the following sections:

- The SSL Protocol
- Digital Certificates
- Certificate Authority
- Certificate Repositories
- A Public Key Infrastructure
- PKCS-5 and PKCS-8 Compliance
- Supported Public Key Algorithms
- Supported Symmetric Key Algorithms
- Supported Message Digest Algorithms
- Supported Cipher Suites
- Standards for Digital Certificates

The SSL Protocol

The Secure Sockets Layer (SSL) protocol allows you to integrate these essential features into your WebLogic Enterprise application:

- Confidentiality

Confidentiality is the ability to keep communications secret from parties other than the intended recipient. It is achieved by encrypting data with strong algorithms. The SSL protocol provides a secure mechanism that enables two communicating parties to negotiate the strongest algorithm they both support and to agree on the keys with which to encrypt the data.

- Integrity

Integrity is a guarantee that the data being transferred has not been modified in transit. The same handshake mechanism which allows the two parties to agree on algorithms and keys also allows the two ends of an SSL connection to establish shared data integrity secrets which are used to ensure that when data is received any modifications will be detected.

- Authentication

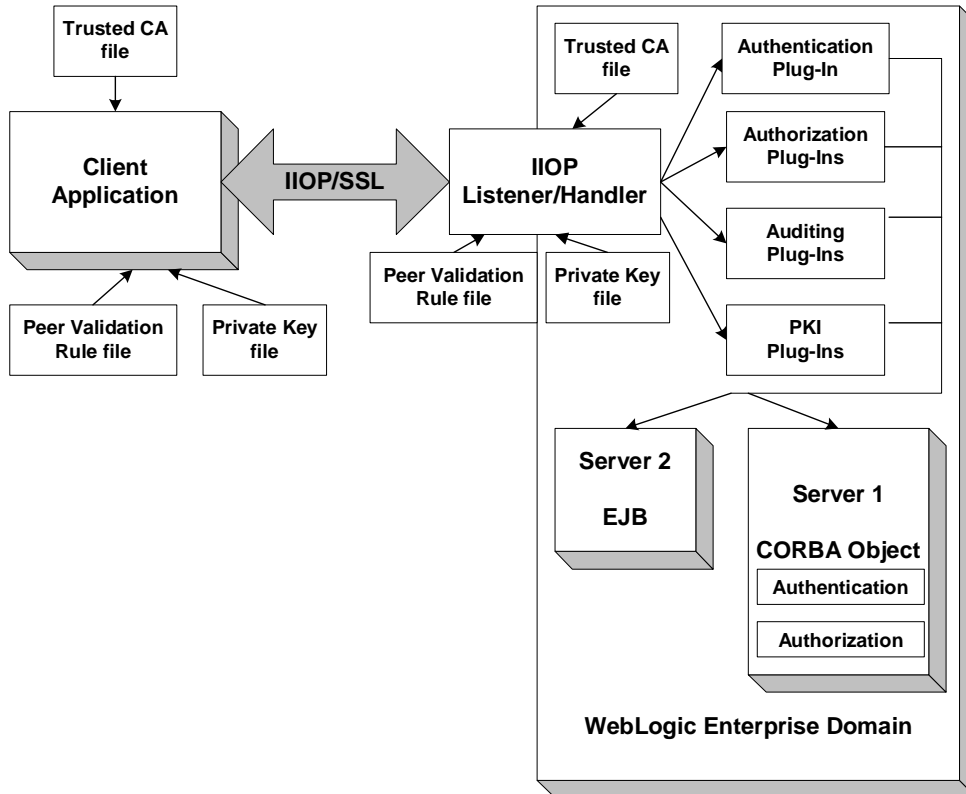
Authentication is the ability to ascertain with whom you are speaking. By using digital certificates and public key security, the client and server applications can each be authenticated to the other. This allows the two parties to be certain they are communicating with someone they trust. The SSL protocol provides a mechanism that can be used to authenticate principals to a WebLogic Enterprise domain using X.509 digital certificates. The use of certificate-based authentication can be used as an alternative to username/password authentication.

The SSL protocol provides secure connections by allowing two applications connecting over a network connection to authenticate the other's identity and by encrypting the data exchanged between the applications. When using the SSL protocol, 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 use, and generate encryption keys used for the remainder of the session.

The SSL protocol uses public key encryption for authentication. With public key encryption, a public key and private key are generated for a principal or other entity such as the IIOP Listener/Handler or an application server. The keys are related such that the data encrypted with the public key can only be decrypted using the corresponding private key. The private key is carefully protected so that only the owner can decrypt messages. The public key, however, is distributed freely so that anyone can encrypt messages intended for the owner.

Figure 2-1 illustrates how the SSL protocol works in the WebLogic Enterprise Security environment.

Figure 2-1 The SSL Protocol in the WebLogic Enterprise Security Environment



When using the SSL protocol in the WebLogic Enterprise Security environment, the IIOP Listener/Handler authenticates itself to initiating principals. The IIOP Listener/Handler presents its digital certificate to the initiating principal. To successfully negotiate a SSL connection, the client application must then authenticate the IIOP Listener/Handler but the IIOP Listener/Handler will accept any client application into the SSL connection. This type of authentication is referred to as *server authentication*.

When using server authentication, the initiating client application is required to have digital certificates for certificate authorities that are to be trusted. The IIOP Listener/Handler must have a private key and digital certificates that represents its identity. Server authentication is common on the Internet where customers want to create secure connections before they share personal data. In this case, the client application has a similar role to that of a Web browser.

With SSL version 3.0, principals can also authenticate to the IIOP Listener/Handler. This type of authentication is referred to as *mutual authentication*. In mutual authentication, the principal presents their digital certificates to the IIOP Listener/Handler. When using mutual authentication, both the IIOP Listener/Handler and the principal need private keys and digital certificates that represent their identity. This type of authentication is useful when you must restrict access to trusted principals only.

The SSL protocol is only available in the WebLogic Enterprise product by installing the WebLogic Enterprise Security Pack. The WebLogic Enterprise Security Pack provides the SSL protocol and the infrastructure needed to use digital certificates. For information about installing the WebLogic Enterprise Security Pack, see the [BEA WebLogic Enterprise Installation Guide](#).

Digital Certificates

Digital certificates are electronic documents used to uniquely identify principals and entities over networks such as the Internet. A digital certificate securely binds the identity of a principal 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 principal or entity. A recipient of a digital certificate can use the public key contained in the 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 particular subject.

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

- The name of the subject (holder, owner) and other identification information required to uniquely identify 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. Thus, digital certificates can be read or written by any application complying with X.509. The PKI in the WebLogic Enterprise Security Service public key recognizes digital certificates that comply with X.509 version 3, or X.509v3.

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 the detection of tampering. The certificate authority then returns the signed digital certificate to the requesting subject.

The subject can verify the digital 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. The second solution 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 whom 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 the digital signature can be trusted.

Certificate Repositories

To make a public key and its identification with a specific subject readily available for use in verification, the digital certificate may be published in a repository or made available by other means. Certificate repositories are databases of digital certificates and other information available for retrieval and use in verifying digital signatures. Retrieval can be accomplished automatically by directly requesting digital certificates from the repository as needed.

In the WebLogic Enterprise Security service, Lightweight Directory Access Protocol (LDAP) is used as a certificate repository. BEA Systems, Inc. does not provide or recommend any specific LDAP server. The LDAP server you choose should support the X.500 scheme definition and the LDAP version 2 or 3 protocol.

A Public Key Infrastructure

A Public Key Infrastructure (PKI) consists of protocols, services, and standards supporting applications of public key cryptography. Because the technology is still relatively new, the term PKI is somewhat loosely defined: sometimes PKI simply refers to a trust hierarchy based on public key digital certificates; in other contexts, it embraces digital signature and encryption services provided to end-user applications as well.

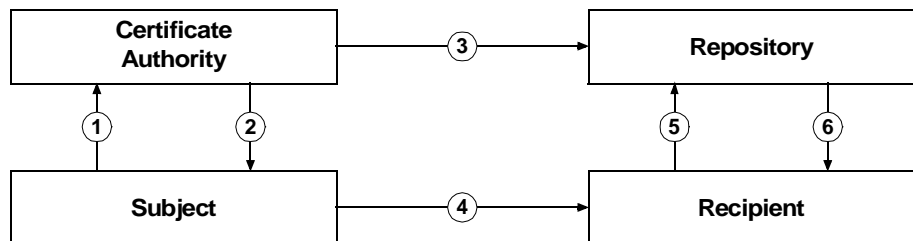
There is no single standard public key infrastructure today, though efforts are underway to define one. It is not yet clear whether a standard will be established or multiple independent PKIs will evolve with varying degrees of interoperability. In this sense, the state of PKI technology today can be viewed as similar to local and wide area (WAN) network technology in the 1980s, before there was widespread connectivity via the Internet.

The following services are likely to be found in a PKI:

- Key registration for issuing a new digital certificate for a public key.
- Certificate revocation for canceling a previously-issued digital certificate and private key.
- Key selection for obtaining a party's public key.
- Trust evaluation for determining whether a digital certificate is valid and which operations it authorizes.

Figure 2-2 shows the PKI process flow.

Figure 2-2 PKI Process Flow



1. The subject applies to a certificate authority for digital certificate.
2. The certificate authority verifies the identity of subject and issues a digital certificate.
3. The certificate authority or the subject publishes the digital certificate in a certificate repository such as LDAP.
4. The subject digitally signs an electronic message with the associated private key to ensure sender authenticity, message integrity, and nonrepudiation, and then sends message to recipient.
5. The recipient retrieves the sender's certificate from the certificate repository and then retrieves the public key from the certificate.

The WebLogic Enterprise product does not provide the tools necessary to be a certificate authority. BEA Systems, Inc. recommends using a third-party certificate authority such as VeriSign or Entrust. By offering a Public Key SPI, BEA Systems, Inc. extends the opportunity to all WebLogic Enterprise customers to use a PKI security solution with the PKI software from their vendor of choice. See [“Single Sign-On” on page 3-26](#) for more information.

PKCS-5 and PKCS-8 Compliance

Informal but recognized industry standards for public key software have been issued by a group of leading communications companies, led by RSA Laboratories. These standards are called “Public-Key Cryptography Standards,” or PKCS. The WebLogic Enterprise product uses PKCS-5 and PKCS-8 to protect the private keys used with the SSL protocol.

- PKCS-5 is a specification of a format for using password-based encryption that uses DES to protect data.
- PKCS-8 is a specification of a format for storing private keys, including the ability to encrypt them with PKCS-5.

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 public key security in the WebLogic Enterprise Security Service also supports digital signature algorithms. Digital signature algorithms are simply public key algorithms used to provide digital signatures.

The WebLogic Enterprise product supports the Rivest, Shamir, and Adelman (RSA) algorithm, the Diffie Hellman algorithm, and Digital Signature Algorithm (DSA). With the exception of DSA, digital signature algorithms can be used for digital signatures and encryption. DSA can be used for digital signatures but not for encryption.

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 symmetric key encryption 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 *plaintext* (unencrypted text) data into a block of *ciphertext* (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 Public key security feature in the WebLogic Enterprise Security Service 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.

- RC2 (Rivest's Cipher 2)

RC2 is a variable key-size block cipher.

- 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 is exportable 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 WebLogic Enterprise public key security restricts the key length to 128 bits.

Customers of the WebLogic Enterprise product cannot expand or modify this list of algorithms.

Supported Message Digest Algorithms

The WebLogic Enterprise Security Service supports the MD5 and SHA-1 (Secure Hash Algorithm 1) message digest algorithms. Both MD5 and SHA-1 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-1 offers more security by using a 160-bit hash, but is slower than MD5.

Supported Cipher Suites

A cipher suite is a SSL encryption method that includes the key exchange algorithm, the symmetric encryption algorithm, and the secure hash algorithm used to protect the integrity of the communication. For example, the cipher suite

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 WebLogic Enterprise Security Service supports the cipher suites described in Table 2-1.

Table 2-1 SSL Cipher Suites Supported by the WebLogic Enterprise Product

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_CDC_SHA	RSA	56
SSL_RSA_EXPORT_WITH_RC4_40_MD5	RSA	40
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	40
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSA	40
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	Diffie Hellman	40
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	Diffie Hellman	40
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	112
SSL_RSA_WITH_NULL_SHA	RSA	0
SSL_RSA_WITH_NULL_MD5	RSA	0

Standards for Digital Certificates

The WebLogic Enterprise Security Service supports the digital certificates that conform to the X.509v3 standard. The X.509v3 standard specifies the format of digital certificates. BEA recommends obtaining certificates from a certificate authority such as Verisign or Entrust.

3 The WebLogic Enterprise Security Features

This topic includes the following sections:

- Authentication
- Authorization
- Auditing
- Link-Level Encryption
- The SSL Protocol
- Single Sign-On
- Commonly Asked Questions About the WebLogic Enterprise Security Features

Authentication

In WebLogic Enterprise product, authentication provides a verified identity to be established for the client application. The WebLogic Enterprise product supports username/password and certificate-based authentication. In addition, a plug-in is provided for authentication so that Privilege Attribute Certificates (PACs) or Kerberos

tickets can be used for authentication in the WebLogic Enterprise environment. The following sections describe the different types of authentication and how they are implemented.

Using Username/Password Authentication

The WebLogic Enterprise product supports a username/password mechanism to provide authentication to existing WebLogic Enterprise applications and to new WebLogic Enterprise applications that are not prepared to deploy a full public key infrastructure (PKI). When using username/password authentication, the applications that initiate invocations on WebLogic Enterprise objects authenticate themselves to the WebLogic Enterprise domain using a defined username and password.

Two levels of Username/Password authentication are provided:

- Application password—in this security scenario, the client application authenticates itself to the WLE domain. An application password is used to authenticate the client application.
- System authentication—in this security scenario, the client application provides a username or client application name and a password that is verified by a WebLogic Enterprise-provided authentication server (AUTHSRV) before access is granted to client application.

Username/Password authentication is available in both the base WebLogic Enterprise product and the WebLogic Enterprise Security Pack. If you install the WebLogic Enterprise Security Pack and choose to use username/password authentication, the SSL protocol can be used to provide confidentiality to communication between client applications. When using Username/Password authentication, you have the option of using the `Tobj::PrincipalAuthenticator::logon()` or the `SecurityLevel2::PrincipalAuthenticator::authenticate()` methods in your client application.

How Username/Password Authentication Works

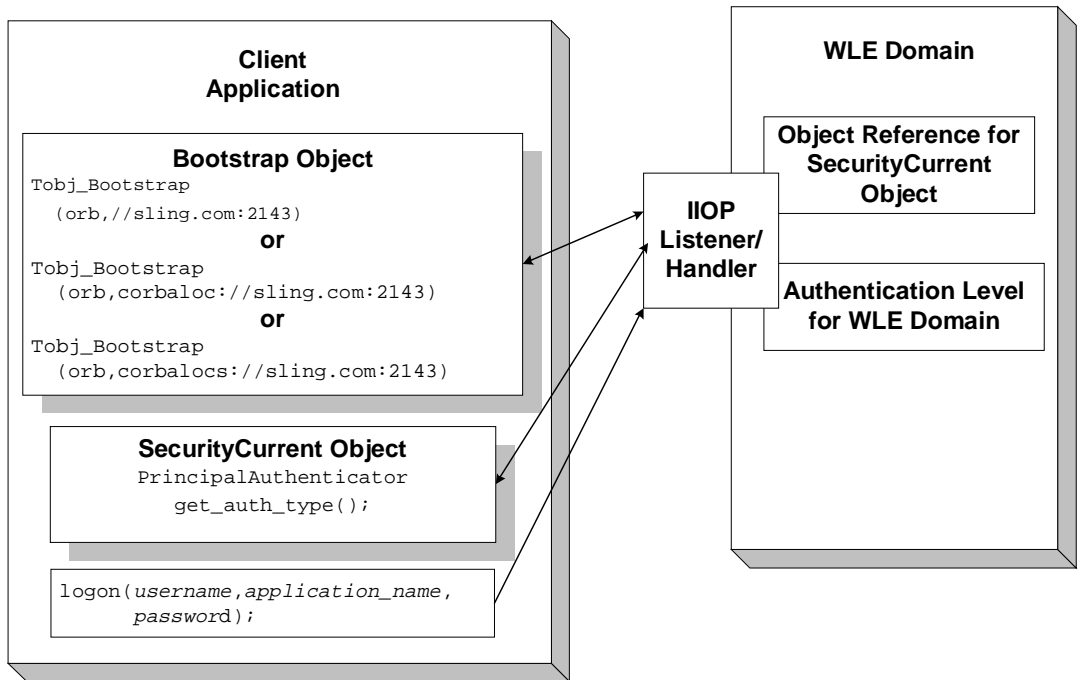
Username/Password authentication works in the following way:

1. The initiating application instantiates the Bootstrap object with any of the URL address formats. For more information about which URL address format to use with the Bootstrap object, see “Understanding the Address Formats of the Bootstrap Object” on page 10-2.
2. The initiating application obtains credentials for the user. The initiating application must provide proof material to be used by the WebLogic Enterprise domain to authenticate the user. This proof material consists of the name of the user and a password.
 - The initiating application creates the security context using a `PrincipalAuthenticator` object. The request for authentication is sent to the IIOP Listener/Handler. The proof material in the authentication request is securely relayed to the authentication server, which verifies the supplied information.
 - If the verification succeeds, the WebLogic Enterprise system constructs a `Credentials` object that is used by all future invocations. The `Credentials` object for the user is associated with the `Current` object that represents the security context.
3. The initiating application invokes a WebLogic Enterprise object in the WebLogic Enterprise domain using an object reference. The request is packaged into an IIOP request and is forwarded to the IIOP Listener/Handler that associates the request with the previously established security context.

If the SSL protocol is used to secure the connection for confidentiality and integrity, the data will also be protected from eavesdropping.
4. The IIOP Listener/Handler receives the request from the initiating application.
5. The IIOP Listener/Handler forwards the request, along with the credentials of the initiating application, to the appropriate WebLogic Enterprise object.

Figure 3-1 illustrates these steps.

Figure 3-1 How Username/Password Authentication Works



Development Process for Username/Password Authentication

Defining Username/Password authentication for a WebLogic Enterprise application includes administration and programming steps. Table 3-1 and Table 3-2 list the administration and programming steps for Username/Password authentication. For a detailed description of the administration steps for Username/Password authentication, see [“Configuring Authentication” on page 7-1](#). For a complete description of the programming steps, see [“Writing a WebLogic Enterprise CORBA Application that Implements Security” on page 10-1](#).

Table 3-1 Administration Steps for Username/Password Authentication

Step	Description
1	Set the <code>SECURITY</code> parameter in the <code>UBBCONFIG</code> file to <code>APP_PW</code> , <code>USER_AUTH</code> , <code>ACL</code> , or <code>MANDATORY_ACL</code> .
2	If you defined the <code>SECURITY</code> parameter as <code>USER_AUTH</code> , <code>ACL</code> , or <code>MANDATORY_ACL</code> , configure the authentication server (<code>AUTHSRV</code>) in the <code>UBBCONFIG</code> file.
3	Use the <code>tpusradd</code> and <code>tpgrpadd</code> commands to define lists of authorized users and groups including the <code>IIOF Listener/Handler</code> .
4	Define the <code>SEC_PRINCIPAL_NAME</code> and <code>SEC_PRINCIPAL_PASSVAR</code> for the <code>ISL</code> server process in the <code>UBBCONFIG</code> file.
5	Use the <code>tmloadcf</code> command to load the <code>UBBCONFIG</code> file. When the <code>UBBCONFIG</code> file is loaded, the system administrator is prompted for a password. The password entered at this time becomes the password for the WebLogic Enterprise application.

Table 3-2 Programming Steps for Username/Password Authentication

Step	Description
1	Write application code that uses the <code>Bootstrap</code> object to obtain a reference to the <code>SecurityCurrent</code> object.
2	Write application code that obtains the <code>PrincipalAuthenticator</code> object from the <code>SecurityCurrent</code> object.
3	Write application code that uses the <code>Tobj::PrincipalAuthenticator::logon()</code> or <code>SecurityLevel2::PrincipalAuthenticator::authenticate()</code> operation to establish a security context with the WebLogic Enterprise domain.
4	Write application code that prompts the user for the password defined when the <code>UBBCONFIG</code> file is loaded.

Certificate-based Authentication

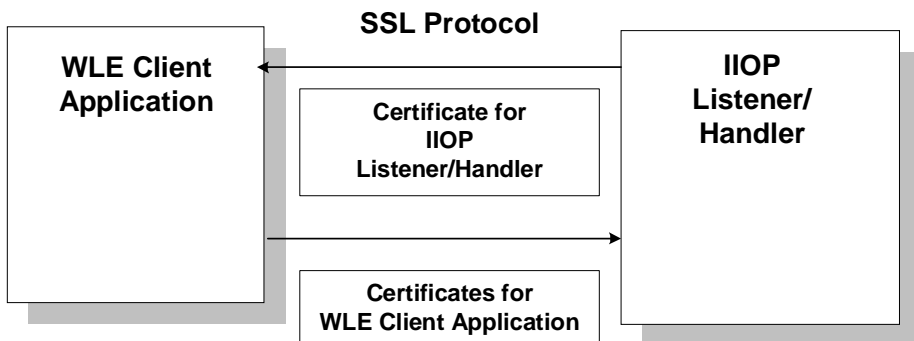
Certificate-based authentication requires that each side of an SSL connection proves its identity to the other side of the connection. In the WebLogic Enterprise product, the IIOP Listener/Handler presents its digital certificate to the principal who initiated the SSL connection. The initiator then provides a chain of digital certificates that are used by the IIOP Listener/Handler to verify the identity of the initiator.

Once a chain of digital certificates is successfully verified, the IIOP Listener/Handler retrieves the value of the distinguished name from the subject of the digital certificate. The WebLogic Enterprise product uses the e-mail address element of the subject's distinguished name as the identity of the principal. The IIOP Listener/Handler uses the identity of the principal to impersonate the principal and establish a security context between the initiating application and the WebLogic Enterprise domain.

Once the principal has been authenticated, the principal that initiated the request and the IIOP Listener/Handler agree on a cipher suite that represents the type and strength of encryption that they both support. They also agree on the encryption key and synchronize to start encrypting all subsequent messages.

Figure 3-2 provides a conceptual overview of the certificate-based authentication.

Figure 3-2 Certificate-based Authentication



How Certificate-based Authentication Works

Certificate-based authentication works in the following manner:

1. The initiating application instantiates the Bootstrap object with a URL in the form of `corbaloc://host:port` or `corbalocs://host:port` and controls the requirement for protection by setting attributes on the `SecurityLevel2::Credentials` object returned as a result of the `SecurityLevel2::PrincipalAuthenticator::authenticate` operation.

Note: You can also use the `SecurityLevel2::Current::authenticate()` method to secure the bootstrapping process and specify that certificate-based authentication is to be used.

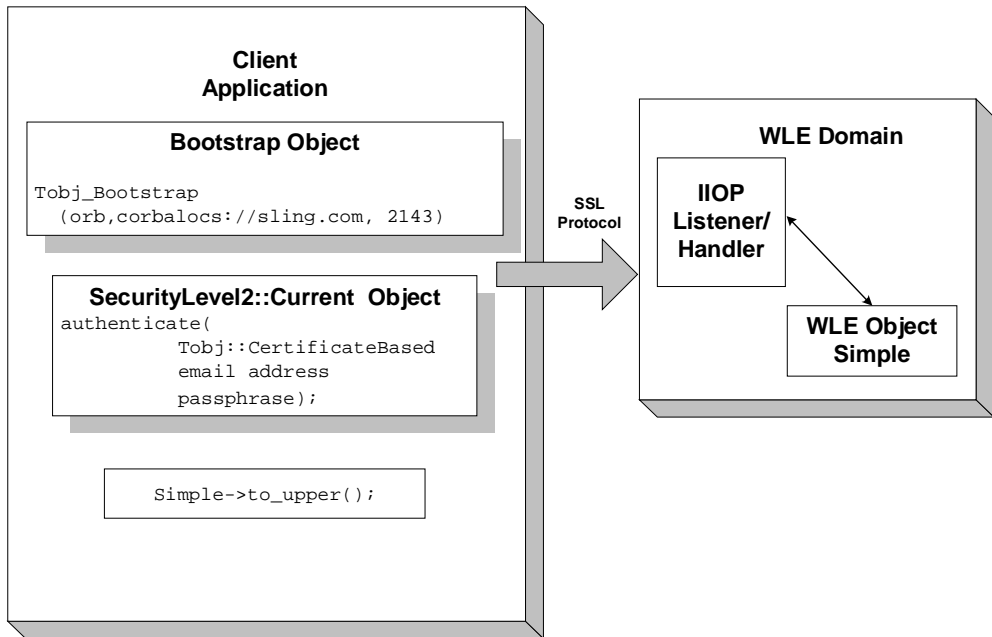
2. The initiating application obtains the digital certificates and the private key of the principal. Retrieval of this information may require proof material to be supplied to gain access to the principal's private key and certificate. The proof material typically is a pass phrase rather than a password.

The security context is established as result of a `SecurityLevel2::PrincipalAuthenticator::authenticate()` method. This step is transparent to the user of the application.

3. If the verification succeeds, the WebLogic Enterprise system constructs a `Credentials` object. The `Credentials` object for the principal represents the security context for the current thread of execution.
4. The initiating application invokes a WebLogic Enterprise object in the WebLogic Enterprise domain using an object reference.
5. The request is packaged into an IIOP request and is forwarded to the IIOP Listener/Handler that associates the request with the established security context.
6. The request is digitally signed and encrypted before it is sent to the IIOP Listener/Handler. The WebLogic Enterprise system performs the signing and sealing of requests.
7. The IIOP Listener/Handler receives the request from the initiating application. The request is decrypted.
8. The IIOP Listener/Handler retrieves the email component of the subjectDN of the principal's and uses that as the identity of the user.

9. The IIOP Listener/Handler forwards the request, along with the associated tokens of the principal, to the appropriate WebLogic Enterprise object.

Figure 3-3 How Certificate-based Authentication Works



Requirements for Using Certificate-based Authentication

Certificate-based authentication uses the SSL protocol so you need to install the WebLogic Enterprise Security Pack. Information about installing the WebLogic Enterprise Security Pack can be found in the [BEA WebLogic Enterprise Installation Guide](#). You need a LDAP server to store digital certificates. You can choose any LDAP-enabled directory service. You also need to choose the certificate authority from which to obtain certificates and private keys used in a WebLogic Enterprise application. You must have an LDAP-enabled directory service and a certificate authority in place before using certificate-based authentication in a WebLogic Enterprise application.

Development Process for Certificate-based Authentication

Using certificate-based authentication in a WebLogic Enterprise application includes administration and programming steps. Table 3-3 and Table 3-4 list the administration and programming steps for certificate-based authentication. For a detailed description of the administration steps, see [“Managing Public Key Security” on page 4-1](#) and [“Configuring the SSL Protocol” on page 6-1](#).

Table 3-3 Administration Steps for Certificate-based Authentication

Step	Description
1	Install the WebLogic Enterprise Security Pack.
2	Set up an LDAP-enabled directory service.
3	Obtain digital certificates and a private key for the IIOP Listener/Handler from a certificate authority.
4	Obtain digital certificates and a private key for each of the WebLogic Enterprise client applications from a certificate authority.
5	Store the private key files for the WebLogic Enterprise client applications and the IIOP Listener/Handler in the Home directory of the user or in <code>\$TUXDIR/udataobj/security/keys</code> .
6	Publish the certificates for the IIOP Listener/Handler, the WebLogic Enterprise application, and the certificate authority in the LDAP-enabled directory service.
7	Define the <code>SEC_PRINCIPAL_NAME</code> , <code>SEC_PRINCIPAL_LOCATION</code> , and <code>SEC_PRINCIPAL_PASSVAR</code> for the ISL server process in the <code>UBBCONFIG</code> file.
8	Use the <code>tpusradd</code> and <code>tpgrpadd</code> commands to define the authorized users and groups of your WebLogic Enterprise application.
9	Define a port for SSL communication on the IIOP Listener/Handler using the <code>-S</code> option of the ISL command.
10	Enable certificate-based authentication in the IIOP Listener/Handler using the <code>-a</code> option of the ISL command.

Table 3-3 Administration Steps for Certificate-based Authentication (Continued)

Step	Description
11	Create a Trusted Certificate Authority file (trust_ca.cer) that defines the certificate authorities trusted by the IIOP Listener/Handler.
12	Create a Trusted Certificate Authority file (trust_ca.cer) that defines the certificate authorities trusted by the WebLogic Enterprise client application.
13	Use the <code>tmloadcf</code> command to load the <code>UBBCONFIG</code> file.
14	Optionally, create a Peer Rules file (<code>peer_val.rul</code>) for both the WebLogic Enterprise client application and the IIOP Listener/Handler.
15	Optionally, modify the LDAP Search filter file to reflect the directory hierarchy in place in your enterprise.

Figure 3-4 illustrates the configuration of a WebLogic Enterprise application that uses certificate-based authentication.

Figure 3-4 Configuration for Using Certificate-based Authentication in a WebLogic Enterprise Application

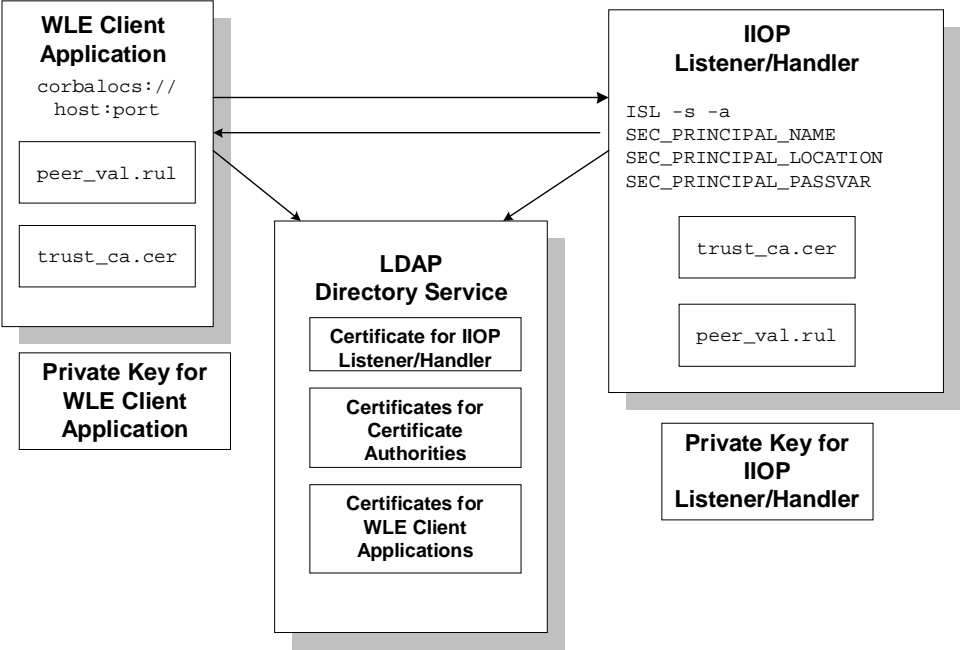


Table 3-4 lists the programming steps for using certificate-based authentication in a WebLogic Enterprise application. For more information, see [“Writing a WebLogic Enterprise CORBA Application that Implements Security” on page 10-1](#). In addition, you can use certificate-based authentication with Enterprise JavaBeans (EJBs), for more information see [“Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security” on page 12-1](#).

Table 3-4 Programming Steps for Certificate-based Authentication

Step	Description
1	Write application code that uses the <code>corbaloc</code> or <code>corbalocs</code> URL address formats of the Bootstrap object. Note that the <code>CommonName</code> in the Distinguished Name of the certificate of the IIOP Listener/Handler must match exactly the host name provided in the URL address format. For more information on the URL address formats, see “Understanding the Address Formats of the Bootstrap Object” on page 10-2.
2	Write application code that uses the <code>authenticate()</code> method of the <code>SecurityLevel2::PrincipalAuthenticator</code> interface to perform authentication. Specify <code>Tobj::CertificateBased</code> for the method argument and the pass phrase for the private key as the <code>auth_data</code> argument for <code>Security::Opaque</code> .

Using an Authentication Plug-in

The WebLogic Enterprise product allows the integration of authentication plug-ins into a WebLogic Enterprise application. The WebLogic Enterprise product can accommodate authentication plug-ins using various authentication technologies, including shared-secret password, one-time password, challenge-response, and Kerberos. The authentication interface is based on the generic security service (GSS) application programming interface (API) where applicable and assumes authentication plug-ins have been written to the GSSAPI.

If you chose to use an authentication plug-in, you must configure the authentication plug-in in the registry of the WebLogic Enterprise system. For more detail about the registry, see [“Configuring Security Plug-Ins” on page 9-1](#).

For more information about an authentication plug-ins, including installation and configuration procedures, see your BEA account executive.

Authorization

Authorization allows system administrators to control access to WebLogic Enterprise applications. Specifically, an administrator can use authorization to allow or disallow principals to use resources or services provided by a WebLogic Enterprise application.

The WebLogic Enterprise product supports the integration of authorization plug-ins. Authorization decisions are based in part on the user identity represented by an authorization token. Authorization tokens are generated during the authentication process so coordination between the authentication plug-in and the authorization plug-in is required.

The authorization plug-in is called for a given client request twice, once when the request is received and once before the reply is returned.

- The preoperation authorization check determines if a certain operation should be permitted or not. The check allows an authorization plug-in to examine the operation, the authorization token, and the data associated with the authorization token. The associated data might be modified and/or saved for the postoperation check.
- The postoperation authorization check determines if an operation should be permitted or not and is based on the analysis of the output data. The postoperation authorization check allows an authorization plug-on to examine the operation, the authorization token, and the associated data.

Postoperation checking is very useful in rule-based security models. For example, suppose a user is authorized for access to Confidential documents but performs an operation that retrieves a Top Secret document. The document's classification label is not easily determined until after it has been retrieved. In this case, the postoperation authorization check is an effective way to either deny the operation.

Multiple implementations of an authorization plug-in can be used in a WebLogic Enterprise application. Using multiple authorization plug-ins causes more than one preoperation and postoperation authorization check to be performed. A voting scheme is used to determine the result of the authorization checks.

When using multiple authorization plug-ins, all the plug-ins are placed under a single master authorization plug-in. Each subordinate authorization plug-in votes `PERMIT`, `DENY`, or `ABSTAIN` on the client request. The `ABSTAIN` return value is necessary for authorization plug-in developers to gracefully handle unexpected situations that are not understood (such as new operation names that might be added in the future).

The rules for result determination are as follows:

1. If all authorization plug-ins permit, the operation is okd.
2. If any authorization plug-in denies, the operation is denied.
3. If no authorization plug-in denies and all plug-ins do not permit:
 - If the `SECURITY` parameter in the `UBBCONFIG` file is `MANDATORY_ACL`, the operation is denied.
 - If the `SECURITY` parameter in the `UBBCONFIG` file is `ACL`, the operation is allowed.
 - If the `SECURITY` parameter in the `UBBCONFIG` file is `NONE`, `APP_PW` or `USER_AUTH`, no authorization checked is performed. This occurs because it is not possible to perform authorization checks without having an authenticated client identity.

If you chose to use an authorization plug-in, you must configure the authorization plug-in in the registry of the WebLogic Enterprise system. For more detail about the registry, see [“Configuring Security Plug-Ins” on page 9-1](#).

For more information about authorization plug-ins, including installation and configuration procedures, see your BEA account executive.

Auditing

Auditing provides a means to collect, store, and distribute information about operating requests and their outcomes. Audit-trail records may be used to determine which principals performed, or attempted to perform, actions that violated the configured security policies of a WebLogic Enterprise application. They may also be used to determine which operations were attempted, which ones failed, and which ones successfully completed.

The current implementation of the auditing feature supports the recording of logon failures, impersonation failures, and disallowed operations into the `u.log` file. In the case of disallowed operations, the value of the parameters to the operation are not provided because there is no way to know the order and data types of the parameter for an arbitrary operation. Audit entries for logon and impersonation include the identity of the principal attempting to be authenticated. For information about setting up the `u.log` file, see [Monitoring Log Files](#) in the *Administration Guide*.

You can enhance the auditing capabilities of your WebLogic Enterprise application by using an auditing plug-in. The WebLogic Enterprise system will invoke the auditing plug-in at predefined execution points, usually before an operation is attempted and then when potential security violations are detected or when operations are successfully completed. The actions taken to collect, process, protect, and distribute auditing information depend on the capabilities of the auditing plug-in. Care should be taken with the performance impact of audit information collection, especially successful operation audits, which may occur at a high rate.

Auditing decisions are based partly on user identity, which is stored in an auditing token. Because auditing tokens are generated by the authentication plug-in, providers of authentication and auditing plug-ins need to ensure that these plug-ins work together.

The purpose of an auditing request is to record an event. Each auditing plug-in returns one of two responses: `success` (the audit succeeded and the event was logged) or `failure` (the audit failed and the event was not logged the event). An auditing plug-in is called once before the operation is performed and once after the operation completes.

- The preoperation audit allows the auditing of both attempts to call an operation, and also allows storage of input data for the postoperation check.
- The postoperation audit reports the status of the completion of an operation. For failure status, the postoperation audit is called to report a potential security violation. Usually this type of report is issued when a preoperation or postoperation authorization check fails or when some other potential security attack is detected.

Multiple implementations of the auditing plug-in can be used in a WebLogic Enterprise application. Using multiple authorization plug-ins causes more than one preoperation and postoperation auditing operation to be performed.

When using multiple auditing plug-ins, all the plug-ins are placed under a single master auditing plug-in. Each subordinate authorization plug-in returns `SUCCESS` or `FAILURE`. If any plug-in fails the operation, the auditing master plug-in determines the outcome to be `FAILURE`. Other error returns are also considered `FAILURE`. Otherwise, `SUCCESS` is the outcome.

In addition, a WebLogic Enterprise system process may call an auditing plug-in when a potential security violation occurs. (Suspicion of a security violation arises when a preoperation or postoperation authorization check fails or when an attack on security is detected.) In response, the auditing plug-in performs a postoperation audit and returns whether the audit succeeded.

The auditing process is somewhat different for users of the auditing feature provided by the WebLogic Enterprise product and users of auditing plug-ins. The default auditing feature does not support preoperation audits. If the default auditing feature receives a preoperation audit request, it returns immediately and does nothing.

If you chose to use an auditing plug-in other than the default auditing plug-in, you must configure the auditing plug-in in the registry of the WebLogic Enterprise system. For more detail about the registry, see [“Configuring Security Plug-Ins” on page 9-1](#).

For more information about auditing plug-ins, including installation and configuration procedures, see your BEA account executive.

Link-Level Encryption

Link-level encryption (LLE) establishes data privacy for messages moving over the network links. The objective of LLE is to ensure confidentiality so that a network-based eavesdropper cannot learn the content of WebLogic Enterprise system messages or WebLogic Enterprise application-generated messages. It employs the symmetric key encryption technique (specifically, RC4), which uses the same key for encryption and decryption.

When LLE is being used, the WebLogic Enterprise system encrypts data before sending it over a network link and decrypts it as it comes off the link. The system repeats this encryption/decryption process at every link through which the data passes. For this reason, LLE is referred to as a point-to-point facility.

LLE can be used on the following types of network links:

- BEA Tuxedo Workstation client to Workstation Handler (WSH)
- Bridge-to-bridge (machine to machine in a WebLogic Enterprise domain)
- Administrative utility (such as `tmboot` or `tmshutdown`) to `tlisten`
- WebLogic Enterprise domain gateway to WebLogic Enterprise domain

Note: LLE cannot be used to protect connections between remote WebLogic Enterprise client applications and the IIOP Listener/Handler.

There are three levels of LLE security: 0-bit (no encryption), 56-bit (International), and 128-bit (United States and Canada). The International LLE version allows 0-bit and 56-bit encryption. The United States and Canada LLE version allows 0, 56, and 128-bit encryption.

How LLE Works

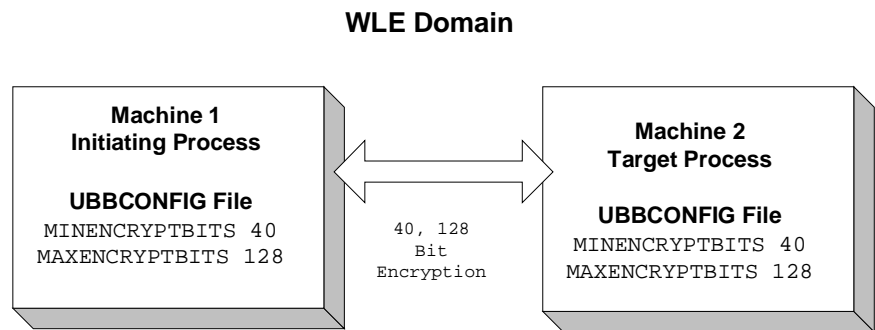
LLE works in the following way:

1. The system administrator sets parameters for any processes that want to use LLE to control the encryption strength.
 - The first configuration parameter is the minimum encryption level that a process will accept. It is expressed as a key length: 0, 56, or 128 bits.
 - The second configuration parameter is the maximum encryption level a process can support. It also is expressed as a key length: 0, 56, or 128 bits.
- For convenience, the two parameters are denoted as (*min*, *max*). For example, the values (56, 128) for a process mean that the process accepts at least 56-bit encryption but can support up to 128-bit encryption.
2. An initiator process begins the communication session.
 3. A target process receives the initial connection and starts to negotiate the encryption level to be used by the two processes to communicate.
 4. The two processes agree on the largest common key size supported by both.

5. The configured maximum key size parameter is reduced to agree with the installed software's capabilities. This step must be done at link negotiation time, because at configuration time it may not be possible to verify a particular machine's installed encryption package.
6. The processes exchange messages using the appropriate encryption level.

Figure 3-5 illustrates these steps.

Figure 3-5 How LLE Works



Encryption Key Size Negotiation

When two processes at the opposite ends of a network link need to communicate, they must first agree on the size of the key to be used for encryption. This agreement is resolved through a two-step process of negotiation.

1. Each process identifies its own *min-max* values.
2. Together, the two processes find the largest key size supported by both.

Determining min-max Values

When either of the two processes starts up, the WebLogic Enterprise system (1) checks the bit-encryption capability of the installed LLE version by checking the LLE licensing information in the `lic.txt` file and (2) checks the LLE *min-max* values for the particular link type as specified in the two configuration files. The WebLogic Enterprise system then proceeds as follows:

- If the configured *min-max* values accommodate the installed LLE version, then the local software assigns those values as the *min-max* values for the process.
- If the configured *min-max* values do not accommodate the installed LLE version, for example, if the International LLE version is installed but the configured *min-max* values are (0, 128), then the local software issues a run-time error; link-level encryption is not possible at this point.
- If there are no *min-max* values specified in the configurations for a particular link type, then the local software assigns 0 as the minimum value and assigns the highest bit-encryption rate possible for the installed LLE versions as the maximum value, that is, (0, 128) for the United States and Canada LLE version.

Finding a Common Key Size

After the *min-max* values are determined for the two processes, the negotiation of key size begins. The negotiation process need not be encrypted or hidden. Once a key size is agreed upon, it remains in effect for the lifetime of the network connection.

Table 3-5 shows which key size, if any, is agreed upon by two processes when all possible combinations of *min-max* values are negotiated. The header row holds the *min-max* values for one process; the far left column holds the *min-max* values for the other.

Table 3-5 Interprocess Negotiation Results

	(0, 0)	(0, 56)	(0, 128)	(56, 56)	(56, 128)	(128, 128)
(0, 0)	0	0	0	ERROR	ERROR	ERROR
(0, 56)	0	56	56	56	56	ERROR
(0, 128)	0	56	128	56	128	128

Table 3-5 Interprocess Negotiation Results

	(0, 0)	(0, 56)	(0, 128)	(56, 56)	(56, 128)	(128, 128)
(56, 56)	ERROR	56	56	56	56	ERROR
(56, 128)	ERROR	56	128	56	128	128
(128, 128)	ERROR	ERROR	128	ERROR	128	128

WSL/WSH Connection Timeout During Initialization

The length of time a Workstation client can take for initialization is limited. By default, this interval is 30 seconds in an application not using LLE, and 60 seconds in an application using LLE. The 60-second interval includes the time needed to negotiate an encrypted link. This time limit can be changed when LLE is configured by changing the value of the `MAXINITTIME` parameter for the Workstation Listener (WSL) server in the `UBBCONFIG` file, or the value of the `TA_MAXINITTIME` attribute in the `T_WSL` class of the `WS_MIB(5)`.

Development Process

LLE is available in the base WebLogic Enterprise product. The implementation of LLE is an administrative task. The system administrators for each WebLogic Enterprise application set *min-max* values in the `UBBCONFIG` file that control encryption strength. When the two WebLogic Enterprise applications establish communication, they negotiate what level of encryption to use to exchange messages. Once an encryption level is negotiated, it remains in effect for the lifetime of the network connection.

The SSL Protocol

The WebLogic Enterprise product provides the industry-standard SSL protocol to establish secure communications between client and server applications. When using the SSL protocol, principals use digital certificates to prove their identity to a peer.

The default behavior of the SSL protocol in the WebLogic Enterprise product is to have the IIOP Listener/Handler prove its identity to the principal who initiated the SSL connection using digital certificates. The digital certificates are verified to ensure that each of the digital certificates has not been tampered with or expired. If there is a problem with any of the digital certificates in the chain, the SSL connection is terminated. In addition, the issuer of a digital certificate is compared against a list of trusted certificate authorities to verify the digital certificate received from the IIOP Listener/Handler has been signed by a certificate authority that is trusted by the WebLogic Enterprise domain.

Figure 3-6 provides a conceptual overview of the SSL protocol.

Figure 3-6 The SSL Protocol



How the SSL Protocol Works

The SSL protocol works in the following manner:

1. The initiating application instantiates the Bootstrap object with a URL in the form of `corbaloc://host:port` or `corbalocs://host:port`.

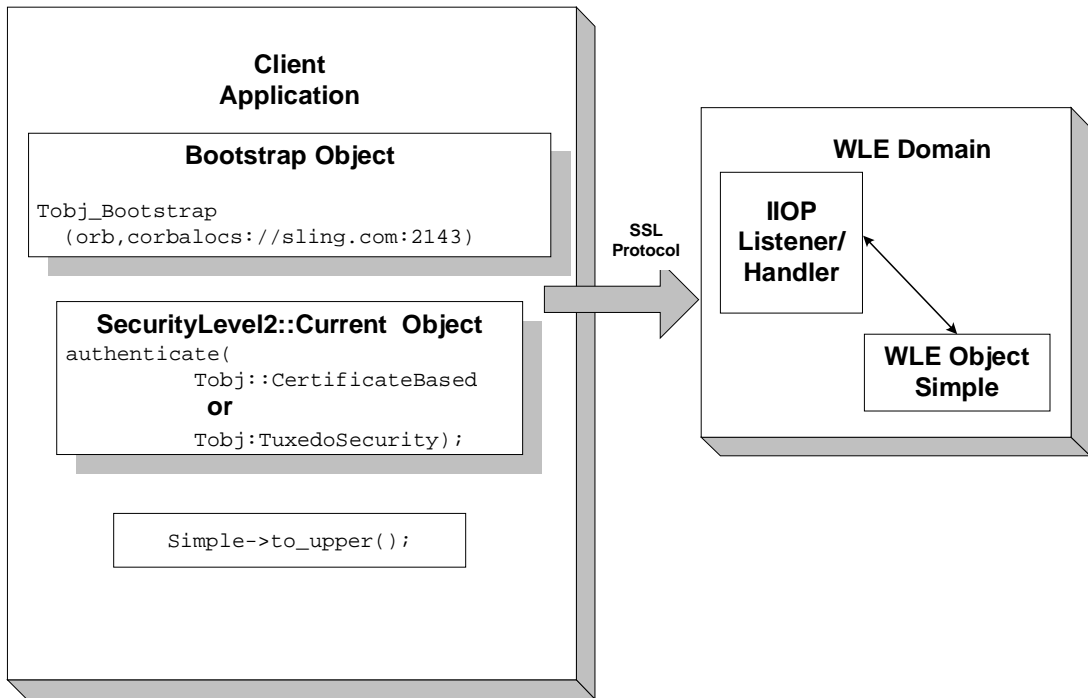
If you use the `corbaloc://host:port` URL address format, the bootstrapping process is unsecure. You can use the `authenticate()` method of the `SecurityLevel2::PrincipalAuthenticator` interface and the `invocations_options_required()` method to secure the bootstrapping process and specify that certificate-based authentication is to be used.

2. The initiating application receives the digital certificates of the principal, in this case the IIOP Listener/Handler. The security context is established as result of a `Tobj_Bootstrap::resolve_initial_references()` or a `Tobj::PrincipalAuthenticator::Logon()` method. This step is transparent to the user of the application.
3. If the verification succeeds, the WebLogic Enterprise system constructs a `Credentials` object. The `Credentials` object for the principal represents the security context for the current thread of execution.
4. The initiating application invokes a WebLogic Enterprise object in the WebLogic Enterprise domain using an object reference.
5. The request is packaged into an IIOP request and is forwarded to the IIOP Listener/Handler that associates the request with the established security context.

The request is digitally signed and/or encrypted before it is sent to the IIOP Listener/Handler. The WebLogic Enterprise system performs the signing and sealing of requests.
6. The IIOP Listener/Handler receives the request from the initiating application. The request is decrypted.
7. The IIOP Listener/Handler forwards the request to the appropriate WebLogic Enterprise object.

Figure 3-7 illustrates these steps.

Figure 3-7 How the SSL Protocol Works in a WebLogic Enterprise Application



Requirements for Using the SSL Protocol

To use the SSL protocol in a WebLogic Enterprise application, you need to install the WebLogic Enterprise Security Pack. Information about installing the WebLogic Enterprise Security Pack can be found in the [BEA WebLogic Enterprise Installation Guide](#).

The WebLogic Enterprise implementation of the SSL protocol is flexible enough to fit into most public key infrastructures. The WebLogic Enterprise product requires that digital certificates are stored in an LDAP-enabled directory. You can choose any LDAP-enabled directory service. You also need choose the certificate authority from which to obtain digital certificates and private keys used in a WebLogic Enterprise application. You must have an LDAP-enabled directory service and a certificate authority in place before using the SSL protocol in a WebLogic Enterprise application.

Development Process for the SSL Protocol

Using the SSL protocol in a WebLogic Enterprise application is primarily an administration process. Table 3-3 lists the administration steps required to set up the infrastructure required to use the SSL protocol and configure the IIOP Listener/Handler for the SSL protocol. For a detailed description of the administration steps, see [“Managing Public Key Security” on page 4-1](#) and [“Configuring the SSL Protocol” on page 6-1](#).

Once the administration steps are complete, you can use either Username/Password authentication or certificate-based authentication in your WebLogic Enterprise application. For more information, see [“Writing a WebLogic Enterprise CORBA Application that Implements Security” on page 10-1](#). In addition, you can use the SSL protocol with Enterprise JavaBeans, for more information, see [“Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security” on page 12-1](#).

Note: If you are using the BEA CORBA C++ or CORBA Java ORB as a server application, the ORB can also be configured to use the SSL protocol. For more information, see [“Configuring the SSL Protocol” on page 6-1](#).

Table 3-6 Administration Steps for the SSL Protocol

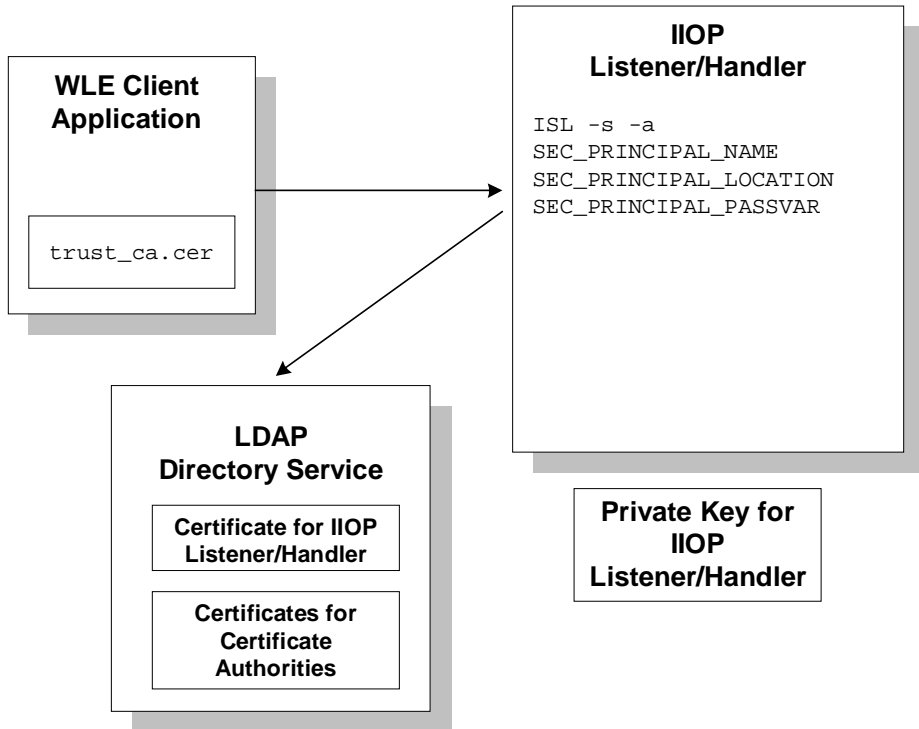
Step	Description
1	Install the WebLogic Enterprise Security Pack.
2	Set up an LDAP-enabled directory service.
3	Obtain a digital certificate and private key for the IIOP Listener/Handler from a certificate authority.
4	Publish the digital certificates for the IIOP Listener/Handler and the certificate authority in the LDAP-enabled directory service.
5	Define the SEC_PRINCIPAL_NAME, SEC_PRINCIPAL_LOCATION, and SEC_PRINCIPAL_PASSVAR parameters for the ISL server process in the UBBCONFIG file.
6	Define a port for secure communication on the IIOP Listener/Handler using the -S option of the ISL command.

Table 3-6 Administration Steps for the SSL Protocol (Continued)

Step	Description
7	Create a Trusted Certificate Authority file (<code>trust_ca.cer</code>) that defines the certificate authorities trusted by the IIOP Listener/Handler.
8	Use the <code>tmloadcf</code> command to load the <code>UBBCONFIG</code> file.
9	Optionally, create a Peer Rules file (<code>peer_val.rul</code>) for the IIOP Listener/Handler.
10	Optionally, modify the LDAP Search filter file to reflect the directory hierarchy in place in your enterprise.

Figure 3-8 illustrates the configuration of a WebLogic Enterprise application that uses the SSL protocol.

Figure 3-8 Configuration for Using the SSL Protocol in a WebLogic Enterprise Application



Single Sign-On

Single sign-on allows authenticated principals in a WebLogic Server realm to make secure requests on CORBA objects or EJBs in the WebLogic Enterprise domain. Single sign-on is only supported over the connection pool provided by WebLogic Enterprise Connectivity and only if the connection pool has established a trust relationship with the WebLogic Enterprise environment. The trust relationship of the pool can be established in one of the following ways:

- With username/password authentication. In this scenario, the WebLogic Server principal is authenticated but the request between the WebLogic Server realm and the WebLogic Enterprise domain is unprotected.
- With username/password authentication and the SSL protocol. In this scenario, the SSL protocol is used to protect the integrity and confidentiality of the request.
- With the SSL protocol and certificate-based authentication. This is the most secure scenario, however, it requires that both the WebLogic Server application and the WebLogic Enterprise application implement public key security.

“Configuring Single Sign-on” on page 8-1 describes how to implement each of the Single sign-on options.

PKI Plug-Ins

The WebLogic Enterprise product provides a PKI environment which includes the SSL protocol and the infrastructure needed to use digital certificates. However, you can use the PKI interfaces to integrate a PKI plug-in that supplies custom message-based digital signature and message-based encryption to your WebLogic Enterprise applications. Table 3-7 describes the PKI interfaces.

Table 3-7 PKI Interfaces

PKI Interface	Description
Public key initialization	Allows public key software to open public and private keys. For example, gateway processes may need to have access to a specific private key in order to decrypt messages before routing them.
Key management	Allows public key software to manage and use public and private keys. Note that message digests and session keys are encrypted and decrypted using this interface, but no bulk data encryption is performed using public key cryptography. Bulk data encryption is performed using symmetric key cryptography.

Table 3-7 PKI Interfaces

PKI Interface	Description
Certificate lookup	Allows public key software to retrieve X.509v3 digital certificates for a given principal. Digital certificates may be stored using any appropriate certificate repository, such as Lightweight Directory Access Protocol (LDAP).
Certificate parsing	Allows public key software to associate a simple principal name with an X.509v3 digital certificate. The parser analyzes a digital certificate to generate a principal name to be associated with the digital certificate.
Certificate validation	Allows public key software to validate an X.509v3 digital certificate in accordance with specific business logic.
Proof material mapping	Allows public key software to access the proof materials needed to open keys, provide authorization tokens, and provide auditing tokens.

The PKI interfaces support the following algorithms:

- Public key algorithms: Rivest, Shamir, and Adelman (RSA) and Digital Signature Algorithm (DSA)
- Symmetric key algorithms:
 - Data Encryption Standard for Cipher Block Chaining (DES-CBC)
 - Two-key triple-DES
 - Rivest's Cipher 4 (RC4)
- Message digest algorithms:
 - Message Digest 5 (MD5)
 - Secure Hash Algorithm 1 (SHA-1)

If you chose to use a PKI plug-in, you must configure the PKI plug-in in the registry of the WebLogic Enterprise system. For more detail about the registry, see [“Configuring Security Plug-Ins” on page 9-1](#).

For more information about PKI plug-ins, including installation and configuration procedures, see your BEA account executive.

Commonly Asked Questions About the WebLogic Enterprise Security Features

The following sections answer some of the commonly asked questions about the WebLogic Enterprise Security features.

Do I Have to Change the Security in an Existing WebLogic Enterprise Application?

The answer is no. If you are using security interfaces from previous versions of the WebLogic Enterprise product in your WebLogic Enterprise application there is no requirement for you to change your WebLogic Enterprise application. You can leave your current security scheme in place and your existing WebLogic Enterprise application will work with WebLogic Enterprise applications built with the WebLogic Enterprise 5.0 product.

For example, if your WebLogic Enterprise application consists of a set of server applications which provide general information to all client applications which connect to them, there is really no need to implement a stronger security scheme. If your WebLogic Enterprise application has a set of server applications which provide information to client applications on an internal network which provides enough security to detect sniffers, you do not need to implement the features in the WebLogic Enterprise Security Pack.

Can I Use the SSL Protocol in an Existing WebLogic Enterprise Application?

The answer is yes. You may want to take advantage of the extra security protection provided by the SSL protocol in your existing WebLogic Enterprise application. For example, if you have a WebLogic Enterprise server application which provides stock prices to a specific set of client applications, you can use the SSL protocol to make sure the client applications are connected to the correct WebLogic Enterprise server application and that they are not being routed to a fake WebLogic Enterprise server application with incorrect data. A username and password is sufficient proof material to authenticate the client application. However, by using the SSL protocol, the message request/reply information can be protected as an additional level of security.

The SSL protocol offers WebLogic Enterprise applications the following benefits:

- Protection of the entire conversation including the initial bootstrapping process. The SSL protocol protects against man-in-the-middle attacks, replay attacks, tampering, and sniffing.
- Even if you only use the default settings, the SSL protocol provides signed and sealed protection since the default encryption settings are a minimum of 56 bit by default.
- Client verification of the connected IIOP Listener/Handler using the digital certificate of the IIOP Listener/Handler. The client application can then apply additional security rules to restrict access to the client application by the IIOP Listener/Handler. This protection also applies to IIOP Listener/Handlers connecting to remote server applications when using callback objects.

To use the SSL protocol in a WebLogic Enterprise application, set up the infrastructure to use digital certificates, change the command-line options on the ISL server process to use the SSL protocol, and configure a port for secure communications on the IIOP Listener/Handler. If your existing WebLogic Enterprise application uses Username/Password authentication, you can use that code with the SSL protocol. If your WebLogic Enterprise C++ CORBA client application does not already catch the `InvalidDomain` exception when resolving initial references to the Bootstrap object and performing authentication, write code to handle this exception. For more information, see “The SSL Protocol” on page 3-21.

Note: The Java implementation of the `Tobj_Bootstrap::resolve_initial_references()` method does not throw an `InvalidDomain` exception. When the `corbaloc` or `corbalocs` URL address formats are used, the `Tobj_Bootstrap::resolve_initial_references()` method internally catches the `InvalidDomain` exception and throws the exception as a `COMM_FAILURE`. The method functions this way in order to provide backward compatibility.

When Should I Use Certificate-based Authentication?

You might be ready to migrate your existing WebLogic Enterprise application to use Internet connections between the WebLogic Enterprise application and web browsers and commercial web servers. For example, users of your WebLogic Enterprise application might be shopping over the Internet. The users must be confident that:

- They are in fact communicating with the server at the online store and not an impostor that mimics the store's server to get credit card information.
- The data exchanged between the user of the WebLogic Enterprise application and the online store will be unintelligible to network eavesdroppers.
- The data exchanged with the online store will arrive unaltered. An instruction to order \$500 worth of merchandise must not accidentally or maliciously become a \$5000 order.

In these situations, the SSL protocol and certificate-based authentication offer WebLogic Enterprise applications the maximum level of protection. In addition to the benefits achieved through the use of the SSL protocol, certificate-based authentication offers WebLogic Enterprise applications:

- IIOP Listener/Handler verification of the client application that initiates a request using the digital certificate of the client application. In addition, the IIOP Listener/Handler can apply additional rules which restrict access to the client application based on the identity established by the digital certificate. A remote ORB acting as a server application can also be configured to allow mutual authentication and verify the identity of a client application based on a digital certificate.
- Inside the WebLogic Enterprise domain, the client application can still have a BEA Tuxedo username and password. The IIOP Listener/Handler maps the

identity defined in a digital certificate to a BEA Tuxedo username and password thus allowing existing WebLogic Enterprise applications to have an identity in native WebLogic Enterprise server applications.

For more information, see “The SSL Protocol” on page 3-21.

Part II Security Administration

- Chapter 4. Managing Public Key Security
- Chapter 5. Configuring Link-Level Encryption
- Chapter 6. Configuring the SSL Protocol
- Chapter 7. Configuring Authentication
- Chapter 8. Configuring Single Sign-on
- Chapter 9. Configuring Security Plug-Ins

4 Managing Public Key Security

This topic includes the following sections:

- Requirements for Using Public Key Security
- Who Needs Digital Certificates and Private/Private Key Pairs
- Requesting a Digital Certificate
- Publishing Certificates in the LDAP Directory Service
- Editing the LDAP Search Filter File
- Storing the Private Keys in a Common Location
- Defining the Trusted Certificate Authorities
- Creating a Peer Rules File

Perform the tasks in this topic only if you are using the SSL protocol, or certificate-based authentication in your WebLogic Enterprise application.

Requirements for Using Public Key Security

To use the SSL protocol and public key security to protect communication between principals and the WebLogic Enterprise domain, you need to install the WebLogic Enterprise Security Pack. The WebLogic Enterprise Security Pack contains the files

necessary to enable the use of the SSL protocol and the infrastructure for public key security. For complete information about installing the WebLogic Enterprise Security Pack, see the [BEA WebLogic Enterprise Installation Guide](#).

You also need to choose a Lightweight Directory Access Protocol server and a certificate authority (either commercial or private) set up for your organization before implementing Public Key Security.

Who Needs Digital Certificates and Private/Private Key Pairs

To use the SSL protocol in the WebLogic Enterprise Security environment, you need a private key and a digitally-signed certificate containing the matching public key. How many digital certificates and private keys you need depends on how you plan to use the SSL protocol.

- If the SSL protocol is being used for protection of a network connection between a remote client and the IIOP Listener/Handler, you only need to obtain a digital certificate and private key for the IIOP Listener/Handler.

In this case, the remote client must only be configured with a list of trusted certificate authorities.

- If the SSL protocol is being used with certificate-based authentication, you need to obtain a digital certificate and private key for the IIOP Listener/Handler and each principal that will access the WebLogic Enterprise application.

Requesting a Digital Certificate

To acquire a digital certificate, you need to submit your request for a digital certificate in a particular format called a certificate signature request (CSR). How you create a CSR depends on the certificate authority you use. Certificate authorities typically

provide a means to generate a public key, private key, and a CSR which contains your public key. To create a CSR follow the steps outlined by your chosen certificate authority.

When you complete the steps to create a CSR, you receive the following files from the certificate authority:

File	Description
<i>key.der</i>	The private key file.
<i>request.pem</i>	The CSR file which you submit to the certificate authority. It contains the same data as the <i>.der</i> file but the file is encoded in ASCII so that you can copy it into e-mail or paste it into a web form.

To purchase a digital certificate from a certificate authority, you submit the CSR to the certificate authority according to the enrollment procedure of the certificate authority. Some commercial certificate authorities allow you to purchase digital certificates over through the Web.

Publishing Certificates in the LDAP Directory Service

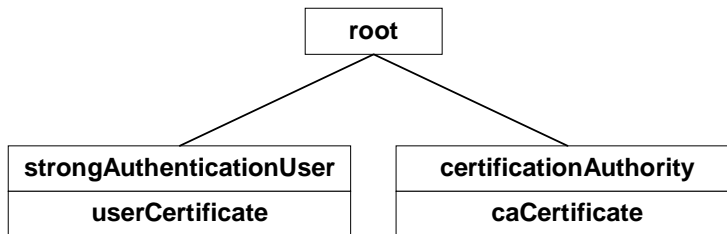
The use of a global directory service is the most popular way to store digital certificates. A directory service simplifies the management of information that needs to be globally available to an ever-growing number of users. An LDAP server provides access to a variety of directory services.

The WebLogic Enterprise product when configured to use the SSL protocol can retrieve digital certificates for principals and certificate authorities from an LDAP directory service, such as Netscape Directory Service or Microsoft Active Directory. Before you can use the SSL protocol or certificate-based authentication, you need to install an LDAP directory service and configure it for your organization. BEA Systems

does not provide nor recommend any specific LDAP directory service. However, the LDAP directory service you choose should support the X.500 scheme definition and the LDAP version 2 or 3 protocol.

LDAP directory services define a hierarchy of object classes. While there are a number of different object classes, there is a small set associated with digital certificates. Figure 4-1 illustrates the object classes typically associated with digital certificates.

Figure 4-1 LDAP Directory Structure for Digital Certificates



Once you receive your digital certificates from the certificate authority, store them in the LDAP directory service as follows:

- Digital certificates for the IIOP Listener/Handler and any principals are stored in the LDAP directory service with an attribute of `userCertificate` on an object class with that attribute defined. Typically, these digital certificates are stored as an instance of the `strongAuthenticationUser` object class as defined by X.500.
- Digital certificates for certificate authorities are stored in LDAP directory service with an attribute of `caCertificate` on an object class with that attribute defined. Typically, these digital certificates are stored as an instance of the `certificationAuthority` class as defined by X.500.

If your LDAP scheme requires the use of different classes, you will need to modify the LDAP search file as described in “Editing the LDAP Search Filter File” on page 4-5.

The WebLogic Enterprise product requires that the digital certificates be stored in the directory service in Privacy Enhanced Mail (PEM) format.

Refer to the [BEA WebLogic Enterprise Installation Guide](#) for information about integrating an LDAP directory service into the WebLogic Enterprise environment.

Editing the LDAP Search Filter File

When configuring a WebLogic Enterprise application to use the SSL protocol or certificate-based authentication, you may need to customize the LDAP search filter file to limit the scope of the search of the directory service or specify the object classes that will be used to hold the digital certificates. Customizing the LDAP search filter file can result in significant performance gains. The WebLogic Enterprise Security Pack ships with the following LDAP search filters:

- A filter stanza that searches the directory service for digital certificates assigned to certificate authorities. The filter limits its search to instances of the `certificationAuthority` object class.
- A filter stanza that searches the directory service for digital certificates assigned to principals. The filter limits its search to instances of the `strongAuthenticationUser` object class.

If the directory service scheme for your organization is defined to store digital certificates in object classes other than `certificationAuthority` and `strongAuthenticationUser`, the LDAP search filter file must be modified to specify those object classes.

You can specify a location of the LDAP search filter file during the installation of the WebLogic Enterprise Security pack. For more information, see the [BEA WebLogic Enterprise Installation Guide](#).

The LDAP search filter file should be owned by the administrator account. BEA recommends that the file be protected so that only the owner has read and write privileges for the file and all other users have only read privileges for the file.

To limit the search of the directory service for digital certificates for principals and certificate authorities, you need to modify the filter stanzas identified by the following tags in the LDAP search filter file:

- `BEA_person_lookup`
- `BEA_issuer_lookup`

These tags identify the stanzas in the LDAP search filter file that contains the filter expression that will be used when looking up information in the directory service. These BEA-specific tags allow the stanzas of an LDAP search filter file to be stored in a common LDAP search filter file with stanzas used by other LDAP-enabled applications that might be found in your organization.

The following is an example of the stanzas of an LDAP search filter file used by the WebLogic Enterprise product for the SSL protocol and certificate-based authentication:

```
"BEA_person_lookup"
".*" " "(|(objectClass=strongAuthenticationUser) (mail=%v))"
                                     "e-mail address"
      "(|(objectClass=strongAuthenticationUser) (mail=%v))"
                                     "start of e-mail address"

"BEA_issuer_lookup"
".*" " " (&(objectClass=certificationAuthority)
        (cn=%v)" "exact match cn"
        (sn=%v))" "exact match sn"
```

- `BEA_person_lookup` specifies to search the LDAP directory service for principals by their e-mail addresses.
- `BEA_issuer_lookup` specifies to search the LDAP directory service for principals by their common names (cn).

See the documentation for your LDAP-enabled directory service for additional information about LDAP search file filters.

Storing the Private Keys in a Common Location

When a principal generates a CSR, they typically get a file with a private key. Principals need this private key file to verify their identity in the authentication process. Assign the private key file protections so that only the owner of the private key file has read privileges and all other users have no privileges to access the file. Private key files must be stored as PEM-encoded PKCS #8 protected format.

The WebLogic Enterprise system uses the e-mail address of the principal to construct a name for the private key file as follows:

1. The @ character in the name is replaced by an underscore (_) character.
2. All characters after the dot (.) character are deleted.
3. A .PEM file extension is appended to the file.

For example, if the name of the principal is `milozzi@bigcompany.com` the resulting private key file is `milozzi_bigcompany.pem`. This naming convention allows an enterprise to have multiple principals that share a common username but are in different e-mail domains.

The WebLogic Enterprise software looks in the following directories for private key files:

Window NT

`%HOMEDRIVE%\%HOMEPATH%`

UNIX

`$HOME`

The WebLogic Enterprise software also looks in the following directory for private key files:

`$TUXDIR/udataobj/security/keys`

The `/keys` directory should be protected so that only the owner has read privileges for the directory and all other users do not have privileges to access the directory.

Listing 4-1 provides an example of a private key file.

Listing 4-1 Example of Private Key File

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIICoDAaBgkqhkiG9w0BBQMwDQItSFrtYcfKygCAQUEggKAEGrMxo8gYB/MOSXG
...
-----END ENCRYPTED PRIVATE KEY-----
```



```
...
-----END CERTIFICATE-----
```

Creating a Peer Rules File

When communicating across network links, it is important to validate the peer to which you are connected is the intended or authorized peer. Without this check, it is possible to make a secure connection, exchange secure messages, and receive a valid chain of digital certificates but still be vulnerable to a Man-in-the-Middle attack. You perform peer validation by verifying a set of specified information contained in the peer digital certificate against a list of information that specifies the rules for validating peer trust. The system administrator maintains the Peer Rules file.

The Peer Rules are maintained in an ASCII file named `peer_val.rul`. Store the `peer_val.rul` file in the following location in the WebLogic Enterprise directory structure:

```
$TUXDIR/udataobj/security/certs
```

Listing 4-3 provides an example of a Peer Rules file.

Listing 4-3 Example of Peer Rules File

```
#
# This file contains the list of rules for validating if
# a peer is authorized as the target of a secure connection
#
O=Ace Industry
O="Acme Systems, Inc."; OU=Central Engineering;L=Herkimer;S=NY
O="Ball, Corp.", C=US
o=Ace Industry, ou=QA, cn=www.ace.com
```

Each rule in the Peer Rules file is comprised of a set of elements that are identified by a key. The WebLogic Enterprise product recognizes the key names listed in Table 4-1.

Table 4-1 Supported Keys for Peer Rules File

Key	Attribute
CN	CommonName
SN	SurName
L	LocalityName
S	StateOrProvinceName
O	OrganizationName
OU	OrganizationalUnitName
C	CountryName
E	EmailAddress

Each key is followed by an optional white space, the character =, an optional white space, and finally the value to be compared. The key is not case sensitive. A rule is not a match unless the subject's distinguished name contains each of the specified elements in the rule and the values of those elements match the values specified in the rule, including case and punctuation.

Each line in the Peer Rules file contains a single rule that is used to determine if a secure connection is to be established. Rules cannot span lines; the entire rule must appear on a single line. Each element in the rule can be separated by either a comma (,) or semi-colon (;) character.

Lines beginning with the pound character (#) are comments. Comments cannot appear on the same line as the name of an organization.

A value must be enclosed in single quotation marks if one of the following cases is true:

- Strings contain any of the following characters:

, + = " " <CR> < > # ;

- Strings have leading or trailing spaces
- Strings contain consecutive spaces

By default, the WebLogic Enterprise product verifies peer information against the Peer Rules file. If you do not want to perform this check, create an empty Peer Rules file.

5 Configuring Link-Level Encryption

This topic includes the following sections

- Understanding min and max Values
- Verifying the Installed Version of LLE
- Configuring LLE on WebLogic Enterprise Application Links
- Configuring LLE on BEA Tuxedo Workstation Client Links
- Configuring LLE on Bridge Links
- Configuring LLE on tlisten Links
- Configuring LLE on Domain Gateway Links

Understanding min and max Values

Before you can configure LLE for your WebLogic Enterprise application, you need to be familiar with the LLE notation: (*min*, *max*). The defaults for these parameters are:

- For *min*: 0
- For *max*: Number of bits that indicates the highest level of encryption possible for the installed LLE version

For example, the default *min* and *max* values for the United States and Canada LLE version are (0, 128). If you want to change the defaults, you can do so by assigning new values to *min* and *max* in the `UBBCONFIG` file for your application.

Verifying the Installed Version of LLE

Before setting the *min* and *max* values for your WebLogic Enterprise application, you need to verify what version of LLE is installed on your machine. You can verify the LLE version installed on a machine by running the `tmadmin` command in verbose mode as follows:

```
tmadmin -v
```

Key lines from the WebLogic Enterprise license file (`lic.txt`) appear on your computer screen, similar to information in Listing 5-1. The entry `128-bit Encryption Package` indicates that the United States and Canada version of LLE is installed.

Listing 5-1 LLE Licence Information

```
INFO: BEA Engine, Version 2.4
INFO: Serial: 212889588, Expiration 2000-3-15, Maxusers 10000
INFO: Licensed to: ACME CORPORATION
INFO: 128-bit Encryption Package
```

WebLogic Enterprise license files are located in the following directories:

Windows NT

```
%TUXDIR%\udataobj\lic.txt
```

UNIX

```
$TUXDIR/udataobj/lic.txt
```

Configuring LLE on WebLogic Enterprise Application Links

To configure LLE in WebLogic Enterprise applications, you need to set the `MINENCRYPTBITS` and `MAXENCRYPTBITS` parameters in the `UBBCONFIG` file for each WebLogic Enterprise application participating in the network connection, as follows:

- The `MINENCRYPTBITS` parameter specifies that at least the defined number of bits are meaningful.
- The `MAXENCRYPTBITS` parameter specifies that encryption should be negotiated up to the defined level.

The possible values for the `MINENCRYPTBITS` and `MAXENCRYPTBITS` parameters are 0, 40, and 128. A value of zero means no encryption is used, while 40 and 128 specify the number of significant bits in the encryption key.

Load the configuration file by running `tmloadcf`. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.

Configuring LLE on BEA Tuxedo Workstation Client Links

If Workstation clients are included in an application, the system administrator must configure one or more Workstation Listeners (WSLs) to listen for connection requests from Workstation clients. Each WSL uses one or more associated Workstation Handlers (WSHs) to handle the Workstation client workload. Each WSH can manage multiple Workstation clients by multiplexing all requests and replies with a particular Workstation client over a single connection.

As the system administrator, you enable Workstation client access to the application by specifying a WSL server in the `SERVERS` section of the application's `UBBCONFIG` file. You need to specify the `-z` and `-Z` command-line options for the WSL server if you want to override the defaults for the LLE *min* and *max* parameters. You can only use link-level encryption if LLE is installed on both the local machine and the Workstation client.

Note: At the Workstation client end of a network connection, you use environment variables `TMINENCRYPTBITS` and `TMAXENCRYPTBITS` to override the defaults for the LLE *min* and *max* parameters.

To configure LLE on Workstation client links, complete the following steps:

1. Ensure that you are working on the application `MASTER` machine and that the application is inactive.
2. Open `UBBCONFIG` with a text editor and add the following lines to the `SERVERS` section.

```
*SERVERS
WSL      SRVGRP="group_name" SRVID=server_number ...
        CLOPT="-A -- -z min -Z max ..."
```

3. Load the configuration by running the `tmloadcf` command. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.

When the `tmboot` command starts the application, it passes the `"-A -- -z min -Z max"` command-line options to the WSL server. When establishing a network link between a Workstation client and the WSH, the Workstation client and WSL negotiate the key size until they agree on the largest key size supported by both.

Configuring LLE on Bridge Links

The WebLogic Enterprise system architecture optimizes network communications by establishing a multiplexed *channel* among the machines in a multiple-machine application. WebLogic Enterprise application messages flow in both directions over this channel, and the message traffic is managed by a specialized WebLogic Enterprise server known as a Bridge server.

As the system administrator, you place an entry in the `NETWORK` section of the `UBBCONFIG` file for each machine in a WebLogic Enterprise application on which a Bridge server resides. You need to specify the `MINENCRYPTBITS` and `MAXENCRYPTBITS` optional run-time parameters for the Bridge server if you want to override the defaults for the LLE *min* and *max* parameters. Using LLE on Bridge-to-Bridge links is possible only if LLE is installed on the machines where the Bridge servers reside.

To configure LLE on Bridge links, complete the following steps:

1. Ensure that you are working on the application `MASTER` machine and that the application is inactive.
2. Open `UBBCONFIG` with a text editor and add the following lines to the `NETWORK` section.

```
*NETWORK
LMID    NADDR="bridge_network_address" BRIDGE="bridge_device"
        NLSADDR="listen_network_address"
        MINENCRYPTBITS=min
        MAXENCRYPTBITS=max
```

LMID is the logical machine where the Bridge server resides; it has direct access to the network device specified in the `BRIDGE` parameter.

3. Load the configuration by running the `tmloadcf` command. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.

When the `tmboot` command starts the WebLogic Enterprise application, the Bridge server reads the `TUXCONFIG` file to access various parameters, including `MINENCRYPTBITS` and `MAXENCRYPTBITS`. When establishing a network link with a remote Bridge server, the local and remote Bridge servers negotiate the key size until they agree on the largest key size supported by both.

Configuring LLE on tlisten Links

The `tlisten` process is a network-independent listener process that provides connections between nodes of a multiple-machine application, on which administrative utilities such as the `tmboot` command can run. The system administrator installs `tlisten` on all machines defined in the `NETWORK` section of the `UBBCONFIG` file.

To configure LLE on `tlisten` links, follow the steps given in the previous topic, “Configuring LLE on Bridge Links” on page 5-4. If you so desire, you can start a separate instance of `tlisten` on the local machine by entering the command:

```
tlisten -l nlsaddr [-z min -Z max]
```

The `nlsaddr` value must be the same as that specified for the `NLSADDR` parameter for this machine in the `NETWORK` section of the `UBBCONFIG` file.

Configuring LLE on Domain Gateway Links

A domain gateway is a `GWTDOMAIN` process that relays service requests and service replies between two or more WebLogic Enterprise applications. It provides interoperability through a specially designed transaction processing (TP) protocol that flows over network transport protocols such as TCP/IP.

A domain gateway belongs to a *domain gateway group*, for which a separate Domains configuration file is required. A domain gateway group consists of a local domain access point (LDM) and the remote domain access points (RDMs) with which it communicates. Like the application configuration files, `UBBCONFIG` and `TUXCONFIG`, a Domains configuration file is created in text format and then converted to binary format. The text and binary files are referred to as `DMCONFIG` and `BDMCONFIG`, respectively. The `DMCONFIG` and `BDMCONFIG` files, and the environment variables associated with them, are described on the `DMCONFIG(5)` reference page in the *BEA Tuxedo File Formats and Data Descriptions Reference*.

As the system administrator, you must place an entry in the `DM_TDOMAIN` section of the `DMCONFIG` file for each local domain access point that will accept requests for local services from remote domain access points. You must also create an entry for each remote domain access point accessible by a defined local domain access point. You need to specify the `MINENCRYPTBITS` and `MAXENCRYPTBITS` optional run-time parameters for each domain access point for which you want to override the defaults for the LLE *min* and *max* parameters. Domain-to-domain link-level encryption is possible only if LLE is installed on the machines where the domains reside.

To configure LLE on domain gateway links, complete the following steps:

1. Ensure that you are working on the application `MASTER` machine and that the application is inactive.
2. Open `DMCONFIG` with a text editor and add the following lines to the `DM_TDOMAIN` section.

```
*DM_TDOMAIN
# Local network addresses
LDOM  NWADDR="local_domain_network_address"
      NWDEVICE="local_domain_device"
      MINENCRYPTBITS=min
      MAXENCRYPTBITS=max
      .
      .
      .

# Remote network addresses
RDOM  NWADDR="remote_domain_network_address"
      NWDEVICE="remote_domain_device"
      MINENCRYPTBITS=min
      MAXENCRYPTBITS=max
      .
      .
      .
```

LDOM is a local domain access point identifier, and *RDOM* is a remote domain access point identifier.

3. Load the configuration by running the `dmloadcf` command. The `dmloadcf` command parses `DMCONFIG` and loads the binary `BDMCONFIG` file to the location referenced by the `BDMCONFIG` variable.

In the preceding example, when the `tmboot` command starts the application, each domain gateway reads the `BDMCONFIG` file to access various parameters, including `MINENCRYPTBITS` and `MAXENCRYPTBITS`, and propagates those parameters to its local

5 *Configuring Link-Level Encryption*

and remote domains. When the local domain is establishing a network link with a remote domain, the two domains negotiate the key size until they agree on the largest key size supported by both.

6 Configuring the SSL Protocol

This topic includes the following sections:

- Setting Parameters for the SSL Protocol
- Defining a Port for SSL Network Connections
- Enabling Host Matching
- Setting the Encryption Strength
- Setting the Interval for Session Renegotiation
- Defining Security Parameters for the IIOP Listener/Handler
- Example of Setting Parameters on the ISL System Process
- Example of Setting Command-Line Options on the CORBA C++ ORB

Setting Parameters for the SSL Protocol

To use the SSL protocol or certificate-based authentication with the IIOP Listener/Handler, the CORBA C++ object request broker (ORB), or the CORBA Java ORB, you need to:

- Specify the secure port on which SSL network connections will be accepted.
- Specify the strength that will be used when encrypting data.
- Optionally, set the interval for session renegotiation (IIOP Listener/Handler only).

The following sections detail how to use the options of the ISL command, the command-line options of the CORBA C++ ORB, or the system properties of the CORBA Java ORB to set these SSL parameters.

Defining a Port for SSL Network Connections

To define a port for SSL network connections:

- Use the `-s` option of the ISL command to specify which port of the IIOP Listener/Handler will listen for secure connections using the SSL protocol. You can configure the IIOP Listener/Handler to allow only SSL connections by setting the `-s` option and `-n` option of the ISL command to the same value.
- If you are using a remote CORBA C++ or CORBA Java ORB, use the `-ORBsecurePort` command-line option or system property on the ORB to specify which port of the ORB will listen for secure connections using the SSL protocol. You should set this command-line option or system property when using callback objects or the WebLogic Enterprise Notification Service.

Note: If you are using the SSL protocol with a joint client/server application, you must specify a port number for SSL network connections. You cannot use the default.

Defining a secure port for SSL network connections requires the WebLogic Enterprise Security Pack to be installed. If the `-s` option or the `-ORBsecurePort` command-line option or system property is executed and a license to enable the use of the SSL protocol does not exist, the IIOP Listener/Handler, CORBA C++ ORB, or CORBA Java ORB will not start.

Enabling Host Matching

The SSL protocol is capable of encrypting messages for confidentiality; however, the use of encryption does nothing to prevent a man-in-the-middle attack. During a man-in-the-middle attack, a principal masquerades as the location from which an initiating application retrieves the initial object references used in the bootstrapping process.

To prevent man-in-the-middle attacks, it is necessary to perform a check to ensure that the digital certificate received during an SSL connection is for the principal for which the connection was intended. Host Matching is a check that the host specified in the object reference used to make the SSL connection matches the common name in the subject in the distinguished name specified in the target's digital certificate. Host Matching is performed only by the initiator of an SSL connection, and confirms that the target of a request is actually located at the same network address specified by the domain name in the target's digital certificate. If this comparison fails, the initiator of the SSL connection refuses to authenticate the target and drops the SSL connection. Host Matching is not technically part of the SSL protocol and is similar to the same check done in Web browsers.

The domain name contained in the digital certificate must match exactly the host information contained in the object reference. Therefore, the use of DNS host names instead of IP addresses is strongly encouraged.

By default, Host Matching is enabled in the IIOP Listener/Handler and in the CORBA C++ and CORBA Java ORBs. If you need to enable Host Matching, do one of the following:

- In the IIOP Listener/Handler, specify the `-v` option of the ISL command.
- In the CORBA C++ or CORBA Java ORBs, specify the `-ORBpeerValidate` command-line option or system property.

The values for the `-v` option and the `-ORBpeerValidate` command-line option or system property are as follows:

- `none`—no host matching is performed.
- `detect`—if the object reference used to make the SSL connection does not match the host name in the target's digital certificate, the IIOP Listener/Handler or the ORB does not authenticate the target and drops the SSL connection. The `detect` value is the default value.
- `warn`—if the object reference used to make the SSL connection does not match the host name in the target's digital certificate, the IIOP Listener/Handler or the ORB sends a message to the user log and continues processing.

If there is more than one IIOP Listener/Handler in a WebLogic Enterprise domain configured for SSL connections (for example, in the case of fault tolerance), BEA recommends using DNS alias names for the IIOP Listener/Handlers or creating different digital certificates for each IIOP Listener/Handler. The `-H` switch on the IIOP Listener can be used to specify the DNS alias name so that object references will be created correctly.

Setting the Encryption Strength

To set the encryption strength:

- Use the `-z` and `-Z` options of the ISL command to set the encryption strength in the IIOP Listener/Handler.
- Use the `-ORBminCrypto` and `-ORBmaxCrypto` command-line option or system property on the ORB to set the encryption strength in the CORBA C++ or CORBA Java ORB.

The `-z` option and the `-ORBminCrypto` command-line option or system property set the minimum level of encryption used when an application establishes an SSL connection with the IIOP Listener/Handler, the CORBA C++ ORB, or the CORBA

Java ORB. The valid values are 0, 40, 56, and 128. 0 means the data is signed but not sealed while 40, 56, and 128 specify the length (in bits) of the encryption key. If this minimum level of encryption is not met, the SSL connection fails. The default is 40.

The `-z` option and the `-ORBmaxCrypto` command-line option or system property set the maximum level of encryption used when an application establishes an SSL connection with the IIOP Listener/Handler, the CORBA C++ ORB, or the CORBA Java ORB. The valid values are 0, 40, 56, and 128. Zero means that data is signed but not sealed while 40, 56, and 128 specify the length (in bits) of the encryption key. The default minimum value is 40. The default maximum value is whatever capability is specified by the license.

The `-z` or `-Z` options and the `-ORBminCrypto` and `-ORBmaxCrypto` command-line options or system properties are available only if the WebLogic Enterprise Security Pack is installed.

To change the strength of encryption currently used in a WebLogic Enterprise application, you need to shut down the IIOP Listener/Handler or the ORB.

The combination in which you set the encryption values is important. The encryption values set in the initiator of an SSL connection need to be a subset of the encryption values set in the target of an SSL connection.

Table 6-1 lists combinations of encryption values and describes the encryption behavior.

Table 6-1 Combinations of Encryption Values

<code>-z</code> <code>-ORBminCrypto</code>	<code>-Z</code> <code>-ORBmaxCrypto</code>	Description
No value specified	No value specified	If the use of the SSL protocol is specified by some other command-line option or system property but no values are specified for <code>ORBminCrypto</code> and <code>ORBmaxCrypto</code> , these command-line options or system properties are assigned their default values.
0	No value specified	Maximum encryption defaults to the maximum value specified in the license. Tamper/replay detection and privacy protection are negotiated.
No value specified	0	Tamper/replay detection is negotiated. Privacy protection is not provided.

Table 6-1 Combinations of Encryption Values (Continued)

<code>-z</code> <code>-ORBminCrypto</code>	<code>-Z</code> <code>-ORBmaxCrypto</code>	Description
0	0	Tamper/replay detection is negotiated. Privacy protection is not provided.
40, 56, 128	No value specified	Maximum encryption defaults to the maximum value specified in the license. Privacy protection can be negotiated to the maximum allowed by the SSL license.
No value specified	40, 56, 12	Privacy protection can be negotiated to the value specified by the <code>-Z</code> option as long as it is less than the maximum allowed by the SSL license. The <code>-z</code> option defaults to 40.
40, 56, 128	40, 56, 128	Privacy protection can be negotiated between the values specified by the <code>-z</code> option up to the value specified by the <code>-Z</code> option as long as the values are less than the maximum allowed by the SSL license.

Note: In all combinations listed in Table 6-1, the value of the SSL license controls the maximum bit strength. If a bit strength is specified beyond the maximum licensed value, the IIOP Listener/Handler or ORB will not start and an error will be generated indicating the bit strength setting is invalid. Stopping the IIOP Listener/Handler or ORB from starting, instead of lowering the maximum value and giving only a warning, protects against an incorrectly configured application running with less protection than was expected.

If a cipher that exceeds the maximum licensed bit strength is somehow negotiated, the SSL connection is not established.

For a list of cipher suites supported by the WebLogic Enterprise Security Service, see [“Supported Cipher Suites” on page 2-11](#).

Setting the Interval for Session Renegotiation

Note: You set the interval for session renegotiation only in the IIOP Listener/Handler.

Use the `-R` option of the ISL command to control the time between session renegotiations. Periodic renegotiation of an SSL session refreshes the symmetric keys used to encrypt and decrypt information which limits the time a symmetric key is exposed. You can keep long-term SSL connections more secure by periodically changing the symmetric keys used for encryption.

The `-R` option specifies the renegotiation interval in minutes. If an SSL connection does renegotiate within the specified interval, the IIOP Listener/Handler will request the application to renegotiate the SSL session for inbound connections or actually perform the renegotiation in the case of outbound connections. The default is 0 minutes which results in no periodic session renegotiations.

You cannot use session renegotiation when enabling certificate-based authentication using the `-a` option of the ISL command.

Defining Security Parameters for the IIOP Listener/Handler

For the IIOP Listener/Handler to participate in SSL connections, the IIOP Listener/Handler authenticates itself to the peer that initiated the SSL connection. This authentication requires a digital certificate. The private key associated with the digital certificate is used as part of establishing an SSL connection that results in an agreement between the principal and the peer (in this case a client application and the IIOP Listener/Handler) on the session key. The session key is a symmetric key (as opposed to the private-public keys) that is used to encrypt data during an SSL session. You define the following information for the IIOP Listener/Handler so that it can be authenticated by peers:

- `SEC_PRINCIPAL_NAME`

Specifies the identity of the IIOP Listener/Handler.

- `SEC_PRINCIPAL_LOCATION`

Specifies the location of the private key file. For example, `$TUXDIR/udataobj/security/keys/milozzi.pem`.

- `SEC_PRINCIPAL_PASSVAR`

Specifies an environment variable that holds the pass phrase for the private key of the IIOP Listener/Handler when the `tmloadcf` command is not run interactively. Otherwise, you will be prompted for the pass phrase when you enter the `tmloadcf` command.

Note: If you define any of the security parameters for the IIOP Listener/Handler incorrectly, the following errors are reported in the ULOG file:

```
ISH.28014: LIBPLUGIN_CAT:2008:ERROR:No such file or
directory SEC_PRINCIPAL_NAME
ISH.28014:ISNAT_CAT:1552:ERROR:Could not open private key,
erro =-3011
ISH.28104:ISNAT_CAT:1544:ERROR:Could not perform SSL accept
from host/port//IPADDRESS:PORT
```

To resolve the errors, correct information in the the security parameters and reboot the IIO Listener/Handler.

These parameters are included in the part of the `SERVERS` section of the `UBBCONFIG` file that defines the ISL system process.

You also need to use the `tpusradd` command to define the IIOP Listener/Handler as an authorized user in the WebLogic Enterprise domain. You will be prompted for a password for the IIOP Listener/Handler. Enter the pass phrase you defined for `SEC_PRINCIPAL_PASSVAR`.

During initialization, the IIOP Listener/Handler includes its principal name as defined by `SEC_PRINCIPAL_NAME` as an argument when calling the authentication plug-in to acquire its credentials. An IIOP Listener/Handler requires credentials so that it can authenticate remote client applications that want to interact with the WebLogic Enterprise application, and get authorization and auditing tokens for remote client applications.

Because the IIOP Listener/Handler must authenticate its own identity to the WebLogic Enterprise domain in order to become a trusted system process, it is necessary to configure an authentication server when using the default authentication plug-in. See “Configuring the Authentication Server” on page 7-1 for more information.

Example of Setting Parameters on the ISL System Process

You set parameters for the SSL protocol in the portion of the `SERVERS` section of the `UBBCONFIG` that defines information for the ISL server process. Listing 6-1 includes code from a `UBBCONFIG` file that set parameters to configure the IIOP Listener/Handler for the SSL protocol and certificate-based authentication.

Listing 6-1 Using the ISL Command in the UBBCONFIG File

```
...
ISL
    SRVGRP = SYS_GRP
    SRVID  = 5
    CLOPT  = "-A -- -a -z40 -Z128 -S3579 -n //ICEPICK:2569
    SEC_PRINCIPAL_NAME="BLOTTO"
    SEC_PRINCIPAL_LOCATION="BLOTTO.pem"
    SEC_PRINCIPAL_VAR="AUDIT_PASS"
```

Example of Setting Command-Line Options on the CORBA C++ ORB

Listing 6-2 contains sample code that illustrates using the command-line options on the CORBA C++ ORB to configure the ORB for the SSL protocol.

Listing 6-2 Example of Setting the Command-Line Options on the CORBA C++ ORB

```
ChatClient    -ORBId BEA_IIOP
              -ORBsecurePort 2100
              -ORBminCrypto 40
              -ORBMaxCrypto 128
TechTopics
```

Example of Setting System Properties on the CORBA Java ORB

Listing 6-3 contains sample code that illustrates using the system properties of the CORBA Java ORB to configure the ORB for the SSL protocol.

Listing 6-3 Example of Setting the System Properties on the CORBA Java ORB

```
ChatClient    -DTOBJADDR=corbalocs://piglet:1900
              -Dorg.omg.CORBA=ORBPort=1948
              -classpath=%CLASSPATH% client
              -ORBMaxCrypto 128
```

7 Configuring Authentication

This topic includes the following sections:

- Configuring the Authentication Server
- Defining Authorized Users
- Defining a Security Level
- Configuring Application Password Security
- Configuring Username/Password Authentication
- Sample UBBCONFIG File for Username/Password Authentication
- Configuring Certificate-based Authentication
- Sample UBBCONFIG File for Certificate-based Authentication
- Configuring Access Control
- Configuring Security to Interoperate with Older WebLogic Enterprise Client Applications

Configuring the Authentication Server

Note: You only need to configure the authentication server, if you have specified a value of `USER_AUTH` or higher for the `SECURITY` parameter and are using the default authentication plug-in..

Authentication requires that an authentication server be configured for the purpose of authenticating users by checking their individual passwords against a file of legal users. The WebLogic Enterprise system uses a default authentication server called `AUTHSVR` to perform authentication. `AUTHSVR` provides a single service, `AUTHSVC`, which performs authentication. `AUTHSVC` is advertised by the `AUTHSVR` server as `AUTHSVC` when the security level is set to `ACL` or `MANDATORY_ACL`.

For a WebLogic Enterprise application to authenticate users, the value of the `AUTHSVC` parameter in the `RESOURCES` section of the `UBBCONFIG` file needs to specify the name of the process to be used as the authentication server for the WebLogic Enterprise application. The service must be called `AUTHSVC`. If the `AUTHSVC` parameter is specified in the `RESOURCES` section of the `UBBCONFIG` file, the `SECURITY` parameter must also be specified with a value of at least `USER_AUTH`. If the value is not specified, an error will occur when the system executes the `tmloadcf` command. If the `-m` option is configured on the `ISL` process in the `UBBCONFIG` file, the `AUTHSVC` must be defined in the `UBBCONFIG` file before the `ISL` process.

In addition, you need to define `AUTHSVR` in the `SERVERS` section of the `UBBCONFIG` file. The `SERVERS` section contains information about the server processes to be booted in the WebLogic Enterprise application. To add `AUTHSVC` to an application, you need to define `AUTHSVC` as the authentication service and `AUTHSVR` as the authentication server in the `UBBCONFIG` file. Listing 7-1 contains the portion of the `UBBCONFIG` file that defines the authentication server.

Listing 7-1 Parameters for the Authentication Server

```
*RESOURCES
SECURITY    USER_AUTH
AUTHSVC     AUTHSVC
.
.
.
```

```
*SERVERS
AUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600 MAXGEN=2
CLOPT="-A"
```

If you omit the parameter-value entry AUTHSVC, the WebLogic Enterprise system calls AUTHSVC by default.

AUTHSVR may be replaced with an authentication server that implements logic specific to the application. For example, a company may want to develop a custom authentication server so that it can use the popular Kerberos mechanism for authentication.

To add a custom authentication service to an application, you need to define your authentication service and server in the UBBCONFIG file. For example:

```
*RESOURCES
SECURITY      USER_AUTH
AUTHSVC       KERBEROS
.
.
.

*SERVERS
KERBEROSSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600
MAXGEN=2 CLOPT="-A"
```

Defining Authorized Users

As part of configuring security for a WebLogic Enterprise application, you need to define the principals and groups of principals who have access to the WebLogic Enterprise application.

Authorized users can be defined in the following ways:

- When using Username/Password authentication, authorized users are defined as a username and an associated password.
- When using certificate-based authentication, authorized users are identified by their e-mail address. The e-mail address maps the external identity of a principal represented by a digital certificate to an identity used by a WebLogic Enterprise application.

- When defining names of authorized users for a WebLogic Enterprise EJB, there is a one-to-one association between the users defined with the `tpusradd` command and the security roles defined in the deployment descriptor of the WebLogic Enterprise EJB.

You use the `tpusradd` command to create files containing lists of authorized principals. The `tpusradd` command adds a new principal entry to the WebLogic Enterprise security data files. This information is used by the authentication server to authenticate principals. The file that contains the principals is called `tpusr`.

The file is a colon-delimited, flat ASCII file, readable only by the system administrator of the WebLogic Enterprise application. The system file entries have a limit of 512 characters per line. The file is kept in the application directory, specified by the environment variable `$APPDIR`. The environment variable `$APPDIR` must be set to the pathname of the WebLogic Enterprise application.

The `tpusradd` file should be owned by the administrator account. BEA recommends that the file be protected so that only the owner has read and write privileges for the file and all other users have only read privileges for the file.

The `tpusradd` command has the following options:

`-u uid`

The user identification number. The UID must be a positive decimal integer below 128K. The UID must be unique within the list of existing identifiers for the application. The UID defaults to the next available (unique) identifier greater than 0.

`-c client_name`

A string of printable characters that specifies the name of the principal. The name may not contain a colon (:), pound sign (#), or a newline (n). The principal name must be unique within the list of existing principals for the WebLogic Enterprise application. The name of the principal can be either the name of a WebLogic Enterprise client application or a WebLogic Enterprise EJB.

Listing 7-2 includes a sample `tpusradd` file.

Listing 7-2 Sample `tpusradd` File

Cltname	Uid
milozzi	122
smart	555

```
patt          1234
butler        15555
```

Note: Use the `tpgrpadd` command to add groups of principals to the WebLogic Enterprise security data files.

In addition to the `tpusradd` and `tpgrpadd` commands, the WebLogic Enterprise product provides the following commands to modify the `tpusr` and `tpgrp` files:

- `tpusrdel`
- `tpusrmod`
- `tpgrpdel`
- `tpgrpmod`

For a complete description of the commands, see the [Commands, System Processes, and MIB Reference](#) in the WebLogic Enterprise online documentation.

You may already have files containing lists of users and groups on your host system. You can use them as the user and group files for your WebLogic Enterprise application, but only after converting them to the format required by the WebLogic Enterprise system. To convert your files, run the `tpaclevt` command, as shown in the following sample procedure. The sample procedure is written for a UNIX host machine.

1. Ensure that you are working on the application MASTER machine and that the application is inactive.
2. To convert the `/etc/password` file into the format needed by the BEA Tuxedo system, enter the following command:

```
tpaclevt -u /etc/password
```

This command creates the `tpusr` file and stores the converted data in it. If the `tpusr` file already exists, `tpaclevt` adds the converted data to the file, but it does *not* add duplicate user information to the file.

Note: For systems on which a shadow password file is used, you are prompted to enter a password for each user in the file.

3. To convert the `/etc/group` file into the format needed by the WebLogic Enterprise system, enter the following command:

```
tpaclevt -g /etc/group
```

This command creates the `tpgrp` file and stores the converted data in it. If the `tpgrp` file already exists, `tpac1cvr` adds the converted data to the file, but it does *not* add duplicate group information to the file.

Defining a Security Level

As part of defining security for a WebLogic Enterprise application, you need to define the `SECURITY` parameter in the `RESOURCES` section of the `UBBCONFIG` file. The `SECURITY` parameter has the following format:

```
*RESOURCES
    SECURITY {NONE|APP_PW|USER_AUTH|ACL|MANDATORY_ACL}
```

Table 7-1 describes the values for the `SECURITY` parameter.

Table 7-1 Values for the SECURITY Parameter

Value	Description
NONE	Indicates that no password or access checking is performed in the WebLogic Enterprise application. <code>Tobj::PrincipalAuthenticator::get_auth_type()</code> returns a value of <code>TOBJ_NOAUTH</code> .
APP_PW	Indicates that client applications are required to supply an application password to access the WebLogic Enterprise domain. The <code>tmloadcf</code> command prompts for an application password. <code>Tobj::PrincipalAuthenticator::get_auth_type()</code> returns a value of <code>TOBJ_SYSAUTH</code> .
USER_AUTH	Indicates that client applications and the IIOP Listener/Handler are required to authenticate themselves to the WebLogic Enterprise domain using a password. The value <code>USER_AUTH</code> is similar to <code>APP_PW</code> but, in addition, indicates that user authentication will be done during client initialization. The <code>tmloadcf</code> command prompts for an application password. <code>Tobj::PrincipalAuthenticator::get_auth_type()</code> returns a value of <code>TOBJ_APPAUTH</code> . No access control checking is performed at this security level.

Table 7-1 Values for the SECURITY Parameter (Continued)

Value	Description
ACL	<p>Indicates that authentication is used in the WebLogic Enterprise application and access control checks are performed on interfaces, services, queue names, and event names. If an associated ACL is not found for a name, it is assumed that permission is granted. The <code>tmloadcf</code> command prompts for an application password.</p> <p><code>Tobj::PrincipalAuthenticator::get_auth_type</code> returns a value of <code>TOBJ_APPAUTH</code>.</p>
MANDATORY_ACL	<p>Indicates that authentication is used in the WebLogic Enterprise application and access control checks are performed on interfaces, services, queue names, and event names. The value <code>MANDATORY_ACL</code> is similar to <code>ACL</code>, but permission is denied if an associated <code>ACL</code> is not found for the name. The <code>tmloadcf</code> command prompts for an application password.</p> <p><code>Tobj::PrincipalAuthenticator::get_auth_type</code> returns a value of <code>TOBJ_APPAUTH</code>.</p>

Note: If the IIOP Listener/Handler is configured for using certificate-based authentication, the value of the `SECURITY` parameter must be `USER_AUTH` or greater.

Configuring Application Password Security

To configure application password security, complete the following steps:

1. Ensure that you are working on the application `MASTER` machine and that the application is inactive.
2. Set the `SECURITY` parameter in the `RESOURCES` section of the `UBBCONFIG` file to `APP_PW`.
3. Load the configuration by running the `tmloadcf` command. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.

4. The system prompts you for a password. The password you enter may be up to 30 characters long. It becomes the password for the application and remains in effect until you change it by using the `passwd` parameter of the `tmadmin` command.
5. Distribute the application password to authorized users of the application through an off-line means such as telephone or letter.

Configuring Username/Password Authentication

Username/Password authentication requires that in addition to the application password, each client application must provide a valid username and user-specific data, such as a password, to interact with the WebLogic Enterprise application. The password must match the password associated with the username stored in the `tpusr` file. The checking of user passwords against the username/password combination in the `tpusr` file is carried out by the authentication service `AUTHSVC`, which is provided by the authentication server `AUTHSVR`.

To enable Username/Password authentication, complete the following steps:

1. Define users and their associated passwords in the `tpusr` file. For more information about the `tpusr` file, see [“Defining Authorized Users” on page 7-3](#).
2. Ensure that you are working on the application MASTER machine and that the application is inactive.
3. Open `UBBCONFIG` with a text editor and add the following lines to the `RESOURCES` and `SERVERS` sections:

```
*RESOURCES
SECURITY    USER_AUTH
AUTHSVC     AUTHSVC
.
.
.

*SERVERS
AUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600 MAXGEN=2
CLOPT="-A"
```

CLOPT="-A" causes the `tmboot` command to pass only the default command-line options (invoked by "-A") to `AUTHSVR` when the `tmboot` command starts the application.

4. Load the configuration by running the `tmloadcf` command. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.
5. The system prompts you for a password. The password you enter may be up to 30 characters long. It becomes the password for the application and remains in effect until you change it by using the `passwd` parameter of the `tmadmin` command.
6. Distribute the application password to authorized users of the application through an off-line means such as telephone or letter.

Sample UBBCONFIG File for Username/Password Authentication

Listing 7-4 includes a `UBBCONFIG` file for an application which uses Username/Password authentication. The key sections of the `UBBCONFIG` file are noted in bold face text.

Listing 7-3 Sample UBBCONFIG File for Username/Password Authentication

```
*RESOURCES
  IPCKEY      55432
  DOMAINID   securapp
  MASTER     SITE1
  MODEL      SHM
  LDBAL      N
SECURITY    USER_AUTH
AUTHSVR    "AUTHSVC"

*MACHINES
  "ICEAXE"
  LMID       = SITE1
  APPDIR     = "D:\M3\samples\corba\SECURAPP"
  TUXCONFIG  = "D:\M3\samples\corba\SECURAPP\results\tuxconfig"
```

7 *Configuring Authentication*

```
TUXDIR          = "D:\WLE5"
MAXWSCLIENTS    = 10

*GROUPS
  SYS_GRP
    LMID         = SITE1
    GRPNO        = 1
  APP_GRP
    LMID         = SITE1
    GRPNO        = 2

*SERVERS
  DEFAULT:
    RESTART      = Y
    MAXGEN       = 5

  AUTHSVR
    SRVGRP       = SYS_GRP
    SRVID        = 1
    RESTART      = Y
    GRACE        = 60
    MAXGEN       = 2

  TMSYSEVT
    SRVGRP       = SYS_GRP
    SRVID        = 1

  TMFFNAME
    SRVGRP       = SYS_GRP
    SRVID        = 2
    CLOPT        = "-A -- -N -M"

  TMFFNAME
    SRVGRP       = SYS_GRP
    SRVID        = 3
    CLOPT        = "-A -- -N"

  TMFFNAME
    SRVGRP       = SYS_GRP
    SRVID        = 4
    CLOPT        = "-A -- -F"

  simple_server
    SRVGRP       = APP_GRP
    SRVID        = 1
    RESTART      = N

  ISL
    SRVGRP       = SYS_GRP
    SRVID        = 5
```

```
CLOPT      = "-A -- -n //PCWIZ::2500"  
SEC_PRINCIPAL_NAME="IIOPListener"  
SEC_PRINCIPAL_PASSVAR="ISH_PASS"
```

Configuring Certificate-based Authentication

Certificate-based authentication uses the SSL protocol so you need to install the WebLogic Enterprise Security Pack and configure the SSL protocol before you can use certificate-based authentication. Information about installing the WebLogic Enterprise Security Pack can be found in the [BEA WebLogic Enterprise Installation Guide](#). For information about configuring the SSL protocol, see “[Configuring the SSL Protocol](#)” on page 6-1.

You also need an LDAP-enabled directory and certificate authority in place before using certificate-based authentication in a WebLogic Enterprise application. You can choose any LDAP-enabled directory service. You can also choose the certificate authority from which to obtain certificates and private keys used in a WebLogic Enterprise application. For more information, see “[Managing Public Key Security](#)” on page 4-1.

To enable certificate-based authentication, complete the following steps:

1. Install the WebLogic Enterprise Security Pack.
2. Set up an LDAP-enabled directory service.
3. Obtain a certificate and private key for the IIOP Listener/Handler from a certificate authority.
4. Obtain a certificate and private key for the WebLogic Enterprise application from a certificate authority.
5. Store the private keys for the WebLogic Enterprise application in the `Home` directory of the user or in the following directories:

Windows NT

```
%TUXDIR%\udataobj\security\keys
```

UNIX

`$TUXDIR/udataobj/security/keys`

6. Publish the certificates for the IIOP Listener/Handler, the WebLogic Enterprise application, and the certificate authority in the LDAP-enabled directory service.
7. Define the `SEC_PRINCIPAL`, `SEC_PRINCIPAL_LOCATION`, and `SEC_PRINCIPAL_PASSVAR` for the ISL server process in the `UBBCONFIG` file. For more information, see [“Defining Security Parameters for the IIOP Listener/Handler” on page 6-7](#).
8. Use the `tpusradd` command to define the authorized users of your WebLogic Enterprise application and IIOP Listener/Handler. Use the e-mail addresss of the user in the `tpusr` file. For more information about the `tpusr` file, see [“Defining Authorized Users” on page 7-3](#). Use the phase phrase you defined in `SEC_PRINCIPAL_PASSVAR` as the password for the IIOP Listener/Handler.
9. Define a port on the IIOP Listener/Handler for secure communications using the `-s` option of the ISL command. For more information, see [“Defining a Port for SSL Network Connections” on page 6-2](#).
10. Enable certificate-based authentication in the IIOP Listener/Handler using the `-a` option of the ISL command.
11. Create a Trusted Certificate Authority file (`trust_ca.cer`) that defines the certificate authorities trusted by the WebLogic Enterprise application. For more information, see [“Defining the Trusted Certificate Authorities” on page 4-8](#).
12. Open `UBBCONFIG` with a text editor and add the following lines to the `RESOURCES` and `SERVERS` sections:

```
*RESOURCES
SECURITY    USER_AUTH
```
13. Load the configuration by running the `tmloadcf` command. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.
14. Optionally, create a Peer Rules file (`peer_val.rul`) for both the WebLogic Enterprise application and the IIOP Listener/Handler. For more information, see [“Creating a Peer Rules File” on page 4-9](#).

15. Optionally, modify the LDAP search file filter to reflect the hierarchy in place in your enterprise. For more information, see [“Editing the LDAP Search Filter File” on page 4-5](#).

To enable certificate-based authentication, complete one of the following:

- Use the `-a` option of the `ISL` command to specify that certificate-based authentication must be used by applications connecting to the IIOP Listener/Handler.
- Use the `-ORBmutualAuth` command-line option or system property on the ORB to specify that certificate-based authentication must be used by applications connecting to the CORBA C++ or CORBA Java ORB.

Enabling certificate-based authentication requires the WebLogic Enterprise Security Pack to be installed. If the `-a` option or the `-ORBmutualAuth` command-line option or system property is executed and a license to enable the use of the SSL protocol does not exist, the IIOP Listener/Handler, CORBA C++ ORB, or CORBA Java ORB will not start.

Sample UBBCONFIG File for Certificate-based Authentication

Listing 7-4 includes a `UBBCONFIG` file for an application which uses certificate-based authentication. The key sections of the `UBBCONFIG` file are noted in bold face text.

Listing 7-4 Sample UBBCONFIG File for Certificate-based Authentication

```
*RESOURCES
  IPCKEY      55432
  DOMAINID    simpapp
  MASTER      SITE1
  MODEL       SHM
  LDBAL       N
  SECURITY   USER_AUTH

*MACHINES
```

```
"ICEAXE"
LMID      = SITE1
APPDIR    = "D:\M3\samples\corba\SIMPAP~1"
TUXCONFIG = "D:\M3\samples\corba\SIMPAP~1\results\tuxconfig"
TUXDIR    = "D:\WLE5"
MAXWSCLIENTS = 10

*GROUPS
SYS_GRP
    LMID      = SITE1
    GRPNO     = 1
APP_GRP
    LMID      = SITE1
    GRPNO     = 2

*SERVERS
DEFAULT:
RESTART = Y
MAXGEN  = 5

TMSYSEVT
    SRVGRP = SYS_GRP
    SRVID  = 1

TMFFNAME
    SRVGRP = SYS_GRP
    SRVID  = 2
    CLOPT  = "-A -- -N -M"

TMFFNAME
    SRVGRP = SYS_GRP
    SRVID  = 3
    CLOPT  = "-A -- -N"

TMFFNAME
    SRVGRP = SYS_GRP
    SRVID  = 4
    CLOPT  = "-A -- -F"

simple_server
    SRVGRP = APP_GRP
    SRVID  = 1
    RESTART = N

ISL
    SRVGRP = SYS_GRP
    SRVID  = 5
    CLOPT  = "-A -- -a -z40 -Z128 -S2458 -n //ICEAXE:2468"
    SEC_PRINCIPAL_NAME="IIOPListener"
```



```
SEC_PRINCIPAL_LOCATION="IIOPListener.pem"  
SEC_PRINCIPAL_PASSVAR="ISH_PASS"
```

Configuring Access Control

Note: Access control only applies to the default authorization implementation. The default authorization provider for CORBA does not enforce access control checks. In addition, the setting of the `SECURITY` parameter in the `UBBCONFIG` file does not control or enforce access control used by third-party authorization implementation.

There are two levels of access control security: optional access control list (ACL) and mandatory access control list (MANDATORY_ACL). Only when users are authenticated to join an application does the access control list become active.

By using an access control list, a system administrator can organize users into groups and associate the groups with objects that the member users have permission to access. Access control is done at the group level for the following reasons:

- System administration is simplified. It is easier to give a group of people access to a new object than it is to give individual users access to the object.
- Performance is improved. Because access permission needs to be checked for each invocation of an entity, permission should be resolved quickly. Because there are fewer groups than users, it is quicker to search through a list of privileged groups than it is to search through a list of privileged users.

When using the default authorization provider, the access control checking feature is based on the following files that are created and maintained by the system administrator:

- `tpusr` contains a list of users
- `tpgrp` contains a list of groups
- `tpacl` contains a list of ACLs

Configuring Optional ACL Security

The difference between `ACL` and `MANDATORY_ACL` is the following.

- In `ACL` mode, a service request will be allowed if there is not a specific `ACL`.
- In `MANDATORY_ACL` mode, the service request is denied if there is not a specific `ACL`.

When using multiple authorization implementations, all the implementations are placed under a single master authorization implementation. Each subordinate authorization implementation votes `PERMIT`, `DENY`, or `ABSTAIN` on the client request. The `ABSTAIN` return value is necessary for authorization implementation developers to gracefully handle unexpected situations that are not understood (such as new operation names that might be added in the future).

The rules for result determination are as follows:

1. If all authorization implementations permit, the operation is okd.
2. If any authorization implementation denies, the operation is denied.
3. If no authorization implementation denies and all implementations do not permit:
 - If the `SECURITY` parameter in the `UBBCONFIG` file is `MANDATORY_ACL`, the operation is denied.
 - If the `SECURITY` parameter in the `UBBCONFIG` file is `ACL`, the operation is allowed.

Optional `ACL` Security requires that each client provide an application password, a username, and user-specific data, such as a password, to join the application.

To configure optional `ACL` security, complete the following steps:

1. Ensure that you are working on the application `MASTER` machine and that the application is inactive.
2. Open `UBBCONFIG` with a text editor and add the following lines to the `RESOURCES` and `SERVERS` sections:

```
*RESOURCES
SECURITY    ACL
AUTHSVC     ..AUTHSVC
.
```

```

.
.
*SERVERS
AUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600 MAXGEN=2
CLOPT="-A"

```

CLOPT="-A" causes the `tmboot` command to pass only the default command-line options (invoked by "-A") to AUTHSVR when the `tmboot` command starts the application. By default, AUTHSVR uses the user information in the `tpusr` file to authenticate clients that want to interact with the WebLogic Enterprise application.

3. Load the configuration by running the `tmloadcf` command. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.
4. The system prompts you for a password. The password you enter may be up to 30 characters long. It becomes the password for the application and remains in effect until you change it by using the `passwd` command of `tmadmin`.
5. Distribute the application password to authorized users of the application through an off-line means such as telephone or letter.

Configuring Mandatory ACL Security

Mandatory ACL security level requires that each client provide an application password, a username, and user-specific data, such as a password, to interact with the WebLogic Enterprise application.

To configure mandatory ACL security, perform the following steps:

1. Ensure that you are working on the application MASTER machine and that the application is inactive.
2. Open `UBBCONFIG` with a text editor and add the following lines to the `RESOURCES` and `SERVERS` sections:

```

*RESOURCES
SECURITY    MANDATORY_ACL
AUTHSVC     ..AUTHSVC
.
.
.

```

```
*SERVERS
AUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600 MAXGEN=2
CLOPT="-A"
```

`CLOPT="-A"` causes the `tmboot` command to pass only the default command-line options (invoked by `-A`) to `AUTHSVR` when the `tmboot` command starts the application. By default, `AUTHSVR` uses the client user information in the `tpusr` file named to authenticate clients that want to join the application. The `tpusr` file resides in the directory referenced by the first pathname defined in the application's `APPDIR` variable.

3. Load the configuration by running the `tmloadcf` command. The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.
4. The system prompts you for a password. The password you enter may be up to 30 characters long. It becomes the password for the application and remains in effect until you change it by using the `passwd` command of `tmadmin`.
5. Distribute the application password to authorized users of the application through an off-line means such as telephone or letter.

Setting ACL Policy between WebLogic Enterprise Applications

As the administrator, you use the following configuration parameters to set and control the access control list (ACL) policy between WebLogic Enterprise applications that reside in different WebLogic Enterprise domains.

Parameter Name	Description	Setting
ACL_POLICY in DMCONFIG (TA_DMACLPOLICY in DM_MIB)	May appear in the DM_REMOTE_DOMAINS section of the DMCONFIG file for each remote domain access point. Its value for a particular remote domain access point determines whether or not the local domain gateway modifies the identity of service requests received from the remote domain.*	LOCAL or GLOBAL. Default is LOCAL. LOCAL means modify the identity of service requests, and GLOBAL means pass service requests with no change. DOMAINID string for the remote domain access point.

* A remote domain access point is also known as an RDOM (pronounced “are dom”) or simply *remote domain*.

The following bullets explain how the ACL_POLICY configuration affects the operation of local domain gateway (GWTDOMAIN) processes.

- When using a local ACL policy, each domain gateway (GWTDOMAIN) modifies inbound CORBA or EJB client requests (requests originating from the remote application and received over the network connection) so that they take on the DOMAINID for the remote domain access point and thus have the same access permissions as that identity. Each domain gateway passes outbound client requests without change.

In this configuration, each application has an ACL database containing entries *only* for users in its own domain.

- When using a global ACL policy, each domain gateway (GWTDOMAIN) passes inbound and outbound CORBA or EJB client requests without change. In this configuration, each application has an ACL database containing entries for users in its own domain *as well as* users in the remote domain.

Impersonating the Remote Domain Gateway

If the domain gateway receives a client request from a remote domain for which the ACL_POLICY parameter is set (or defaulted) to LOCAL in the local DMCONFIG file, the domain gateway removes any tokens from the request and creates an application key containing the DOMAINID of the remote domain access point.

Example DMCONFIG Entries for ACL Policy

In the following example, the connection through the remote domain access point b01 is configured for global ACL in the local DMCONFIG file, meaning that the domain gateway process for domain access point c01 passes client requests *from* and *to* domain access point b01 without change.

```
*DM_LOCAL_DOMAINS
# <LDOM name> <Gateway Group name> <domain type> <domain id>
#      [<connection principal name>] [<security>]...
c01    GWGRP=bankg1
        TYPE=TDOMAIN
        DOMAINID="BA.CENTRAL01"
        CONN_PRINCIPAL_NAME="BA.CENTRAL01"
        SECURITY=DM_PW
.
.
.

*DM_REMOTE_DOMAINS
# <RDOM name> <domain type> <domain id> [<ACL policy>]
#      [<connection principal name>] [<local principal name>]...
b01    TYPE=TDOMAIN
        DOMAINID="BA.BANK01"
        ACL_POLICY=GLOBAL
        CONN_PRINCIPAL_NAME="BA.BANK01"
```

Configuring Security to Interoperate with Older WebLogic Enterprise Client Applications

It may be necessary for CORBA and EJB server applications in a WebLogic Enterprise domain to securely interoperate with client applications that were built with the security features available in the 4.2 and 5.0 releases of the WebLogic Enterprise product. To allow CORBA or EJB server applications to interoperate with older, secure client applications, you need to either set the CLOPT -t option in the UBBCONFIG file or specify the -ORBinterOp command-line option or system property on the CORBA object request broker (ORB).

By setting the `CLOPT -t` option or specifying the `-ORBinterOP` command-line option or system property, you are lowering the effective level of security for a CORBA or EJB server. Therefore, the use of compatibility mode should be carefully considered before enabling the mode in a server application.

You need to set the `CLOPT -t` option on any server applications that will interoperate with the older client application. The `CLOPT -t` option is specified in the `*SERVERS` section of the `UBBCONFIG` file.

Listing 7-5 Example UBBCONFIG File Entries for Interoperability

```
*SERVERS
SecureSrv      SRVGRP=group_name SRVID=server_number
                CLOPT=A -t..
```

If you are using a remote CORBA C++ or CORBA Java ORB, specify the `-ORBinterOp` command-line option or system property on the ORB to allow the ORB to interoperate with client application using the security features in the 4.2 or 5.0 releases of the WebLogic Enterprise product.

8 Configuring Single Sign-on

This topic includes the following sections:

- Single Sign-on with Username/Password Authentication
- Single Sign-on with Username/Password Authentication and the SSL Protocol
- Single Sign-on with the SSL Protocol and Certificate-Based Authentication

Single Sign-on with Username/Password Authentication

The steps for implementing single sign-on with username/password authentication are as follows:

1. In the `CORBA.connectionpool` section of the `weblogic.properties` file define the following properties:

- `appaddrlist=//host:port`

where the `host` and `port` specify the name and port number of the IIOP Listener/Handler in the WebLogic Enterprise domain you want to access. For more information about the different address formats supported in the WebLogic Enterprise product, see “Understanding the Address Formats of the Bootstrap Object” on page 10-2.

- `username` as the name of the WebLogic Server principal.
- `userpassword` as the password for the WebLogic Server principal
- `apppassword` as the password of the WebLogic Enterprise application you want to access.
- `securitycontext` as Yes. Yes indicates that you want the security context of the WebLogic Server principal passed to the WebLogic Enterprise domain.

Note: There are other properties in the `CORBA.connectionpool` section of the `weblogic.properties` file that are used to set up the connection pool. For more information about setting up CORBA connection pools, see *Using WebLogic Enterprise Connectivity* in the WebLogic Server portion of the WebLogic Enterprise online documentation.

2. Use the `tpusradd` command to define the WebLogic Server principal as an authorized user in the WebLogic Enterprise domain. The username and password for the WebLogic Server principal must appear in the `tpusr` file exactly as they are defined in the `weblogic.properties` file.
3. Set `-E` option of the `ISL` command to configure the IIOP Listener/Handler to detect and utilize the propagated security context from the WebLogic Server realm. The `-E` option of the `ISL` command requires you to specify a principal name. The principal name is the username as defined in the `weblogic.properties` file. The `ISL` command for the IIOP Listener/Handler is defined for the `CLOPT` parameter in the `UBBCONFIG` file for the WebLogic Enterprise domain.
4. Set the `SECURITY` parameter in the `UBBCONFIG` file to `USER_AUTH` or higher.

Single Sign-on with Username/Password Authentication and the SSL Protocol

The steps for implementing single sign-on with username/password authentication and the SSL protocol are as follows:

1. Configure the SSL protocol in the WebLogic Server and the WebLogic Enterprise environments.

For information about configuring the SSL protocol in the WebLogic Server environment, see *Using WebLogic SSL* in the WebLogic Server portion of the WebLogic Enterprise online documentation.

For information about configuring the SSL protocol in the WebLogic Enterprise environment, see “The SSL Protocol” on page 3-20.

2. In the `CORBA.connectionpool` section of the `weblogic.properties` file define the following properties:

- `appaddrlist=corbalocs://host:port`

where the `host` and `port` specify the name and port number of the IIOP Listener/Handler in the WebLogic Enterprise domain you want to access. For more information about the different address formats supported in the WebLogic Enterprise product, see “Understanding the Address Formats of the Bootstrap Object” on page 10-2.

- `username` as the name of the WebLogic Server principal.
- `userpassword` as the password for the WebLogic Server principal
- `apppassword` as the password of the WebLogic Enterprise application you want to access.
- `securitycontext` as `Yes`. `Yes` indicates that you want the security context of the WebLogic Server principal passed to the WebLogic Enterprise domain.
- `minencryptionlevel` and `maxencryptionlevel`. These are optional properties. The valid values are 0, 40, 56, and 128. The default is 40 for the `minencryptionlevel` property. The `maxencryptionlevel` property defaults to the maximum strength allowed by the license. These two properties are used at the time of the SSL handshake to determine the encryption strength that will be used between the WebLogic Server and WebLogic Enterprise environments.

Note: There are other properties in the `CORBA.connectionpool` section of the `weblogic.properties` file that are used to set up CORBA connection pools. For more information about setting up connection pools, see *Using WebLogic Enterprise Connectivity* in the WebLogic Server portion of the WebLogic Enterprise online documentation.

3. Use the `tpusradd` command to define the WebLogic Server principal as an authorized user in the WebLogic Enterprise domain. The username and password for the WebLogic Server principal must appear in the `tpusr` file exactly as they are defined in the `weblogic.properties` file.
4. Set `-E` option of the `ISL` command to configure the IIOP Listener/Handler to detect and utilize the propagated security context from the WebLogic Server realm. The `-E` option of the `ISL` command requires you to specify a principal name. The principal name is the username as defined in the `weblogic.properties` file. The `ISL` command for the IIOP Listener/Handler is defined for the `CLOPT` parameter in the `UBBCONFIG` file for the WebLogic Enterprise domain.
5. Set the `SECURITY` parameter in the `UBBCONFIG` file to `USER_AUTH` or higher.

Single Sign-on with the SSL Protocol and Certificate-Based Authentication

The steps for implementing single sign-on with the SSL protocol and certificate-based authentication are as follows:

1. Configure the SSL protocol in the WebLogic Server and the WebLogic Enterprise environments.

For information about configuring the SSL protocol in the WebLogic Server environment, see *Using WebLogic SSL* in the WebLogic Server portion of the WebLogic Enterprise online documentation.

For information about configuring the SSL protocol in the WebLogic Enterprise environment, see “The SSL Protocol” on page 3-20.

2. In the `CORBA.connectionpool` section of the `weblogic.properties` file define the following properties:
 - `appaddrlist=corbalocs://host:port`
where the `host` and `port` specify the name and port number of the IIOP Listener/Handler in the WebLogic Enterprise domain you want to access.
 - `username` as email address of the subject of the digital certificate.

- `userpassword` as private key of the digital certificate.
- `apppassword` as the password of the WebLogic Enterprise application you want to access.
- `securitycontext` as `Yes`. `Yes` indicates that you want the security context of the WebLogic Server principal passed to the WebLogic Enterprise domain.
- `minencryptionlevel` and `maxecryptionlevel`. These are optional properties. The valid values are 0, 40, 56, and 128. The default is 40 for the `minencryptionlevel` property. The `maxecryptionlevel` property defaults to the maximum strength allowed by the license. These two properties are used at the time of the SSL handshake to determine the encryption strength that will be used between the WebLogic Server and WebLogic Enterprise environments.
- `certificatebasedauth` as `Yes`. `Yes` indicates that certificate-based authentication is to be used.

Note: There are other properties in the `CORBA.connectionpool` section of the `weblogic.properties` file that are used to set up the CORBA connection pool. For more information about setting up connection pools, see *Using WebLogic Enterprise Connectivity* in the WebLogic Server portion of the WebLogic Enterprise online documentation.

3. Use the `tpusradd` command to define the WebLogic Server principal as an authorized user in the WebLogic Enterprise domain. The username and password for the WebLogic Server principal must appear in the `tpusr` file exactly as they are defined in the `weblogic.properties` file.
4. Set `-E` option of the ISL command to configure the IIOP Listener/Handler to detect and utilize the propagated security context from the WebLogic Server realm. The `-E` option of the ISL command requires you to specify a principal name. The principal name is the username as defined in the `weblogic.properties` file. The ISL command for the IIOP Listener/Handler is defined for the `CLOPT` parameter in the `UBBCONFIG` file for the WebLogic Enterprise domain.
5. Set the `-a` option of the ISL command to configure the IIOP Listener/Handler to enable certificate-based authentication. The ISL command for the IIOP Listener/Handler is defined for the `CLOPT` parameter in the `UBBCONFIG` file for the WebLogic Enterprise domain.
6. Set the `SECURITY` parameter in the `UBBCONFIG` file to `USER_AUTH` or higher.

Using certificate-based authentication between the WebLogic Server environment and the WebLogic Enterprise environment implies performing a new SSL handshake to establish a connection from the WebLogic Server environment on a CORBA object, RMI object, or EJB in a WebLogic Enterprise environment is initiated. In order to support multiple client requests over the same SSL network connection, certificate-based authentication must be set up as follows:

- Obtain a digital certificate for the WebLogic Enterprise Connectivity process. This digital certificate is presented to the WebLogic Enterprise environment for the purpose of authenticating the identity of the WebLogic Enterprise Connectivity process. Once established, the authenticated connection between the WebLogic Enterprise Connectivity product and the WebLogic Enterprise environment remains.
- When a client request is made from the WebLogic Server environment on a CORBA object, RMI object, or EJB in the WebLogic Enterprise environment, digital certificates are exchanged between the WebLogic Server and WebLogic Enterprise environments and session keys are generated for both sides of the connection. Because WebLogic Connectivity is part of WebLogic Server, the WebLogic Connectivity process will accept any message from WebLogic Enterprise that has the sessions keys that were created when the SSL connection was established between the WebLogic Server and WebLogic Enterprise environments. The WebLogic Enterprise Connectivity process then forwards the client request using the established SSL connection to the WebLogic Enterprise environment.

9 Configuring Security Plug-Ins

This topic includes the [Registering the Security Plug-Ins \(SPIs\)](#) section.

Registering the Security Plug-Ins (SPIs)

Most BEA middleware products use a common transaction processing (TP) infrastructure that consists of a set of core services, such as security. The TP infrastructure is available to WebLogic Enterprise applications through well defined interfaces. These interfaces allow system administrators to change the default behavior of the TP infrastructure by loading and linking their own service code modules, referred to as security plug-ins.

In order to use a security plug-in, you need to register the security plug-in with the WebLogic Enterprise system. The registry of the WebLogic Enterprise system is a disk-based repository for storing information related to the security plug-ins. Initially, this registry holds information about the default security plug-ins. Additional entries are made to the registry as custom security plug-ins are added to the WebLogic Enterprise system. The registry entry for a security plug-in is a set of binary files that stores information about the plug-in. There is one registry per WebLogic Enterprise installation. Every client application, server application, and server machine in a particular WebLogic Enterprise application must use the same set of security plug-ins.

The registry is located in the following directory:

Windows NT

\$TUXDIR/udataobj

UNIX

\$TUXDIR/udataobj

The system administrator of a WebLogic Enterprise application in which custom security plug-ins are used is responsible for registering those plug-ins. A system administrator can register security plug-ins in the registry of the WebLogic Enterprise system only from the local machine. That is, a system administrator cannot register security plug-ins while logged on to the host machine from a remote location.

The following commands are available for managing security plug-ins:

- `epifregplugin`—for registering a security plug-in
- `epifunregplugin`—for unregistering a security plug-in
- `epifregedit`—for editing registry information

Instructions for using these commands are available in *Guide to Providing Security Services for BEA Products*. (This document contains the specifications for the Security SPIs, and describes the WebLogic Enterprise plug-in framework feature that makes the dynamic loading and linking of security plug-ins possible.) To obtain this document, see your BEA account executive.

When installing custom security plug-ins, the security vendor that provided the plug-in should provide instructions for using the commands to set up the registry for the WebLogic Enterprise system in order to access the customer security plug-ins.

Part III Security Programming

Chapter 10. Writing a WebLogic Enterprise CORBA Application that Implements Security

Chapter 11. Building and Running the CORBA Sample Applications

Chapter 12. Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security

Chapter 13. Troubleshooting

10 Writing a WebLogic Enterprise CORBA Application that Implements Security

This topic contains the following sections:

- Understanding the Address Formats of the Bootstrap Object
- Using Username/Password Authentication
- Using Certificate-based Authentication
- Using the `Invocations_Options_Required()` Method

Understanding the Address Formats of the Bootstrap Object

The Bootstrap object in the WebLogic Enterprise product has been enhanced so that users can specify that all communication to a given IIOP Listener/Handler be protected. The Bootstrap object supports `corbaloc` and `corbalocs` Uniform Resource Locator (URL) address formats to be used when specifying the location of the IIOP Listener/Handler. The type of security provided depends on the format of URL used to specify the location of the IIOP Listener/Handler.

As with the Host and Port address format, you use the URL address formats to specify the location of the IIOP Listener/Handler, but the bootstrapping process behaves differently. When using the `corbaloc` or `corbalocs` URL address format, the initial connection to the IIOP Listener/Handler is deferred until either:

- The principal uses Username/Password authenticate with either the `Tobj::PrincipalAuthenticator::logon` or the `SecurityLevel2::PrincipalAuthenticator::authenticate` methods.
- The principal calls the `Tobj_Bootstrap::resolve_initial_references` method using an object ID value other than `SecurityCurrent`.

Using the `corbalocs` URL address format indicates that the SSL protocol is used to protect at least the integrity of the connection between the principal and the IIOP Listener/Handler.

Table 10-1 highlights the differences between the two URL address formats.

Table 10-1 Differences Between corbaloc and corbalocs URL Address Formats

URL Address Formats	Functionality
corbaloc	By default, invocations on the IIOP Listener/Handler are unprotected. Configuring the IIOP Listener/Handler for the SSL protocol is optional. A principal can secure the bootstrapping process by using the <code>authenticate()</code> method of the <code>SecurityLevel2::PrincipalAuthenticator</code> interface and the <code>invocation_options_required()</code> method of the <code>SecurityLevel2::Credentials</code> interface to specify that certificate-based authentication is to be used.
corbalocs	Invocations on the IIOP Listener/Handler are protected and the IIOP Listener/Handler, the CORBA C++ ORB, or the CORBA Java ORB must be configured to enable the use of the SSL protocol. For more information, see “Configuring the SSL Protocol” on page 6-1.

Both the `corbaloc` and `corbalocs` URL address formats provide stringified object references that are easily manipulated in both TCP/IP and Domain Name System (DNS) environments. The `corbaloc` and `corbalocs` URL address formats contain a DNS-style host name or an IP address and port.

The URL address formats follow and extend the definition of object URLs adopted by the Object Management Group (OMG) as part of the Interoperable Naming Service submission. The WebLogic Enterprise software also extends the URL format described in the OMG Interoperable Naming Service submission to support a secure form that is modeled after the URL for secure HTTP, as well as to support functionality in previous releases of the WebLogic Enterprise product.

Listing 10-1 contains examples of the new URL address formats.

Listing 10-1 Examples of the corbaloc and corbalocs URL Address Formats

```
corbaloc://555xyz.com:1024,corbaloc://555backup.com:1022,  
corbaloc://555last.com:1999  
corbalocs://555xyz.com:1024,(corbalocs://555backup.com:1022|corba  
locs://555last.com:1999)  
corbaloc://555xyz.com:1111  
corbalocs://24.128.122.32:1011, corbalocs://24.128.122.34
```

As an enhancement to the URL syntax described in the OMG Interoperable Naming Service submission, the WebLogic Enterprise product extends the syntax to support a list of multiple URLs, each with a different scheme. Listing 10-2 contains examples of specifying multiple URLs.

Listing 10-2 Examples of Specifying Multiple URL Address Formats

```
corbalocs://555xyz.com:1024,corbaloc://555xyz.com:1111  
corbalocs://ctxobj.com:3434,corbalocs://mthd.com:3434,corbaloc://force.com:1111
```

In the examples in Listing 10-2, if the parser reaches the URL `corbaloc://force.com:1111`, it resets its internal state as if it had never attempted secure connections, and then begins attempting unprotected connections. This situation occurs if the client application has not set any SSL parameters on the Credentials object.

The following sections describe the behavior when using the different address formats of the Bootstrap object.

Using the Host and Port Address Format

If a WebLogic Enterprise client application uses the Host and Port address format of the Bootstrap object, the constructor method of the Bootstrap object constructs an object reference using the specified host name and port number. The invocation to the IIOP Listener/Handler is made without the protections offered by the SSL protocol.

The client application can still authenticate using Username/Password authentication. However, since the bootstrapping process is performed over an unprotected and unverified link, all communications are vulnerable to the following security attacks:

- The Man-in-the-Middle attack, because there was no verification that the principal to which the connection was made was the desired principal.
- The Denial of Service attack, because no object references were returned, the object references returned were invalid, or the security token was invalid.
- The Sniffer attack, because the information was sent in the clear so that anyone with a packet sniffer can see the content of a message that was not encrypted (for example, only the username/password information is encrypted).

- The Tamper attack, because the integrity of the information is not protected. The contents of the message could be changed and the change would not be detected.
- The Replay attack, because the same request can be sent repeatedly without detection.

Note: If the IIOP Listener/Handler is configured for the SSL protocol and the Host and Port address format of the Bootstrap object is used, the invocation on the specified WebLogic Enterprise object results in a `INVALID_DOMAIN` exception.

Using the corbaloc URL Address Format

By default, the invocation on the IIOP Listener/Handler is unprotected when using the `corbaloc` URL address format and Username/Password authentication. Therefore, all communications are vulnerable to the following security attacks:

- The Man-in-the-Middle attack, because there was no verification that the principal to which the connection was made was the desired principal.
- The Denial of Service attack, because no object references were returned, the object references returned were invalid, or the security token was invalid.
- The Sniffer attack, because the information was sent in the clear so that anyone with a packet sniffer can see the content of a message that was not encrypted (for example, only the username/password information is encrypted).
- The Tamper attack, because the integrity of the information is not protected. The content of the message could be changed and the change would not be detected.
- The Replay attack, because the same request can be sent repeatedly without detection.

You can protect the bootstrapping process when using the `corbaloc` URL address format by using the

`SecurityLevel2::PrincipalAuthenticator::authenticate()` method, specifying that certificate-based authentication is to be used, and setting the `invocation_methods_required` method on the `Credentials` object.

Note: If the IIOP Listener/Handler is configured for the SSL protocol and not for certificate-based authentication and the `corbaloc` URL address format is used, the invocation on the specified WebLogic Enterprise object results in an `INVALID_DOMAIN` exception.

BEA recommends that existing WebLogic Enterprise applications migrate to the `corbaloc` URL address format instead of using the Host and Port Address format.

Using the `corbalocs` URL Address Format

The `corbalocs` URL address format is the recommended format to use to ensure that communications between principals and the IIOP Listener/Handler are protected. The `corbalocs` URL address format functions in the same way as the `corbaloc` URL address format, except the SSL protocol is used to protect all communications with the IIOP Listener/Handler, the CORBA C++ ORB, or the CORBA Java ORB regardless of the type of authentication used.

When the defaults are used with the `corbalocs` URL address format, communications are vulnerable only to Denial of Service security attacks. Using the SSL protocol and certificate-based authentication guards against Sniffer, Tamper, and Replay attacks. In addition, the validation check of the host specified in the digital certificate guards against Man-in-the-Middle attacks.

To use the `corbalocs` URL address format, the IIOP Listener/Handler, the CORBA C++ ORB, or the CORBA Java ORB must be configured to enable the use of the SSL protocol. For more information about configuring the IIOP Listener/Handler, the CORBA C++ ORB, or the CORBA Java ORB for the SSL protocol, see “Configuring the SSL Protocol” on page 6-1.

Using Username/Password Authentication

This section describes implementing Username/Password authentication in WebLogic Enterprise applications.

The Security Sample Application

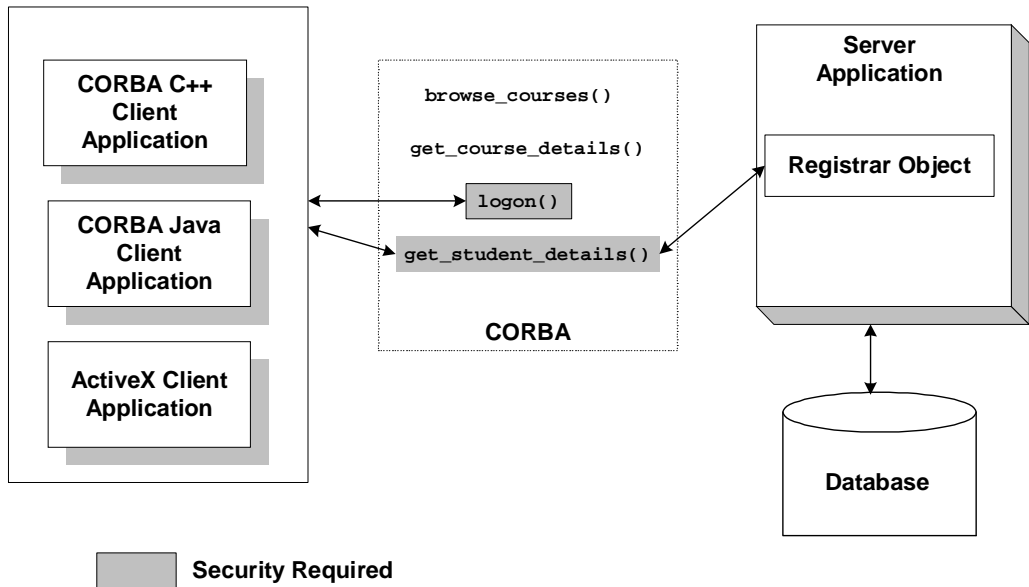
The Security sample application demonstrates Username/Password authentication.

The Security sample application requires each student using the application to have an ID and a password. The Security sample application works in the following manner:

1. The client application has a `logon` method. This method invokes operations on the `PrincipalAuthenticator` object, which is obtained as part of the process of logging on to access the domain.
2. The server application implements a `get_student_details()` method on the `Registrar` object to return information about a student. After the user is authenticated and the logon is complete, the `get_student_details()` method accesses the student information in the database to obtain the student information needed by the client logon method.
3. The database in the Security sample application contains course and student information.

Figure 10-1 illustrates the Security sample application.

Figure 10-1 Security Sample Application



The source files for the Security sample application are located in the `\samples\corba\university` directory in the WebLogic Enterprise software. For information about building and running the Security sample application, see the [Guide to the University Sample Applications](#).

Writing the Client Application

When using Username/Password authentication, write client application code that does the following:

1. Uses the Bootstrap object to obtain a reference to the SecurityCurrent object for the specific WebLogic Enterprise domain. You can use the Host and Port Address format, the `corbaloc` URL address format, or the `corbalocs` URL address format.
2. Gets the PrincipalAuthenticator object from the SecurityCurrent object.
3. Uses one of the following methods to authenticate the principal:

- C++—`SecurityLevel2::PrincipalAuthenticator::authenticate()`
using `Tobj::TuxedoSecurity`
- Java—`SecurityLevel2.PrincipalAuthenticator.authenticate()`
using `Tobj::TuxedoSecurity`
- C++—`Tobj::PrincipalAuthenticator::logon()`
- Java—`Tobj.PrincipalAuthenticator.logon()`

The `SecurityLevel2::PrincipalAuthenticator` interface is defined in the CORBAServices Security Service specification. This interface contains two methods that are used to accomplish the authentication of the principal. There are two methods because authentication of principals may require more than one step. The `authenticate()` method allows the caller to authenticate and optionally select attributes for the principal of this session.

The WebLogic Enterprise product extends the `PrincipalAuthenticator` object with functionality to support similar security to that found in BEA Tuxedo. The enhanced functionality is provided by the `Tobj::PrincipalAuthenticator` interface.

The methods defined for the `Tobj::PrincipalAuthenticator` interface provide a focused, simplified form of the equivalent CORBA-defined interface. You can use either the CORBA-defined or the WebLogic Enterprise extensions when developing a WebLogic Enterprise application.

The `Tobj::PrincipalAuthenticator` interface provides the same functionality as the `SecurityLevel2::PrincipalAuthenticator` interface. However, unlike the

`SecurityLevel2::PrincipalAuthenticator::authenticate()` method, the `logon()` method of the `Tobj::PrincipalAuthenticator` interface does not return a `Credentials` object. As a result, WebLogic Enterprise applications that need to use more than one principal identity are required to call the `Current::get_credentials()` method immediately after the `logon()` method to retrieve the `Credentials` object as a result of the `logon`. Retrieval of the `Credentials` object directly after a `logon` method should be protected with serialized access.

The following sections contain C++ and Java code examples that illustrate implementing Username/Password authentication. For a Visual Basic code example, see “Automation Security Reference” on page 18-1.

C++ Code Example of Using the SecurityLevel2::PrincipalAuthenticator::authenticate() Method

Listing 10-3 contains C++ code that performs Username/Password authentication using the SecurityLevel2::PrincipalAuthenticator::authenticate() method.

Listing 10-3 C++ Client Application That Uses the SecurityLevel2::PrincipalAuthenticator::authenticate() Method

```
...
//Create Bootstrap object
    Tobj_Bootstrap* bootstrap = new Tobj_Bootstrap(orb,
        corbalocs://sling.com:2143);

//Get SecurityCurrent object
CORBA::Object_var var_security_current_oref =
    bootstrap.resolve_initial_references("SecurityCurrent");
SecurityLevel2::Current_var var_security_current_ref =
    SecurityLevel2::Current::_narrow(var_security_current_oref.in());

//Get the PrincipalAuthenticator
SecurityLevel2::PrincipalAuthenticator_var var_principal_authenticator_oref =
    var_security_current_oref->principal_authenticator();

const char * user_name = "john"
const char * client_name = "university";
char system_password[31] = {'\0'};
char user_password[31] = {'\0'};

//Determine the security level
Tobj::AuthType auth_type = var_bea_principal_authenticator->get_auth_type();
switch (auth_type)
{
    case Tobj::TOBJ_NOAUTH;
        break;

    case Tobj::TOBJ_SYSAUTH
        strcpy(system_password, "sys_pw");

    case Tobj::TOBJ_APPAUTH
        strcpy(system_password, "sys_pw");
        strcpy(user_password, "john_pw");
        break;
}
```

```
if (auth_type != Tobj::TOBJ_NOAUTH)

{
    SecurityLevel2::Credentials_var      creds;
    Security::Opaque_var                 auth_data;
    Security::AttributeList_var          privileges;
    Security::Opaque_var                 cont_data;
    Security::Opaque_var                 auth_spec_data;

    var_bea_principalauthenticator->build_auth_data(user_name,
                                                    client_name,
                                                    system_password,
                                                    user_password,
                                                    NULL,
                                                    auth_data,
                                                    privileges);

    Security::AuthenticationStatus status =
        var_bea_principalauthenticator->authenticate(
                                                    Tobj::TuxedoSecurity,
                                                    user_name,
                                                    auth_data,
                                                    privileges,
                                                    creds,
                                                    cont_data, auth_spec_data);

    if (status != Security::SecAuthSuccess)
    {
        //Failed authentication
        return;
    }
}

// Proceed with application
...
```

Java Code Example of Using the `SecurityLevel2.PrincipalAuthenticator.authenticate()` Method

Listing 10-4 contains Java code that performs Username/Password authentication using the `SecurityLevel2.PrincipalAuthenticator.authenticate()` method.

Listing 10-4 Java Client Application That Uses the SecurityLevel2.PrincipalAuthenticator.authenticate() Method

```
...
// Create Bootstrap object
Tobj_Bootstrap bs =
    new Tobj_Bootstrap(orb, corbalocs://sling.com:2143);

// Get SecurityCurrent object
org.omg.CORBA.Object secCurObj =
    bs.resolve_initial_references( "SecurityCurrent" );
org.omg.SecurityLevel2.Current secCur2Obj =
    org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);

// Get Principal Authenticator
org.omg.Security.PrincipalAuthenticator princAuth =
    secCur2Obj.principal_authenticator();
com.beasys.Tobj.PrincipalAuthenticator auth =
    Tobj.PrincipalAuthenticatorHelper.narrow(princAuth);

// Get Authentication type
com.beasys.Tobj.AuthType authType = auth.get_auth_type();

// Initialize arguments
String userName = "John";
String clientName = "Teller";
String systemPassword = null;
String userPassword = null;
byte[] userData = new byte[0];

// Prepare arguments according to security level requested
switch(authType.value())
{
    case com.beasys.Tobj.AuthType._TPNOAUTH:
        break;

    case com.beasys.Tobj.AuthType._TPSYSAUTH:
        systemPassword = "sys_pw";
        break;

    case com.beasys.Tobj.AuthType._TPAPPAUTH:
        systemPassword = "sys_pw";
        userPassword = "john_pw";
        break;
}

// Build security data
org.omg.Security.OpaqueHolder auth_data =
    new org.omg.Security.OpaqueHolder();
```

```
org.omg.Security.AttributeListHolder privs =
    new Security.AttributeListHolder();
auth.build_auth_data(userNname, clientName, systemPassword,
    userPassword, userData, authData,
    privs);

// Authenticate user
org.omg.SecurityLevel2.CredentialsHolder creds =
    new org.omg.SecurityLevel2.CredentialHolder();
org.omg.Security.OpaqueHolder cont_data =
    new org.omg.Security.OpaqueHolder();
org.omg.Security.OpaqueHolder auth_spec_data =
    new org.omg.Security.OpaqueHolder();

org.omg.Security.AuthenticationStatus status =
    auth.authenticate(com.beasys.Tobj.TuxedoSecurity.value,
        0, userName, auth_data.value(),
        privs.value(), creds, cont_data,
        auth_spec_data);
if (status != AuthenticatoInStatus.SecAuthSuccess)
    System.exit(1);
}
```

...

C++ Code Example of Using the Tobj::PrincipalAuthenticator::logon() Method

Listing 10-5 contains C++ code that performs Username/Password authentication using the Tobj::PrincipalAuthenticator::logon() method.

Listing 10-5 C++ Client Application That Uses the Tobj::PrincipalAuthenticator::logon() Method

```
...
CORBA::Object_var var_security_current_oref =
    bootstrap.resolve_initial_references("SecurityCurrent");
SecurityLevel2::Current_var var_security_current_ref =
    SecurityLevel2::Current::_narrow(var_security_current_oref.in());

//Get the PrincipalAuthenticator
SecurityLevel2::PrincipalAuthenticator_var var_principal_authenticator_oref =
    var_security_current_oref->principal_authenticator();

//Narrow the PrincipalAuthenticator
Tobj::PrincipalAuthenticator_var var_bea_principal_authenticator =
```

10 Writing a WebLogic Enterprise CORBA Application that Implements Security

```
Tobj::PrincipalAuthenticator::_narrow
                                var_principal_authenticator_oref.in());

const char * user_name = "john"
const char * client_name = "university";
char system_password[31] = {'\0'};
char user_password[31] = {'\0'};

//Determine the security level
Tobj::AuthType auth_type = var_bea_principal_authenticator->get_auth_type();
switch (auth_type)
{
    case Tobj::TOBJ_NOAUTH;
        break;

    case Tobj::TOBJ_SYSAUTH
        strcpy(system_password, "sys_pw");

    case Tobj::TOBJ_APPAUTH
        strcpy(system_password, "sys_pw");
        strcpy(user_password, "john_pw");
        break;
}
if (auth_type != Tobj::TOBJ_NOAUTH)
{
    SecurityLevel2::Credentials_var          creds;
    Security::Opaque_var                    auth_data;
    Security::AttributeList_var              privileges;
    Security::Opaque_var                    cont_data;
    Security::Opaque_var                    auth_spec_data;
    var_bea_principalauthenticator->build_auth_data(user_name,
                                                    client_name,
                                                    system_password,
                                                    user_password,
                                                    NULL,
                                                    auth_data,
                                                    privileges);

    //Determine the security level
    Tobj::AuthType auth_type = var_bea_principal_authenticator->get_auth_type();
    Security::AuthenticationStatus status = var_bea_principal_authenticator->logon(
                                                    user_name,
                                                    client_name,
                                                    system_password,
                                                    user_password,
                                                    0);
}
```



```
if (status != Security::SecAuthSuccess)
{
    //Failed authentication
    return;
}
// Proceed with application
...
// Log off
    try
    {
        logoff();
    }
...

```

Java Code Example of Using the Tobj.PrincipalAuthenticator.logon() Method

Listing 10-6 contains Java code that performs Username/Password authentication using the Tobj.PrincipalAuthenticator.logon() method.

Listing 10-6 Java Client Application That Uses the Tobj.PrincipalAuthenticator.logon() Method

```
...
    // Create bootstrap object
    Tobj_Bootstrap bs =
        new Tobj_Bootstrap(orb, corbaloc://sling.com;2143);

    // Get security current
    org.omg.CORBA.Object secCurObj =
        bs.resolve_initial_references( "SecurityCurrent" );
    org.omg.SecurityLevel2.Current secCur2Obj =
        org.omg.SecurityLevel2.CurrentHelper.narrow(secCurObj);

    // Get Principal Authenticator
    org.omg.Security.PrincipalAuthenticator princAuth =
        secCur2Obj.principal_authenticator();
    com.beasys.Tobj.PrincipalAuthenticator auth =
        Tobj.PrincipalAuthenticatorHelper.narrow(princAuth);

    // Get Authentication type
    com.beasys.Tobj.AuthType authType = auth.get_auth_type();

```

```
// Initialize arguments
String userName = "John";
String clientName = "Teller";
String systemPassword = null;
String userPassword = null;
byte[] userData = new byte[0];

// Prepare arguments according to security level requested
switch(authType.value())
{
    case com.beasys.Tobj.AuthType._TPNOAUTH:
        break;

    case com.beasys.Tobj.AuthType._TPSYSAUTH:
        systemPassword = "sys_pw";
        break;

    case com.beasys.Tobj.AuthType._TPAPPAUTH:
        systemPassword = "sys_pw";
        userPassword = "john_pw";
        break;
}

// Tuxedo-style Authentication
org.omg.Security.AuthenticationStatus status =
    auth.logon(userName, clientName, systemPassword,
        userPassword, userData);

...

// Proceed with application

// Log off
try
{
    auth.logoff();
}

...
```

Using Certificate-based Authentication

This section describes implementing certificate-based authentication in WebLogic Enterprise applications.

The Secure Simpapp Sample Application

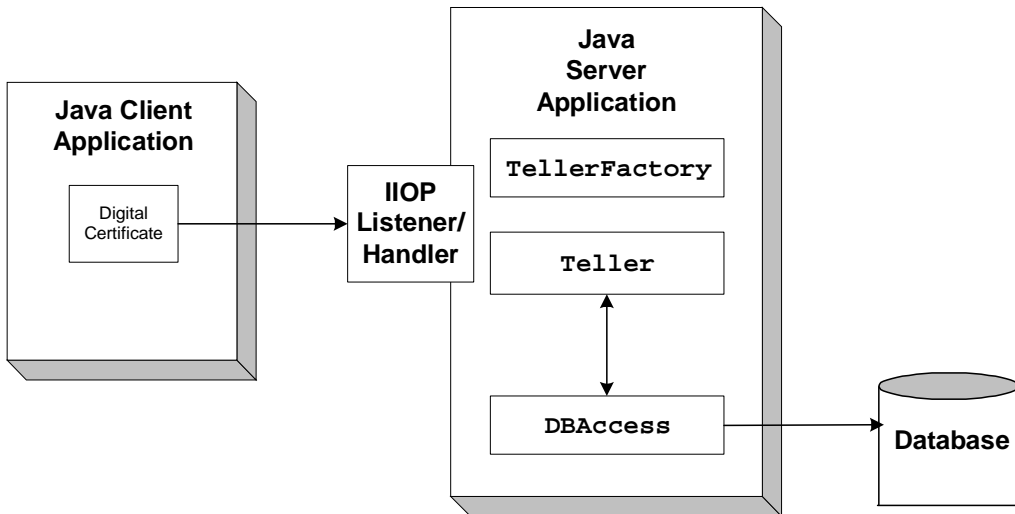
The Secure Simpapp sample application uses the existing Simpapp sample application and modifies the code and configuration files to support secure communications through the SSL protocol and certificate-based authentication.

The server application in the Secure Simpapp sample application provides an implementation of a CORBA object that has the following two methods:

- The `upper` method accepts a string from the client application and converts the string to uppercase letters.
- The `lower` method accepts a string from the client application and converts the string to lowercase letters.

Figure 10-2 illustrates how the Secure Simpapp sample application works.

Figure 10-2 Secure Simpapp Sample Application



The Simpapp sample application was modified in the following ways to support certificate-based authentication and the SSL protocol:

- In the `ISL` section of the `UBBCONFIG` file, the `-a`, `-S`, `-z`, and `-Z` options of the `ISL` command are specified to configure the IIOP Listener/Handler for the SSL protocol.

- In the `ISL` section of the `UBBCONFIG` file, the `SEC_PRINCIPAL_NAME`, the `SEC_PRINCIPAL_LOCATION`, and the `SEC_PRINCIPAL_PASSVAR` parameters are defined to specify proof material for the IOP Listener/Handler.
- The code for the client application uses the `corbalocs` URL address format.
- The code for the client application uses the `authenticate()` method of the `SecurityLevel2:PrincipalAuthenticator` interface to authenticate the principal and obtain credentials for the principals.

The source files for the C++ and Java versions of the Secure Simpapp sample application are located in the `\samples\corba\simpappSSL` and `\samples\corba\simpappSSL_java` directories of the WebLogic Enterprise software. For instructions for building and running the Secure Simpapp sample application, see “Building and Running the CORBA Sample Applications” on page 11-1.

Writing the Client Application

When using certificate-based authentication, write client application code that does the following:

1. Uses the Bootstrap object to obtain a reference to the `SecurityCurrent` object for the specific WebLogic Enterprise domain. Use the `corbalocs` URL address format.
2. Gets the `PrincipalAuthenticator` object from the `SecurityCurrent` object.
3. Uses the `authenticate()` method of the `SecurityLevel2:PrincipalAuthenticator` interface to authenticate the principals and obtain credentials for the principals. When using certificate-based authentication, specify `Tobj::CertificateBased` for the method argument and the pass phrase for the private key as the `auth_data` argument for `Security::Opaque`.

The following sections contain C++ and Java code examples that illustrate implementing certificate-based authentication.

C++ Code Example of Certificate-based Authentication

Listing 10-7 illustrates using certificate-based authentication in a C++ client application.

Listing 10-7 C++ Client Application That Uses Certificate-based Authentication

```
....

// Initialize the ORB
CORBA::ORB_var v_orb = CORBA::ORB_init(argc, argv, "");

// Create the bootstrap object
Tobj_Bootstrap bootstrap(v_orb.in(), corbalocs://sling.com:2143);

// Resolve SecurityCurrent
CORBA::Object_ptr seccurobj =
    bootstrap.resolve_initial_references("SecurityCurrent");
SecurityLevel2::Current_ptr seccur =
    SecurityLevel2::Current::_narrow(seccurobj);

// Perform certificate-based authentication
SecurityLevel2::Credentials_ptr the_creds;
Security::AttributeList_var privileges;
Security::Opaque_var continuation_data;
Security::Opaque_var auth_specific_data;
Security::Opaque_var response_data;

//Principal email address
char emailAddress[] = "milozzi@bigcompany.com;";
// Pass phrase for principal's digital certificate
char password[] = "asdawrew98infldi7;";

// Convert the certificate private key password to opaque
unsigned long password_len = strlen(password);
Security::Opaque ssl_auth_data(password_len);

// Authenticate principal certificate with principal authenticator
for(int i = 0; (unsigned long) i < password_len; i++)
    ssl_auth_data[i] = password[i];
Security::AuthenticationStatus auth_status;
SecurityLevel2::PrincipalAuthenticator_var PA =
    seccur->principal_authenticator();

auth_status = PA->authenticate(Tobj::CertificateBased,
                              emailAddress,
                              ssl_auth_data,
```

```
        privileges,
        the_creds,
        continuation_data,
        auth_specific_data);

    while(auth_status == Security::SecAuthContinue) {
        auth_status = PA->continue_authentication(
            response_data,
            the_creds,
            continuation_data,
            auth_specific_data);
    }

    ...
```

Java Code Example of Certificate-based Authentication

Listing 10-8 illustrates using certificate-based authentication in a C++ client application.

Listing 10-8 Java Client Application That Uses Certificate-based Authentication

```
...

// Initialize the ORB.

Properties Prop;
Prop = new Properties(System.getProperties());
Prop.put("org.omg.CORBA.ORBClass", "com.beasys.CORBA.iiop.ORB");
Prop.put("org.omg.CORBA.ORBSingletonClass",
        "com.beasys.CORBA.idl.ORBSingleton");

ORB orb = ORB.init(args, Prop);

// Create the Bootstrap object

Tobj_Bootstrap bs = new Tobj_Bootstrap(orb,
        corbalocs://foo:2501);

//Resolve SecurityCurrent
org.omg.CORBA.object occur =
    bs.resolve_initial_references("SecurityCurrent");
org.omg.SecurityLevel2.Current curr =
    org.omg.SecurityLevel2.CurrentHelper.narrow(occur);
```

```
// Get Principal Authenticator

com.beasys.Tobj.PrincipalAuthenticator pa =
    (com.beasys.Tobj.PrincipalAuthenticator)
        curr.principal_authenticator();

OpaqueHolder auth_data = new OpaqueHolder();
AttributeListHolder privileges = new AttributeListHolder();
org.omg.SecurityLevel2.CredentialsHolder creds =
    new org.omg.SecurityLevel2.CredentialsHolder();
OpaqueHolder continuation_data = new OpaqueHolder();
OpaqueHolder auth_specific_data = new OpaqueHolder();
auth_data.value=new String ("deathstar").getBytes("UTF8");
if(pa.authenticate(com.beasys.Tobj.CertificateBased.value,
                    "vader@largecompany.com",
                    auth_data.value,
                    privileges.value,
                    the_creds,
                    continuation_data,
                    auth_specific_data)

!AuthenticationStatus.SecAuthSuccess) {
    System.err.println("logon failed");
    System.exit(1);
}
...
```

Using the Invocations_Options_Required() Method

When using certificate-based authentication, it may be necessary for a principal to explicitly define the security attributes it requires. For example, a bank application may have specific security requirements it needs to meet before the bank application can transfer data to a database. The `invocation_options_required()` method of the `SecurityLevel2::Credentials` interface allows the principal to explicitly control the security characteristics of the SSL connection. When using the `corbaloc` URL address format, you can secure the bootstrapping process by using the `authenticate()` and `invocation_options_required()` methods of the `SecurityLevel2::Credentials` interface.

To use the `invocation_options_required()` method, complete the following steps:

1. Write application code that uses the `authenticate()` method of the `SecurityLevel2::PrincipalAuthenticator` object to specify certificate-based authentication is being used.
2. Use the `invocation_options_required()` method to specify the security attributes the principal requires. See the description of the `invocation_options_required()` method in the “C++ Security Reference” on page 16-1 and “Java Security Reference” on page 17-1 for a complete list of security options.

Listing 10-9 provides a C++ example of using the `invocation_options_required()` method.

Listing 10-9 C++ Example of Using the `invocation_options_required()` Method

```
// Initialize the ORB
CORBA::ORB_var v_orb = CORBA::ORB_init(argc, argv, "");

// Create the bootstrap object
Tobj_Bootstrap bootstrap(v_orb.in(), corbalocs://sling.com:2143);

// Resolve SecurityCurrent
CORBA::Object_ptr seccurobj =
    bootstrap.resolve_initial_references("SecurityCurrent");
SecurityLevel2::Current_ptr seccur =
    SecurityLevel2::Current::_narrow(seccurobj);

// Perform certificate-based authentication
SecurityLevel2::Credentials_ptr the_creds;
Security::AttributeList_var privileges;
Security::Opaque_var continuation_data;
Security::Opaque_var auth_specific_data;
Security::Opaque_var response_data;

//Principal email address
char emailAddress[] = "milozzi@bigcompany.com;";
// Pass phrase for principal's digital certificate
char password[] = "asdawrewe98infldi7;";

// Convert the certificate private key password to opaque
unsigned long password_len = strlen(password);
Security::Opaque ssl_auth_data(password_len);
```



```
// Authenticate principal certificate with principal authenticator
for(int i = 0; (unsigned long) i < password_len; i++)
    ssl_auth_data[i] = password[i];
Security::AuthenticationStatus auth_status;
SecurityLevel2::PrincipalAuthenticator_var PA =
    secur->principal_authenticator();

auth_status = PA->authenticate(Tobj::CertificateBased,
                              emailAddress,
                              ssl_auth_data,
                              privileges,
                              the_creds,
                              continuation_data,
                              auth_specific_data);

the_creds->invocation_options_required(
    Security::Integrity|
    Security::DetectReplay|
    Security::DetectMisordering|
    Security::EstablishTrustInTarget|
    Security::EstablishTrustInClient|
    Security::SimpleDelegation);

while(auth_status == Security::SecAuthContinue) {
    auth_status = PA->continue_authentication(
        response_data,
        the_creds,
        continuation_data,
        auth_specific_data);
}

...
```

Listing 10-10 provides a Java example of using the `invocation_options_required()` method

Listing 10-10 Java Example of Using the `invocation_options_required()` Method

```
...

// Initialize the ORB.

Properties Prop;
Prop = new Properties(System.getProperties());
Prop.put("org.omg.CORBA.ORBClass", "com.beasys.CORBA.iiop.ORB");
Prop.put("org.omg.CORBA.ORBSingletonClass",
        "com.beasys.CORBA.idl.ORBSingleton");
```

10 Writing a WebLogic Enterprise CORBA Application that Implements Security

```
ORB orb = ORB.init(args, Prop);

// Create the Bootstrap object

Tobj_Bootstrap bs = new Tobj_Bootstrap(orb,
    corbalocs://foo:2501);

//Resolve SecurityCurrent
org.omg.CORBA.object occur =
    bs.resolve_initial_references("SecurityCurrent");
org.omg.SecurityLevel2.Current curr =
    org.omg.SecurityLevel2.CurrentHelper.narrow(occur);

// Get Principal Authenticator

com.beasys.Tobj.PrincipalAuthenticator pa =
    (com.beasys.Tobj.PrincipalAuthenticator)
        curr.principal_authenticator();

OpaqueHolder auth_data = new OpaqueHolder();
AttributeListHolder privileges = new AttributeListHolder();
org.omg.SecurityLevel2.CredentialsHolder creds =
    new org.omg.SecurityLevel2.CredentialsHolder();
OpaqueHolder continuation_data = new OpaqueHolder();
OpaqueHolder auth_specific_data = new OpaqueHolder();
auth_data.value=new String ("deathstar").getBytes("UTF8");
if(pa.authenticate(com.beasys.Tobj.CertificateBased.value,
    "vader@largecompany.com",
    auth_data.value,
    privileges.value,
    the_creds,
    continuation_data,
    auth_specific_data)
org.omg.SecurityLevel2.Credentials credentials = curr.get_credentials(
    org.omg.Security.CredentialType.SecInvocationCredentials);

credentials.invocation_options_required(
    (short) (org.omg.Security.Integrity.value |
    org.omg.Security.DetectReplay.value|
    org.omg.Security.DetectMisordering.value|
    org.omg.Security.EstablishTrustInTarget.value|
    org.omg.Security.EstablishTrustInClient.value|
    org.omg.Security.SimpleDelegation.value)
);
!AuthenticationStatus.SecAuthSuccess) {
    System.err.println("logon failed");
    System.exit(1);
}
...

```

11 Building and Running the CORBA Sample Applications

The topic contains the following sections:

- Building and Running the Security Sample Application
- Building and Running the Secure Simpapp Sample Application

Building and Running the Security Sample Application

The Security sample application demonstrates using Username/Password authentication. For instructions for building and running the Security sample application, see the [Guide to the University Sample Applications](#).

Building and Running the Secure Simpapp Sample Application

The Secure Simpapp sample application demonstrates using the SSL protocol and certificate-based authentication to protect communications between client applications and the WebLogic Enterprise domain. There are C++ and Java versions of the Secure Simpapp sample application.

To build and run the Secure Simpapp sample application, complete the following steps:

1. Copy the files for the Secure Simpapp sample application into a work directory.
2. Change the protection attribute on the files for the Secure Simpapp sample application.
3. Verify the environment variables.
4. Execute the `runme` command.

Before you can use the Secure Simpapp sample application, obtain a certificate and private key (`IIOPListener.pem`) for the IIOP Listener/Handler from the certificate authority in your enterprise and load the certificate in a Lightweight Directory Access Protocol (LDAP)-enabled directory service. The `runme` command prompts you for the pass phrase for the private key for the IIOP Listener/Handler.

Step 1: Copy the Files for the Secure Simpapp Sample Application into a Work Directory

You need to copy the files for the Secure Simpapp sample application into a work directory on your local machine. The following sections detail the directory location and sources files for the C++ and Java versions of the Secure Simpapp sample application.

C++ Version of the Secure Simpapp Sample Application

The files for the C++ version of the Secure Simpapp sample application are located in the following directories:

Windows NT

`drive:\WLEdir\samples\corba\simpappSSL`

UNIX

`/usr/local/WLEdir/samples/corba/simappSSL`

You will use the files listed in Table 11-1 to build and run the C++ version of the Secure Simpapp sample application.

Table 11-1 Files Included in the C++ Version of the Secure Simpapp Sample Application

File	Description
<code>Simple.idl</code>	The OMG IDL code that declares the <code>Simple</code> and <code>SimpleFactory</code> interfaces. This file is copied from the WLE <code>simpappSSL_java</code> directory by the <code>runme</code> command file.
<code>Simples.cpp</code>	The C++ source code that overrides the default <code>Server::initialize</code> and <code>Server::release</code> methods.
<code>Simplec.cpp</code>	The C++ source code for the client application in the Secure Simpapp sample application.

Table 11-1 Files Included in the C++ Version of the Secure Simpapp Sample Application (Continued)

File	Description
<code>Simple_i.cpp</code>	The C++ source code that implements the <code>Simple</code> and <code>SimpleFactory</code> methods.
<code>Simple_i.h</code>	The C++ header file that defines the implementation of the <code>Simple</code> and <code>SimpleFactory</code> methods.
<code>Readme.html</code>	This file provides the latest information about building and running the C++ version of the Secure Simpapp sample application.
<code>runme.cmd</code>	The Windows NT batch file that builds and runs the C++ version of the Secure Simpapp sample application.
<code>runme.ksh</code>	The UNIX Korn shell script that builds and executes the C++ version of the Secure Simpapp sample application.
<code>makefile.mk</code>	The makefile for the C++ version of the Secure Simpapp sample application on the UNIX operating system. This file is used to manually build the C++ version of the Secure Simpapp sample application. Refer to the <code>Readme.html</code> file for information about manually building the C++ version of the Secure Simpapp sample application. The UNIX <code>make</code> command needs to be in the path of your machine.
<code>makefiles.nt</code>	The makefile for the C++ version of the Secure Simpapp sample application on the Windows NT operating system. This makefile can be used directly by the Visual C++ <code>nmake</code> command. This file is used to manually build the C++ version of the Secure Simpapp sample application. Refer to the <code>Readme.html</code> file for information about manually building the C++ version of the Secure Simpapp sample application. The Windows NT <code>nmake</code> command needs to be in the path of your machine.

Java Version of the Secure Simpapp Sample Application

The files for the Java version of the Secure Simpapp sample application are located in the following directories:

Windows NT

`drive:\WLEdir\samples\corba\simpappSSL_java`

UNIX

`/usr/local/WLEdir/samples/corba/simpappSSL_java`

You will use the files listed in Table 11-2 to build and run the Java Secure Simpapp sample application.

Table 11-2 Files Included in the Java Version of the Secure Simpapp Sample Application

File	Description
<code>Simple.idl</code>	The OMG IDL code that declares the <code>Simple</code> and <code>SimpleFactory</code> interfaces. This file is copied from the WLE <code>simpappSSL_java</code> directory by the <code>runme</code> command file.
<code>ServerImpl.java</code>	The Java source code that overrides the <code>Server.initialize</code> and <code>Server.release</code> methods.
<code>SimpleClient.java</code>	The Java source code for the client application in the Secure Simpapp sample application.
<code>SimpleFactoryImpl.java</code>	The Java source code that implements the <code>SimpleFactory</code> methods.
<code>SimpleImpl.java</code>	The Java source code that implements the <code>Simple</code> methods.
<code>Simple.xml</code>	The Server Description File used to associate activation and transaction policy values with CORBA interfaces. For the Java version of the Secure Simpapp sample application, the <code>Simple</code> and <code>SimpleFactory</code> interfaces have an activation policy of <code>method</code> and a transaction policy of <code>optional</code> .

Table 11-2 Files Included in the Java Version of the Secure Simpapp Sample Application (Continued)

File	Description
<code>Readme.html</code>	The file that provides the latest information about building and running the Java version of the Secure Simpapp sample application.
<code>runme.cmd</code>	The Windows NT batch file that builds and runs the Java version of the Secure Simpapp sample application.
<code>runme.ksh</code>	The UNIX Korn shell script that builds and executes the Java version of the Secure Simpapp sample application.
<code>makefile.mk</code>	The makefile for the Java version of the Secure Simpapp sample application on the UNIX operating system. This file is used to manually build the Secure Simpapp sample application. Refer to the <code>Readme.html</code> file for information about manually building the Secure Simpapp sample application. The UNIX <code>make</code> command needs to be in the path of your machine.
<code>makefiles.nt</code>	The makefile for the Secure Simpapp sample application on the Windows NT operating system. This file is used to manually build the Java version of the Secure Simpapp sample application. Refer to the <code>Readme.html</code> file for information about manually building the Secure Simpapp sample application. The Windows NT <code>nmake</code> command needs to be in the path of your machine.

Step 2: Change the Protection Attribute on the Files for the Secure Simpapp Sample Application

During the installation of the WebLogic Enterprise software, the sample application files are marked read-only. Before you can edit or build the files in the Secure Simpapp sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows NT

```
prompt>attrib -r drive:\workdirectory\*.*
```

UNIX

```
prompt>/bin/ksh
```

```
ksh prompt>chmod u+w /workdirectory/*.*
```

On the UNIX operating system platform, you also need to change the permission of `runme.ksh` to give execute permission to the file, as follows:

```
ksh prompt>chmod +x runme.ksh
```

Step 3: Verify the Settings of the Environment Variables

Before building and running the Secure Simpapp sample application, you need to ensure that certain environment variables are set on your system. In most cases, these environment variables are set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Table 11-3 lists the environment variables required to run the Secure Simpapp sample application.

Table 11-3 Required Environment Variables for the Secure Simpapp Sample Application

Environment Variable	Description
APPDIR	The directory path where you copied the sample application files. For example: Windows NT APPDIR=c:\work\simpappSSL UNIX APPDIR=/usr/work/simpappSSL
TUXCONFIG	The directory path and name of the configuration file. For example: Windows NT TUXCONFIG=c:\work\simpappSSL\tuxconfig UNIX TUXCONFIG=/usr/work/simpappSSL/tuxconfig
JDKDIR	The directory path where you installed the JDK software. For example: Windows NT JDKDIR=c:\jdk1.2.2 UNIX JDKDIR=/usr/local/jdk1.2.1 You need to specify this parameter only if you plan to use the Java version of the Secure Simpapp sample application.
TOBJADDR	The host name and port number of the IIOP Listener/Handler. The port number must be defined as a port for SSL communications. For example: Windows NT TOBJADDR=trixie:1111 UNIX TOBJADDR=trixie:1111

Table 11-3 Required Environment Variables for the Secure Simpapp Sample Application

Environment Variable	Description
JAVA_HOME	<p>The directory path where you installed the JDK software. For example:</p> <p>Windows NT</p> <p>JAVA_HOME=c:\jdk1.2</p> <p>UNIX</p> <p>JAVA_HOME=/usr/local/jdk1.2</p> <p>You need to define this environment variable only when you use the Java version of the Secure Simpapp sample application.</p>
RESULTSDIR or JRESULTSDIR	<p>A subdirectory of APPDIR where files that are created as a result of executing the <code>runme</code> command are stored. For example:</p> <p>Windows NT</p> <p>RESULTSDIR=c:\workdirectory\</p> <p>UNIX</p> <p>RESULTSDIR=/usr/local/workdirectory/</p> <p>When using the Java version of the Secure Simpapp sample application, specify the JRESULTSDIR environment variable.</p>

To verify that the information for the environment variables defined during installation is correct, perform the following steps:

Windows NT

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.
The Control Panel appears.
3. Click the System icon.
The System Properties window appears.
4. Click the Environment tab.
The Environment page appears.
5. Check the settings of the environment variables.

UNIX

```
ksh prompt>printenv TUXDIR
```

```
ksh prompt>printenv JAVA_HOME
```

To change the settings, perform the following steps:

Windows NT

1. On the Environment page in the System Properties window, click the environment variable you want to change or enter the name of the environment variable in the Variable field.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

UNIX

```
ksh prompt>export TUXDIR=directorypath
```

```
ksh prompt>export JAVA_HOME=directorypath
```

Step 4: Execute the runme Command

The `runme` command automates the following steps:

1. Setting the system environment variables
2. Loading the `UBBCONFIG` file
3. Compiling the code for the client application
4. Compiling the code for the server application
5. Starting the server application using the `tmboot` command
6. Starting the client application
7. Stopping the server application using the `tmshutdown` command

Note: You can also run the Secure Simpapp sample application manually. The steps for manually running the Secure Simpapp sample application are described in the `Readme.html` file.

To build and run the Secure Simpapp sample application, enter the `runme` command, as follows:

Windows NT

```
prompt>cd workdirectory
```

```
prompt>runme
```

UNIX

```
ksh prompt>cd workdirectory
```

```
ksh prompt>./runme.ksh
```

The Secure Simpapp sample application runs and prints the following messages:

```
Testing simpapp
  cleaned up
  prepared
  built
  loaded ubb
  booted
  ran
  shutdown
  saved results
PASSED
```

During execution of the `runme` command, you are prompted for a password. Enter the pass phrase of the private key of the IIOP Listener/Handler.

Table 11-4 lists the C++ files in the work directory generated by the `runme` command.

Table 11-4 C++ Files Generated by the runme Command

File	Description
<code>Simple_c.cpp</code>	Generated by the <code>idl</code> command, this file contains the client stubs for the <code>SimpleFactory</code> and <code>Simple</code> interfaces.
<code>Simple_c.h</code>	Generated by the <code>idl</code> command, this file contains the client definitions of the <code>SimpleFactory</code> and <code>Simple</code> interfaces.
<code>Simple_s.cpp</code>	Generated by the <code>idl</code> command, this file contains the server skeletons for the <code>SimpleFactory</code> and <code>Simple</code> interfaces.

Table 11-4 C++ Files Generated by the runme Command (Continued)

File	Description
<code>Simple_s.h</code>	Generated by the <code>idl</code> command, this file contains the server definition for the <code>SimpleFactory</code> and <code>Simple</code> interfaces.
<code>.adm/.keybd</code>	A file that contains the security encryption key database. The subdirectory is created by the <code>tmloadcf</code> command in the <code>runme</code> command.
<code>results</code>	A directory generated by the <code>runme</code> command.

Table 11-5 lists the Java files in the work directory generated by the `runme` command.

Table 11-5 Java Files Generated by the runme Command

File	Description
<code>SimpleFactory.java</code>	Generated by the <code>m3idltojava</code> command for the <code>SimpleFactory</code> interface. The <code>SimpleFactory</code> interface contains the Java version of the OMG IDL interface. It extends <code>org.omg.CORBA.Object</code> .
<code>SimpleFactoryHolder.java</code>	Generated by the <code>m3idltojava</code> command for the <code>SimpleFactory</code> interface. This class holds a public instance member of type <code>SimpleFactory</code> . The class provides operations for <code>out</code> and <code>inout</code> arguments that are included in CORBA, but that do not map exactly to Java.
<code>SimpleFactoryHelper.java</code>	Generated by the <code>m3idltojava</code> command for the <code>SimpleFactory</code> interface. This class provides auxiliary functionality, notably the <code>narrow</code> method.
<code>_SimpleFactoryStub.java</code>	Generated by the <code>m3idltojava</code> command for the <code>SimpleFactory</code> interface. This class is the client stub that implements the <code>SimpleFactory.java</code> interface.

Table 11-5 Java Files Generated by the runme Command (Continued)

File	Description
<code>_SimpleFactoryImplBase.java</code>	Generated by the <code>m3idltojava</code> command for the <code>SimpleFactory</code> interface. This abstract class is the server skeleton. It implements the <code>SimpleFactory.java</code> interface. The user-written server class <code>SimpleFactoryImpl</code> extends <code>_SimpleFactoryImplBase</code> .
<code>Simple.java</code>	Generated by the <code>m3idltojava</code> command for the <code>Simple</code> interface. The <code>Simple</code> interface contains the Java version of the OMG IDL interface. It extends <code>org.omg.CORBA.Object</code> .
<code>SimpleHolder.java</code>	Generated by the <code>m3idltojava</code> command for the <code>Simple</code> interface. This class holds a public instance member of type <code>Simple</code> . The class provides operations for <code>out</code> and <code>inout</code> arguments that CORBA has but that do not match exactly to Java.
<code>SimpleHelper.java</code>	Generated by the <code>m3idltojava</code> command for the <code>Simple</code> interface. This class provides auxiliary functionality, notably the <code>narrow</code> method.
<code>_SimpleStub.java</code>	Generated by the <code>m3idltojava</code> command for the <code>Simple</code> interface. This class is the client stub that implements the <code>Simple.java</code> interface.
<code>_SimpleImplBase.java</code>	Generated by the <code>m3idltojava</code> command for the <code>Simple</code> interface. This abstract class is the server skeleton. It implements the <code>Simple.java</code> interface. The user-written server class <code>SimpleImpl</code> extends <code>_SimpleImplBase</code> .
<code>Simple.ser</code>	The Server Descriptor File generated by the <code>buildjobjsrver</code> command in the <code>runme</code> command.
<code>Simple.jar</code>	The server Java ARchive (JAR) file generated by the <code>buildjavaserver</code> command in the <code>runme</code> command.

Table 11-5 Java Files Generated by the runme Command (Continued)

File	Description
<code>.adm/.keybd</code>	A file that contains the security encryption key database. The subdirectory is created by the <code>tmloadcf</code> command in the <code>runme</code> command.
<code>results</code>	A directory generated by the <code>runme</code> command.

Table 11-6 lists files in the `RESULTS` or `JRESULTS` directory generated by the `runme` command.

Table 11-6 Files in the results Directory Generated by the runme Command

File	Description
<code>input</code>	Contains the input that the <code>runme</code> command provides to the Java client application.
<code>output</code>	Contains the output produced when the <code>runme</code> command executes the Java client application.
<code>expected_output</code>	Contains the output that is expected when the Java client application is executed by the <code>runme</code> command. The data in the <code>output</code> file is compared to the data in the <code>expected_output</code> file to determine whether or not the test passed or failed.
<code>log</code>	Contains the output generated by the <code>runme</code> command. If the <code>runme</code> command fails, check this file for errors.
<code>setenv.cmd</code>	Contains the commands to set the environment variables needed to build and run the Java Secure Simpapp sample application on the Windows NT operating system platform.
<code>setenv.ksh</code>	Contains the commands to set the environment variables needed to build and run the Java Secure Simpapp sample application on the UNIX operating system platform.

Table 11-6 Files in the results Directory Generated by the runme Command

File	Description
<code>stderr</code>	Generated by the <code>tmboot</code> command, which is executed by the <code>runme</code> command. If the <code>-noredirect</code> JavaServer option is specified in the <code>UBBCONFIG</code> file, the <code>System.err.println</code> method sends the output to the <code>stderr</code> file instead of to the <code>ULOG</code> file.
<code>stdout</code>	Generated by the <code>tmboot</code> command, which is executed by the <code>runme</code> command. If the <code>-noredirect</code> JavaServer option is specified in the <code>UBBCONFIG</code> file, the <code>System.out.println</code> method sends the output to the <code>stdout</code> file instead of to the <code>ULOG</code> file.
<code>tmsysevt.dat</code>	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the <code>tmboot</code> command in the <code>runme</code> command.
<code>tuxconfig</code>	A binary version of the <code>UBBCONFIG</code> file.
<code>ubb</code>	The <code>UBBCONFIG</code> file for the Java Secure Simpapp sample application.
<code>ULOG.<date></code>	A log file that contains messages generated by the <code>tmboot</code> command.

Using the Secure Simpapp Sample Application

Run the server application in the Secure Simpapp sample application, as follows:

Windows NT

```
prompt>tmboot
```

UNIX

```
ksh prompt>tmboot
```

11 *Building and Running the CORBA Sample Applications*

Run the client application in the Secure Simpapp sample application, as follows:

Windows NT

```
prompt>java -classpath .;%TUXDIR%\udataobj\java\jdk\m3envobj.jar
-DTOBJADDR=%TOBJADDR% SimpleClient
String?
Hello World
HELLO WORLD
hello world
```

UNIX

```
ksh prompt>java -classpath .:$TUXDIR/udataobj/java/jdk\
/m3envobj.jar -DTOBJADDR=$TOBJADDR SimpleClient
String?
Hello World
HELLO WORLD
hello world
```

Note: The Secure Simpapp sample client application uses the client-only JAR file `m3envobj.jar`. However, you can also use the `m3.jar` file to run the client application.

Before using another sample application, enter the following commands to stop the Secure Simpapp sample application and to remove unnecessary files from the work directory:

Windows NT

```
prompt>tmshutdown -y
prompt>nmake -f makefile.nt clean
```

UNIX

```
ksh prompt>tmshutdown -y
ksh prompt>make -f makefile.mk clean
```

12 Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security

This topic includes the following sections:

- Before You Begin
- How Authentication Works with WebLogic Enterprise EJBs
- Development Steps
 - Step 1: Define Security Roles for the Methods of the WebLogic Enterprise EJB
 - Step 2: Specify Security Roles in the Deployment Descriptor of the EJB
 - Step 3: Define the JNDI Environment Properties
 - Step 4: Establish the InitialContext
 - Step 5: Use Home to Get a WebLogic Enterprise EJB
 - Step 6: Use the getCallerPrincipal Method to Authenticate a WebLogic Enterprise EJB

Before You Begin

This topic describes the BEA implementation of the Security feature. The information in this topic supplements the Sun Microsystems, Inc. Enterprise JavaBeans Specification 1.1.

Note: Before proceeding with the remainder of this topic, you should be familiar with the entire content of Sun's specification, particularly Chapter 15, "Security Management."

This topic describes only the integrating security into WebLogic Enterprise EJBs. For a complete description of developing an EJB using the WebLogic Enterprise product, see [Getting Started](#) in the WebLogic Enterprise online documentation.

Note: An EJB in the WebLogic Enterprise domain that issues a callback to a remote J2EE client application cannot propagate the security context of that client application in the callback.

How Authentication Works with WebLogic Enterprise EJBs

From the perspective of an EJB container, EJBs are nontrusted entities that require authentication. The WebLogic Enterprise product uses a JNDI implementation that runs within the EJB container's trusted environment. Using the `WLEInitialContextFactory` JNDI factory with security environment properties establishes the security context for the WebLogic Enterprise client application. The WebLogic Enterprise client application authenticates itself with the WebLogic Enterprise domain when establishing the JNDI Initial context.

Development Steps

Table 12-1 lists the development steps required to implement security in a WebLogic Enterprise EJB.

Table 12-1 Development Steps for Implementing Security in a WebLogic Enterprise EJB

Step	Description
1	Define security roles for the methods of the WebLogic Enterprise EJB.
2	Specify security roles in the Deployment Descriptor of the EJB.
3	Define the JNDI environment properties.
4	Establish the InitialContext.
5	Use Home to get the WebLogic Enterprise EJB.
6	Use the <code>getCallerPrincipal</code> method to authenticate the WebLogic Enterprise EJB.

Step 1: Define Security Roles for the Methods of the WebLogic Enterprise EJB

During the assembly and deployment of an EJB package, you define security roles and associate roles with methods in the deployment descriptor. Security roles are mapped to groups of users in the WebLogic Enterprise security environment. You can use any of the techniques described in the Security Management chapter of the Enterprise JavaBeans 1.1 Specification to define security roles for the methods of a WebLogic Enterprise EJB.

It is possible that two methods with the same name or name/signature appear in both the bean's home and remote interfaces. To handle this case, the optional `<method-interfaces>` element may further restrict the selection to either Home or Remote interface methods.

In a mandatory access control environment, any method invocation not specifically authorized is denied. Sometimes a method does not have a defined `<method-permission>` element. If the `SECURITY` parameter in the `RESOURCES` section of the `UBBCONFIG` file is set to `MANDATORY_ACL` access on a method without an associated `<method-permission>` element, access is denied. This is the recommended setting for production environments. For all other settings of the `SECURITY` parameter, access to a method without an associated `<method-permission>` element is allowed.

There may be methods that should be available to everyone, even in a mandatory access control environment. The WebLogic Enterprise system defines a special role name `*` which means everyone has access to the method.

Note: `ACL` is the only WebLogic Enterprise security level that gives the expected EJB behavior. The `MANDATORY_ACL` security level rejects user access on a method when no `<method-permission>` element is defined for the specified method. The `USER_AUTH` security level fails requests on methods with a `<method-permission>` element defined because that level of security Other.

Table 12-2 describes different combinations of the WebLogic Enterprise security levels (as defined by the `SECURITY` parameter in the `UBBCONFIG` file) and the definition of `<method-permission>` element on an EJB.

- Yes indicates access to the method is granted.
- No means acces to the method is not granted.
- Check indicates access to the method is granted if a role is defined in the `ejb-jar.xml` file. Otherwise, access to the method is denied.

Table 12-2 Combinations of WebLogic Enterprise Security Levels and EJB `<method-permissions>`

Setting for WebLogic Enterprise Security Level	Setting for <code><method-permission></code> element	Is the method accessible?
None	No	Yes
	Yes	No

Table 12-2 Combinations of WebLogic Enterprise Security Levels and EJB <method-permissions> (Continued)

Setting for WebLogic Enterprise Security Level	Setting for <method-permission> element	Is the method accessible?
APP_PW	No	Yes
	Yes	No
USER_AUTH	No	Yes
	Yes	Yes, if the role name of the method is <code>other</code>
ACL	No	Yes
	Yes	Check
MANDATORY_ACL	No	No
	Yes	Check

Step 2: Specify Security Roles in the Deployment Descriptor of the EJB

You specify security roles for the methods of an EJB in the deployment descriptor of the bean. In the WebLogic Enterprise product, there is a one-to-one association between the security roles defined in the deployment descriptor of the EJB and the groups defined with the `tpgrpadd` commands. Role names may be referenced in deployment descriptors before the corresponding group exists. At run time, if a bean's deployment descriptor references a role that does not have a corresponding group, the role is ignored.

Role names are restricted to any alphanumeric characters, a dash (-), an underscore (_), the at-sign (@), and a period (.). The maximum length of a role name is 30 characters. If the name of a security role does not conform to these limitations, it will not be possible for users to have the defined security role.

Listing 12-1 includes code that defines a security role.

Listing 12-1 Defining a Security Role for a Method in an EJB

```
...
<assembly-descriptor>
  <security-role>
    <description>
      "teller" is a role name
    </description>
  </security-role>

  <method-permission>
    <role-name>teller</role-name>
    <method>
      <ejb-name>Accounting</ejb-name>
      <method-name>withdraw</methodname>
    </method>
    ...
  </method-permission>
  ...
</assembly-descriptor>
...
```

Step 3: Define the JNDI Environment Properties

The following sections describe the JNDI environment properties that must be set to enable either Username/Password or certificate-based authentication.

WLEContext.INITIAL_CONTEXT_FACTORY Property

The class `com.beasys.jndi.WLEInitialContextFactory` is the JNDI Service Provider Interface (SPI). This initial context provides an entry point into the WebLogic Enterprise domain. Set `WLEContext.INITIAL_CONTEXT_FACTORY` to `com.beasys.jndi.WLEInitialContextFactory` to access the WebLogic Enterprise domain.

Listing 12-2 includes code that defines the `WLEContext.INITIAL_CONTEXT_FACTORY` property for the WebLogic Enterprise environment.

Listing 12-2 WLEContext.INITIAL_CONTEXT_FACTORY Property

```

Hashtable env = new Hashtable();
/*
 *Specify the initial context implementation to use.
 *The service provider supplies the factory class.
 */
env.put(WLEContext.INITIAL_CONTEXT_FACTORY,
        "com.beasys.jndi.WLEInitialContextFactory");
...

```

WLEContext.PROVIDER_URL Property

Specifies the entry point into the WebLogic Enterprise domain. The value should reflect the host and port of the IIOP Listener/Handler of the target WebLogic Enterprise domain. Use one of the following URL address formats when specifying the location of the IIOP Listener/Handler:

- `corbaloc://hostname:portnumber`

Indicates that the IIOP/RMI protocol is to be used to communicate with the WebLogic Enterprise domain. This URL address format only supports Username/Password authentication.

- `corbalocs://hostname:portnumber`

Indicates that the SSL protocol is to be used to communicate with the WebLogic Enterprise domain. This URL address format supports both Username/Password and certificate-based authentication.

The host and port combination in the URL must match the ISL parameter in the WebLogic Enterprise application's `UBBCONFIG` file. The format of the host and port combination as well as the capitalization must match. If the addresses do not match, communication with the WebLogic Enterprise domain fails.

Listing 12-3 includes code that defines the `WLEContext.PROVIDER_URL` property for the WebLogic Enterprise environment.

Listing 12-3 WLEContext.PROPERTY_URL Property

```
...  
env.put(WLEContext.PROVIDER_URL,  
        "corbaloc://myhost:1000");  
...
```

A WebLogic Enterprise server application that acts as a client application (referred to as a joint client/server application) must set the `WLEContext.PROPERTY_URL` as an empty or null string. The joint client/server application connects to the current application in which it was booted.

WLEContext.SECURITY_AUTHENTICATION Property

Set this property to indicate the type of authentication to be used. The valid values for this property are as follows:

- `None`—indicates that no authentication is performed
- `Simple`—indicates that Username/Password authentication is performed
- `Strong`—indicates that certificate-based authentication is performed

See Table 12-3 for additional keys that need to be specified to use Username/Password or certificate-based authentication.

Listing 12-4 includes code that defines the `WLEContext.SECURITY_AUTHENTICATION` property for the WebLogic Enterprise environment.

Listing 12-4 WLEContext.SECURITY_AUTHENTICATION Property

```
...  
env.put(WLEContext.SECURITY_AUTHENTICATION, "strong");  
...
```

Table 12-3 WebLogic Enterprise Property Keys for Security

Property Key	Meaning
<code>WLEContext.SECURITY_PRINCIPAL</code>	Specifies the identity of the principal used when authenticating the caller to the WebLogic Enterprise domain.
<code>WLEContext.SECURITY_CREDENTIALS</code>	<p>Specifies the credentials of the principal when authenticating the caller to the WebLogic Enterprise domain.</p> <ul style="list-style-type: none"> ■ For certificate-based authentication enabled via <code>SECURITY_AUTHENTICATION="strong"</code>, it specifies the pass phrase used to access the private key and certificate for the EJB. ■ For password-based authentication enabled via <code>SECURITY_AUTHENTICATION="simple"</code>, it specifies a string that is the user's password or an arbitrary object <code>user_data</code> used by the authentication server (AUTHSVR) to verify the credentials of the EJB.
<code>WLEContext.CLIENT_NAME</code>	Specifies the name of the EJB defined by the <code>-c</code> option of the <code>tpusradd</code> command. For more information, see "Defining Authorized Users" on page 7-3.
<code>WLEContext.SYSTEM_PASSWORD</code>	The system password. Required only when using Username/Password authentication.

Listing 12-5 includes the WebLogic Enterprise keys used to define Username/Password authentication.

Listing 12-5 WebLogic Enterprise Keys for Username/Password Authentication

```

...
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL, "corbalocs://myhost:1000")

env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.beasys.jndi.WLEInitialContextFactory");

//Password-Based Authentication
env.put(WLEContext.SECURITY_PRINCIPAL, "milozzi");
env.put(WLEContext.SYSTEM_CREDENTIALS, "mypassword");
env.put(WLEContext.CLIENT_NAME, "writers");

```

12 Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security

```
env.put(WLEContext.SECURITY_AUTHENTICATION, "simple");
env.put(WLEContext.SYSTEM_PASSWORD, "password");
...
```

Listing 12-6 includes the WebLogic Enterprise keys used to define certificate-based authentication.

Listing 12-6 WebLogic Enterprise Keys for Certificate-based Authentication

```
...
//Certificate-Based Authentication
env.put(WLEContext.SECURITY_AUTHENTICATION, "strong");
env.put(WLEContext.SYSTEM_PASSWORD, "SSL");
env.put(WLEContext.SECURITY_PRINCIPAL, "milozzi");
env.put(WLEContext.SECURITY_CREDENTIALS, "credentials");
...
```

Step 4: Establish the InitialContext

To access a WebLogic Enterprise EJB using JNDI, you establish an InitialContext using the following code:

```
Context ctx = new InitialContext(env);
```

Specifying `env` as `com.beasys.com.jndi.WLEInitialContextFactory`. After the context is created, the client application has access to bean homes in the WebLogic Enterprise domain using WebLogic Enterprise as the name service provider.

A WebLogic Enterprise EJB is implicitly associated with the security context specified when the `WLEContext` object is created. To specify a new security context, the EJB needs to close the current security context and establish a new security context with new security attributes. Use the following code to close the current security context:

```
ctx.close();
```

Step 5: Use Home to Get a WebLogic Enterprise EJB

Client applications use the bean's home interface to create or find beans. The beans's home is obtained by using the `lookup` method on the InitialContext.

Step 6: Use the `getCallerPrincipal` Method to Authenticate a WebLogic Enterprise EJB

Use the `getCallerPrincipal` method on the `javax.ejb.EJBContext` associated with a WebLogic Enterprise EJB to authenticate the principal. You can also use the `isCallerInRole` method to determine the role of the client application invoking methods on the EJB. The default principal is `IIOP Client`.

Limitations and Restrictions

It is possible to deploy the same EJB more than once with different deployment descriptors that set different access control policies. In this case access control is based on the deployment descriptor from which a particular bean is loaded. Security policies are not considered when the WebLogic Enterprise system has a choice of how to route a request to any particular bean or container.

Example of Using Security in a WebLogic Enterprise EJB

Listing 12-7 illustrates using Username/Password authentication in a WebLogic Enterprise EJB.

Note: The code example in Listing 12-7 uses the `corbaloc` URL address format so that the SSL protocol is used to protect the integrity of the communications.

Listing 12-7 Username/Password Authentication in a WebLogic Enterprise EJB

```
static public Context getInitialContext() throws Exception {  
    Hashtable env = new Hashtable ();
```

12 Writing a WebLogic Enterprise Enterprise JavaBean That Implements Security

```
env.put(WLEContext.INITIAL_CONTEXT_FACTORY,
        "com.beasys.jndi.WLEInitialContextFactory");

env.put(WLEContext.PROVIDER_URL, "corbalocs://myhost:7002");

return new InitialContext(env);

//Password-Based Authentication
env.put(WLEContext.SECURITY_AUTHENTICATION, "simple");
env.put(WLEContext.SYSTEM_PASSWORD, "RMI");
env.put(WLEContext.SECURITY_PRINCIPAL, "milozzi");
env.put(WLEContext.CLIENT_NAME, "writers");
env.put(WLEContext.SECURITY_CREDENTIALS, "password");
```

Listing 12-8 illustrates using certificate-based authentication in a WebLogic Enterprise EJB.

Listing 12-8 Certificate-based Authentication in a WebLogic Enterprise EJB

```
...
Hashtable env = new Hashtable ();
env.put(WLEContext.INITIAL_CONTEXT_FACTORY,
        "com.beasys.jndi.WLEInitialContextFactory");

env.put(WLEContext.PROVIDER_URL, "corbalocs://myhost:7002");

return new InitialContext(env);

//Certificate-Based Authentication
env.put(WLEContext.SECURITY_AUTHENTICATION, "strong");
env.put(WLEContext.SECURITY_PRINCIPAL, "milozzi@bigcompany.com");
env.put(WLEContext.CLIENT_NAME, "writers");
env.put(WLEContext.SYSTEM_PASSWORD, "SSL");
env.put(WLEContext.SECURITY_CREDENTIALS, "credentials");
```

13 Troubleshooting

This topic includes the following sections:

- Using ULOGS and ORB Tracing
- CORBA::ORB_init Problems
- Username/Password Authentication Problems
- Certificate-based Authentication Problems
- Tobj::Bootstrap:: resolve_initial_references Problems
- IIOP Listener/Handler Startup Problems
- Configuration Problems
- Problems with Using Callbacks Objects with the SSL Protocol
- Troubleshooting Tips for Digital Certificates

Note: The problems in this topic pertain to using the SSL protocol and certificate-based authentication with WebLogic Enterprise CORBA applications.

Using ULOGS and ORB Tracing

In general, Object Request Brokers (ORBs) write important failures to the `ULOG` file. When using the CORBA C++ ORB, you can also enable ORB internal tracing which may provide information in addition to the information that appears in the `ULOG` file.

When looking at the ULOG file, note that remote ORB processes by default do not write data to the ULOG file in APPDIR.

- On UNIX, the remote ORB writes information to a ULOG file in the current directory.
- On Windows NT, the remote ORB writes information to a ULOG file in the `c:\ulog` directory.

You can set the ULOGPFX environment variable to control the location of the ULOG file for remote ORBs (for example, you can set the location of the ULOG file to APPDIR so that all information is put in the same ULOG file). Set the ULOGPFX environment variable as follows:

Windows NT

```
set ULOGPFX=%APPDIR%\ULOG
```

UNIX

```
setenv ULOGPFX $APPDIR/ULOG
```

To enable ORB tracing, complete the following steps:

1. Create a file named `trace.dat` in APPDIR. The contents of `trace.dat` should have `all=on`.
2. Use the following command to set the `OBB_TRACE_INPUT` environment variable to point to the `trace.dat` file before running the application:

```
set OBB_TRACE_INPUT=%APPDIR%\trace.dat
```

If you want ORB tracing sent to separate files, add the following line to the `trace.dat` file:

```
output=obbtrace%p.log
```

This command sends the trace output to files that are named after each running process. You may want to do this if you are using ORB tracing on UNIX to an NFS mounted drive. In this case, trace performance is slow due to the user log opening, writing, and closing the file for each trace statement.

The CORBA Java ORB logs error messages to the ULOG file in all error situations as well as puts minor codes to all system exceptions thrown by the ORB. Therefore, tracing is not necessary.

CORBA::ORB_init Problems

Note: This section applies to the CORBA C++ ORB only.

The `ORB_init` routine does not perform internal ORB tracing so you will not see any trace output for invalid argument processing. Therefore, you need to double check the arguments that were passed to the `ORB_init` routine.

If a `CORBA::BAD_PARAM` exception occurs when executing the `ORB_init` routine, verify that all required arguments have values. Also, check that arguments which expect a value from a specific set of valid values have the correct value. Note that values for the arguments of the `ORB_init` routine are case sensitive.

If a `CORBA::NO_PERMISSION` exception occurs and an SSL argument was specified to the `ORB_init` routine, make sure the WebLogic Enterprise Security Pack is installed. Also, verify that the specified level of encryption does not exceed the encryption level supported by the WebLogic Enterprise Security Pack.

If a `CORBA::IMP_LIMIT` exception occurs when executing the `ORB_init` routine, verify that the `ORBport` and `ORBSecurePort` system properties have the same value.

If a `CORBA::Initialize` exception occurs when executing the `ORB_init` routine, verify that the values for `OrbId` or `configset` are valid.

Note: The `OrbId` and `configset` values apply to the CORBA C++ ORB only.

If Secure Sockets Layer (SSL) arguments are passed to the `ORB_init` routine, the ORB attempts to load and initialize the SSL protocol. If no SSL arguments are passed, the ORB does not attempt to initialize the SSL protocol.

The ORB is not aware of the new URL address formats for the Bootstrap object so if you specify a `corbaloc` or `corbalocs` URL address format, the ORB does not try to load the SSL protocol during the `ORB_init` routine.

If SSL arguments were specified to the `ORB_init` routine, check the following:

- The specified values for the SSL arguments do not conflict with each other or other ORB arguments.
- Whether or not the ORB is a native process. If the ORB is a native process, SSL arguments are not supported.

- That the value specified for the `maxCrypto` system property is less than the value specified for the `minCrypto` system property.
- Application-controlled SSL configuration parameters that are not correct. The `ORB_init` routine does not perform digital certificate lookups check so look for missing or corrupted files that would case the dynamic libraries not to be loaded. Also, verify the dynamic libraries are loaded. The ORB trace function will provide information about whether or not the dynamic libraries are loaded.

If the problem persists, turn on ORB tracing. ORB tracing will log SSL failures that occur when the `liborbssl` dynamic library is loaded and initialized.

Username/Password Authentication Problems

If the client application fails when using the `corbalocs` URL address format with Username/Password authentication, check the following:

- The proper configuration steps were performed. See “Configuring the SSL Protocol” and “Configuring Authentication” for the list of the required configuration steps.
- An initialization error occurred. Specify a valid SSL system property to the `ORB_init` routine, an error occurs if:
 - The IIOP Listener/Handler is not available. The ORB trace log will show failed connection attempts.
 - The IIOP Listener/Handler is available but it does not support the SSL protocol. The `ULOG` file will show that a non-IIOP message was received.
 - The IIOP Listener/Handler was available and configured for the SSL protocol but the SSL connection could not be established. This error can occur when the range of encryption strengths supported by the IIOP Listener/Handler and the range of encryption strengths required by the client application do not match.

Certificate-based Authentication Problems

If the client application fails when using the `corbalocs` URL address format with certificate-based authentication, check the following:

- The proper configuration steps were performed. See “Configuring the SSL Protocol” on page 6-1 and “Configuring Authentication” on page 7-1 for the list of the required configuration steps.
- Determine whether or not an initialization error occurred.
- Specify a valid SSL system property to the `ORB_init` routine, an error occurs if:
 - The IIOP Listener/Handler is not available. The ORB trace log will show failed connection attempts.
 - The IIOP Listener/Handler is available but it does not support the SSL protocol. The `ULOG` file will show that a non-GIOP message was received.
 - The IIOP Listener/Handler was available and configured for the SSL protocol but the SSL connection could not be established. This error can occur when the range of encryption strengths supported by the IIOP Listener/Handler and the range of encryption strengths required by the client application do not match. The error can also occur when the client application does not trust the certificate chain of the IIOP Listener/Handler or the client application did not receive a certificate from the IIOP Listener/Handler. The error will be written to the `ULOG` file and the error will also show up in the ORB trace output.

If an error does not occur, the problem is in the authentication process and the `ULOG` file will contain one of the following error statements indicating the problem:

- `Couldn't connect to an LDAP server`
- `Couldn't find a filter that matched the client certificate`
- `The client certificate was not found in LDAP`
- `The private key file could not be found`
- `The passphrase used to open the private key is not correct`

- The public key from the client certificate did not match the private key

Additional certificate problems can also occur. See “Tobj::Bootstrap::resolve_initial_references Problems” for more information about the types of certificate errors that can occur.

Note: At this point of the initialization process, the failure is not due to a problem in the IIOP Listener/Handler.

Tobj::Bootstrap:: resolve_initial_references Problems

If a failure occurs when performing a

Tobj::Bootstrap::resolve_initial_references with the corbaloc or corbalocs URL address format, a CORBA::InvalidDomain exception is raised. This exception may mask CORBA::NO_PERMISSION or CORBA::COMM_FAILURE exceptions that are raised internally. Look at the ULOG file and turn on ORB tracing to get more details on the error. The following errors may occur:

- If the IIOP Listener/Handler is not available, the ORB trace log will show failed connection attempts.
- If the IIOP Listener/Handler is available but it does not support the SSL protocol, the ULOG file will show that a non-GIOP message was received.
- If the IIOP Listener/Handler is available and configured for the SSL protocol but the SSL connection could not be established. An error can occur if the range of encryption strengths supported by the IIOP Listener/Handler and required by the client application do not match.
- The IIOP Listener/Handler could not map a certificate to a Username/Password username. Verify that the security level for the WebLogic Enterprise application is set to USER_AUTH and that Username/Password username matches the principal name passed into the authenticate call. Also, check that the username does not exceed the 30 character limit.

Additional certificate problems can occur. See ““Troubleshooting Tips for Digital Certificates” on page 13-9 for more information about the types of certificate errors that can occur.

Note: The Java implementation of the `Tobj_Bootstrap::resolve_initial_references()` method does not throw an `InvalidDomain` exception. When the `corbaloc` or `corbalocs` URL address formats are used, the `Tobj_Bootstrap::resolve_initial_references()` method internally catches the `InvalidDomain` exception and throws the exception as a `COMM_FAILURE`. The method functions this way in order to provide backward compatibility.

IIOP Listener/Handler Startup Problems

This section describes problems that can occur during the startup of the IIOP Listener/Handler.

If a failure occurs when starting the IIOP Listener/Handler, check the `ULOG` file for a description of the error. The IIOP Listener/Handler verifies that the values for the SSL arguments specified in the `CLOPT` parameters are valid. If any of the values are invalid, the appropriate error is recorded in the `ULOG` file. This check is similar to the argument checking done by the ORB.

The IIOP Listener/Handler will not start its processes unless the `-m` option is specified. The ISH is the process that actually loads and initializes the SSL libraries. If there is a problem loading and initializing the SSL libraries in the ISH process, the error will not be recorded in the `ULOG` file until the ISH process starts to handle incoming requests from client application.

If you suspect a problem with the startup of the IIOP Listener/Handler processes, check the `ULOG` file.

Configuration Problems

The following are miscellaneous tips to resolve the common configuration problems which may occur when using the WebLogic Enterprise Security Pack:

- The ORB `-ORBpeerValidate` command-line option or system property and the `-v` option of the ISL command do not control the peer validation rules checking. This system property and option only control the checking of the host name specified in the peer certificate against the host name of the machine to which the principal was connected.
- The only way to disable the peer validation rules on an installed kit is to create an empty file for `%TUXDIR%\udataobj\security\certs\peer_val.rul`. If you are writing a script that builds your WebLogic Enterprise application, you can also not register the `peer_val.rul` file in the script.
- When enabling renegotiation intervals in the IIOP Listener/Handler, check that the option on the ISL command is `-R` not `-r`. If you use an `-r`, the IIOP Listener/Handler will use the SSL protocol but the renegotiation interval will not be used. In addition, the `ULOG` file will note that an unknown option was specified on the IIOP Listener/Handler.

Another way to determine if the IIOP Listener/Handler is performing renegotiations is to enable ORB tracing on the client side and check whether the cipher suite negotiation callback is being called the configured renegotiation interval. Note that the client application must be sending requests for in order for renegotiations to occur.

- If you have defined the `SECURITY` parameter in the WebLogic Enterprise application's `UBBCONFIG` file to be `APP_PW` or greater and you have configured the IIOP Listener/Handler to use the SSL protocol but not mutual authentication, you must use Username/Password authentication with the `corbalocs` URL address format to communicate with the IIOP Listener/Handler. If you try to use certificate-based authentication, the IIOP Listener/Handler will not ask the principal for a certificate when establishing an SSL connection and the IIOP Listener/Handler is not able to map the identity of the principal to a Tuxedo identity.

Problems with Using Callbacks Objects with the SSL Protocol

If you have a joint client/server application and the client portion of the joint client/server application specifies security requirements using either the `corbalocs` URL address format or by requiring credentials, you must use the `-ORBsecurePort` system property with the `ORB_init` routine to specify that a secure port be used.

If you do not specify the `-ORBsecurePort` system property, the server registration will fail with a `CORBA::NO_PERMISSION` exception. To verify this is the problem, enable ORB tracing and look for the following trace output:

```
TCPTransport::Listen: FAILURE: Attempt to listen on clear port  
while Credentials require SSL be used
```

If you want to use the SSL protocol with callback objects, the joint client/server application must use the

`SecurityLevel2::PrincipalAuthenticator::authenticate()` method with certificate-based authentication. Otherwise, the joint client/server application does not have a certificate with which to identify itself to the IIOP Listener/Handler which in this case is the initiator of the SSL connection.

Troubleshooting Tips for Digital Certificates

In general, problems with digital certificates occur when:

- One of the digital certificates in the certificate chain of the IIOP Listener/Handler is not from a certificate authority defined in the `trust_ca.cer` file.
- The name the IIOP Listener/Handler connected to the client application does not match the host name specified in digital certificates of the IIOP Listener/Handler when a host match is performed. The name of the IIOP Listener/Handler is specified in the `CommonName` attribute of the distinguish name of the IIOP

Listener/Handler. The host name and the `CommonName` attribute must match exactly.

You can verify this error by setting the `-ORBpeerValidate` system property to `none` and executing the `ORB_init` routine again.

- One of the digital certificates in the certificate chain of the IIOP Listener/Handler does not match the specified peer validation rules.
- The digital certificate of the IIOP Listener/Handler is invalid. The digital certificate of the IIOP Listener/Handler becomes invalid when the digital certificate is tampered with, it expires, or the certificate authority that issued the digital certificate expires.

If a digital certificate is rejected for no explainable reason, complete the following steps:

1. Open the digital certificate in a viewer, for example, Windows Explorer.
2. Look at the `KeyUsage` and `BasicConstraints` properties of the digital certificate. A small yellow triangle with an exclamation mark indicates the property is critical. Any digital certificate with a property marked critical is rejected by the WebLogic Enterprise software.
3. If the none of the properties of the digital certificate are critical, check the properties of the next digital certificate in the certificate chain. Perform this step until all the properties of all the digital certificates in the certificate chain have been verified.

Part IV Security Reference

Chapter 14. WebLogic Enterprise Security Service APIs

Chapter 15. Security Modules

Chapter 16. C++ Security Reference

Chapter 17. Java Security Reference

Chapter 18. Automation Security Reference

14 WebLogic Enterprise Security Service APIs

This topic includes the following sections:

- The WebLogic Enterprise Security Model
- Functional Components of the WebLogic Enterprise Security Service
- The Principal Authenticator Object
- The Credentials Object
- The SecurityCurrent Object

For the C++, Java, and Automation method descriptions for the WebLogic Enterprise Security Service, see the following topics:

- “C++ Security Reference” on page 16-1
- “Java Security Reference” on page 17-1
- “Automation Security Reference” on page 18-1

The WebLogic Enterprise Security Model

The security model in the WebLogic Enterprise product defines only a framework for security. The WebLogic Enterprise product provides the flexibility to support different security mechanisms and policies that can be used to achieve the appropriate level of functionality and assurance for a particular WebLogic Enterprise application.

The security model in the WebLogic Enterprise product defines:

- Under what conditions client applications may access objects in a WebLogic Enterprise domain
- What type of proof material principals are required to authenticate themselves to the WebLogic Enterprise domain

The security model in the base WebLogic Enterprise product is a combination of the security model defined in the CORBA Services Security Service specification and the value-added extensions that provide a focused, simplified form of the security model found in BEA Tuxedo.

The following sections describe the general characteristics of the WebLogic Enterprise security model.

Authentication of Principals

Authentication of principals (for example, an individual user, a client application, a server application, a joint client/server application, or an IIOP Listener/Handler) provides security officers with the ability to ensure that only registered principals have access to the objects in the system. An authenticated principal is used as the primary mechanism to control access to objects. The act of authenticating principals allows the security mechanisms to:

- Make principals accountable for their actions
- Control access to protected objects
- Identify the originator of a request
- Identify the target of request

Controlling Access to Objects

The WebLogic Enterprise security model provides a simple framework through which a security officer can limit access to the WebLogic Enterprise domain to authorized users only. Limiting access to objects allows security officers to prohibit access to objects by unauthorized principals. The access control framework consists of two parts:

- The object invocation policy that is enforced automatically on object invocation
- An application access policy that the user-written application can enforce

Administrative Control

The system administrator is responsible for setting security policies for the WebLogic Enterprise application. The WebLogic Enterprise product provides a set of configuration parameters and utilities. Using the configuration parameters and utilities, a system administrator can configure the WebLogic Enterprise application to force the principals to be authenticated to access a system on which WebLogic Enterprise software is installed. To enforce the configuration parameters, the system administrator uses the `tmloadcf` command to update the configuration file for a particular WebLogic Enterprise application.

For more information about configuring security for your WebLogic Enterprise application, see “Configuring the SSL Protocol” on page 6-1 and “Configuring Authentication” on page 7-1.

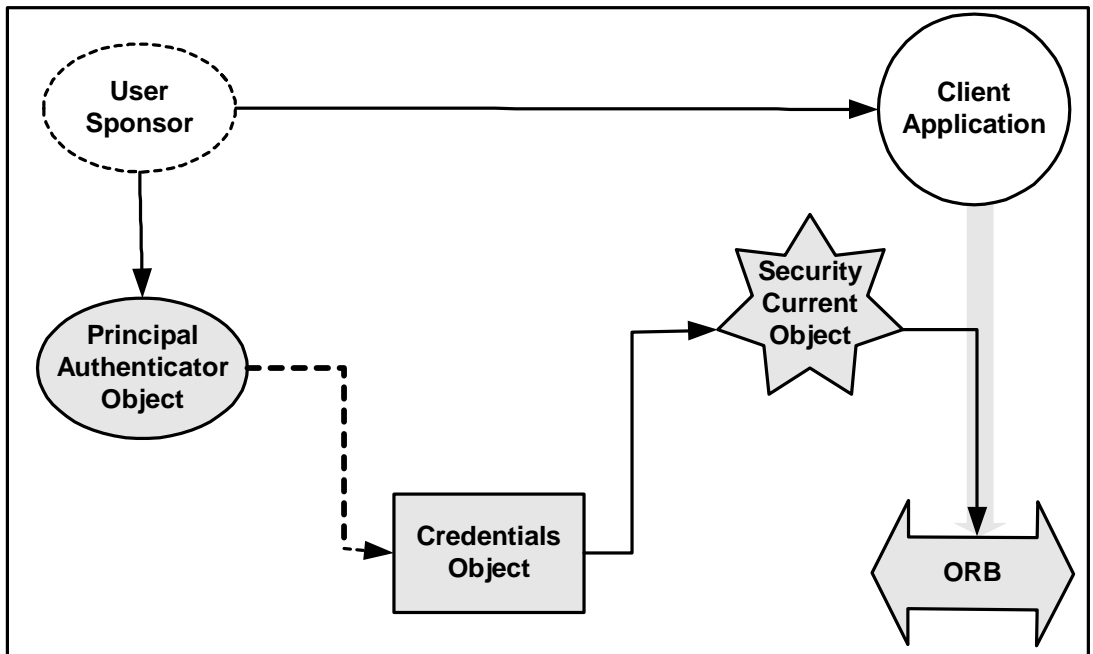
Functional Components of the WebLogic Enterprise Security Service

The WebLogic Enterprise security model is based on the process of authenticating principals to the WebLogic Enterprise domain. The objects of the WebLogic Enterprise Security Service are used to authenticate a principal. The principal provides identity and authentication data, such as a password, to the client application. The

client application uses the Principal Authenticator object to make the calls necessary to authenticate the principal. The credentials for the authenticated principal are associated with the security system's implementation of the SecurityCurrent object and are represented by a Credentials object.

Figure 14-1 illustrates the authentication process used in the WebLogic Enterprise security model.

Figure 14-1 Authentication Process in the WebLogic Enterprise Product



The following sections describe the objects in the WebLogic Enterprise security model.

The Principal Authenticator Object

The Principal Authenticator object is used by a principal that requires authentication but has not been authenticated prior to calling the object system. The act of authenticating a principal results in the creation of a Credentials object that is made available as the default credentials for the application.

The Principal Authenticator object is a singleton object; there is only a single instance allowed in a process address space. The Principal Authenticator object is also stateless. A Credentials object is not associated with the Principal Authenticator object that created it.

All Principal Authenticator objects support the `SecurityLevel2::PrincipalAuthenticator` interface defined in the CORBAServices Security Service specification. This interface contains two methods that are used to accomplish the authentication of the principal. This is because authentication of principals may require more than one step. The `authenticate` method allows the caller to authenticate, and optionally select, attributes for the principal of this session.

Any invocation that fails because the security infrastructure does not permit the invocation will raise the standard exception `CORBA::NO_PERMISSION`. A method that fails because the feature requested is not supported by the security infrastructure implementation will raise the `CORBA::NO_IMPLEMENT` standard exception. Any parameter that has inappropriate values will raise the `CORBA::BAD_PARAM` standard exception. If a timing-related problem occurs, they raise a `CORBA::COMM_FAILURE`. The Bootstrap object maps most system exceptions to `CORBA::Invalid_Domain`.

The Principal Authenticator object is a locality-constrained object. Therefore, a Principal Authenticator object may not be used through the DII/DSI facilities of CORBA. Any attempt to pass a reference to this object outside of the current process, or any attempt to externalize it using `CORBA::ORB::object_to_string`, will result in the raising of the `CORBA::MARSHAL` exception.

Using the Principal Authenticator Object with Certificate-based Authentication

The Principal Authenticator object has been enhanced to support certificate-based authentication. The use of certificate-based authentication is controlled by specifying the `Security::AuthenticationMethod` value of `Tobj::CertificateBased` as a parameter to the `PrincipalAuthenticator::authenticate` operation. When certificate-based authentication is used, the implementation of the `PrincipalAuthenticator::authenticate` operation must retrieve the credentials for the principal by obtaining the private key and digital certificates for the principal and registering them for use with the SSL protocol.

The values of the `security_name` and `auth_data` parameters of the `PrincipalAuthenticator::authenticate` operation are used to open the private key for the principal. If the user does not specify the proper values for both of these parameters, the private key cannot be opened and the user fails to be authenticated. As a result of successfully opening the private key, a chain of digital certificates that represent the local identity of the principal is built. Both the private key and the chain of digital certificates must be registered to be used with the SSL protocol.

WebLogic Enterprise Extensions to the Principal Authenticator Object

The WebLogic Enterprise product extends the Principal Authenticator object to support a security mechanism similar to the security in BEA Tuxedo. The enhanced functionality is provided by defining the `Tobj::PrincipalAuthenticator` interface. This interface contains methods to provide similar capability to that available from BEA Tuxedo through the `tpinit` function. The interface `Tobj::PrincipalAuthenticator` is derived from the CORBA `SecurityLevel2::PrincipalAuthenticator` interface.

The extended Principal Authenticator object adheres to all the same rules as the Principal Authenticator object defined in the CORBA services Security Service specification.

The implementation of the extended Principal Authenticator object requires users to supply a username, client name, and additional authentication data (for example, passwords) used for authentication. Because the information needs to be transmitted over the network to the IIOP Listener/Handler, it is protected to ensure confidentiality. The protection must include encryption of any information provided by the user.

An extended Principal Authenticator object that supports the `Tobj::PrincipalAuthenticator` interface provides the same functionality as if the `SecurityLevel2::PrincipalAuthenticator` interface were used to perform the authentication of the principal. However, unlike the `SecurityLevel2::PrincipalAuthenticator::authenticate` method, the `logon` method defined on the `Tobj::PrincipalAuthenticator` interface does not return a Credentials object.

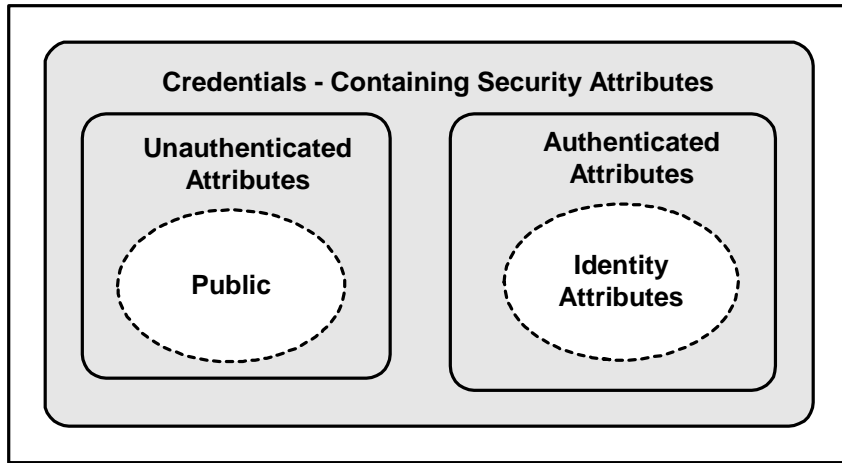
The Credentials Object

A Credentials object (as shown in Figure 14-2) holds the security attributes of a principal. The Credentials object provides methods to obtain and set the security attributes of the principals it represents. These security attributes include its authenticated or unauthenticated identities and privileges. It also contains information for establishing security associations.

Credentials objects are created as the result of:

- Authentication
- Copying an existing Credentials object
- Asking for a Credentials object via the SecurityCurrent object

Figure 14-2 The Credentials Object



Multiple references to a Credentials object are supported. A Credentials object is stateful. It maintains state on behalf of the principal for which it was created. This state includes any information necessary to determine the identity and privileges of the principal it represents. Credentials objects are not associated with the Principal Authenticator object that created it, but must contain some indication of the authentication authority that certified the principal's identity.

The Credentials object is a locality-constrained object; therefore, a Credentials object may not be used through the DII/DSI facilities. Any attempt to pass a reference to this object outside of the current process, or any attempt to externalize it using `CORBA::ORB::object_to_string`, will result in the raising of the `CORBA::MARSHAL` exception.

The Credentials object has been enhanced to allow application developers to indicate the security attributes for establishing secure connections. These attributes allow developers to indicate whether a secure connection requires integrity, confidentiality, or both. To support this capability, two new attributes were added to the `SecurityLevel2::Credentials` interface.

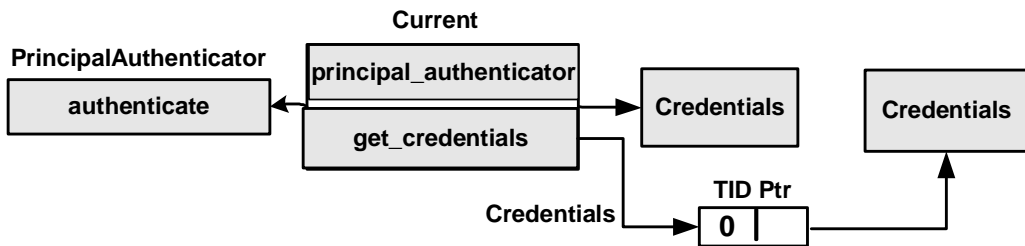
- The `invocation_options_supported` attribute indicates which security options are allowed when establishing a secure connection.

- The `invocation_options_required` attribute allows the application developer to specify the minimum set of security options that must be used in establishing a secure connection.

The SecurityCurrent Object

The SecurityCurrent object (see Figure 14-3) represents the current execution context at both the principal and target objects. The SecurityCurrent object represents service-specific state information associated with the current execution context. Both client and server applications have SecurityCurrent objects that represent state associated with the thread of execution and the process in which the thread is executing.

Figure 14-3 The SecurityCurrent Object



The SecurityCurrent object is a singleton object; there is only a single instance allowed in a process address space. Multiple references to the SecurityCurrent object are supported.

The CORBA services Security Service specification defines two interfaces for the SecurityCurrent object associated with security:

- `SecurityLevel1::Current`, which derives from `CORBA::Current`
- `SecurityLevel2::Current`, which derives from the `SecurityLevel1::Current` interface

Both interfaces give access to security information associated with the execution context.

At any stage, a client application can determine the default credentials for subsequent invocations by calling the `Current::get_credentials` method and asking for the invocation credentials. These default credentials are used in all invocations that use object references.

When the `Current::get_attributes` method is invoked by a client application, the attributes returned from the `Credentials` object are those of the principal.

The `SecurityCurrent` object is a locality-constrained object; therefore, a `SecurityCurrent` object may not be used through the DII/DSI facilities. Any attempt to pass a reference to this object outside of the current process, or any attempt to externalize it using `CORBA::ORB::object_to_string`, results in a `CORBA::MARSHAL` exception.

15 Security Modules

This topic contains the Object Management Group (OMG) Interface Definition Language (IDL) definitions for the following modules that are used in the WebLogic Enterprise Security Service:

- CORBA
- TimeBase
- Security
- Security Level 1
- Security Level 2
- Tobj

CORBA Module

The OMG added the `CORBA::Current` interface to the CORBA module to support the Current pseudo-object. This change enables the CORBA module to support Security Replaceability and Security Level 2.

Listing 15-1 shows the `CORBA::Current` interface OMG IDL statements.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 15-230. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Listing 15-1 CORBA::Current Interface OMG IDL Statements

```
module CORBA {  
    // Extensions to CORBA  
    interface Current {  
    };  
};
```

TimeBase Module

All data structures pertaining to the basic Time Service, Universal Time Object, and Time Interval Object are defined in the TimeBase module. This allows other services to use these data structures without requiring the interface definitions. The interface definitions and associated enums and exceptions are encapsulated in the TimeBase module.

Listing 15-2 shows the TimeBase module OMG IDL statements.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 14-5. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Listing 15-2 TimeBase Module OMG IDL Statements

```
// From time service
module TimeBase {
    // interim definition of type ulonglong pending the
    // adoption of the type extension by all client ORBs.
    struct ulonglong {
        unsigned long    low;
        unsigned long    high;
    };
    typedef ulonglong    TimeT;
    typedef short        TdftT;
    struct UtcT {
        TimeT            time;        // 8 octets
        unsigned long    inacclo;    // 4 octets
        unsigned short    inacchi;    // 2 octets
        TdftT            tdf;        // 2 octets
                                // total 16 octets
    };
};
```

Table 15-1 defines the TimeBase module data types.

Note: This information is taken from *CORBA services: Common Object Services Specification*, p. 14-6. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Table 15-1 TimeBase Module Data Type Definitions

Data Type	Definition
Time ulonglong	OMG IDL does not at present have a native type representing an unsigned 64-bit integer. The adoption of technology submitted against that RFP will provide a means for defining a native type representing unsigned 64-bit integers in OMG IDL. Pending the adoption of that technology, you can use this structure to represent unsigned 64-bit integers, understanding that when a native type becomes available, it may not be interoperable with this declaration on all platforms. This definition is for the interim, and is meant to be removed when the native unsigned 64-bit integer type becomes available in OMG IDL.

Table 15-1 TimeBase Module Data Type Definitions (Continued)

Data Type	Definition
Time TimeT	TimeT represents a single time value, which is 64 bit in size, and holds the number of 100 nanoseconds that have passed since the base time. For absolute time, the base is 15 October 1582 00:00.
Time Tdft	Tdft is of size 16 bits short type and holds the time displacement factor in the form of seconds of displacement from the Greenwich Meridian. Displacements east of the meridian are positive, while those to the west are negative.
Time UtcT	UtcT defines the structure of the time value that is used universally in the service. When the UtcT structure is holding, a relative or absolute time is determined by its history. There is no explicit flag within the object holding that state information. The inacclo and inacchi fields together hold a value of type InaccuracyT packed into 48 bits. The tdf field holds time zone information. Implementation must place the time displacement factor for the local time zone in this field whenever it creates a Universal Time Object (UTO). The content of this structure is intended to be opaque; to be able to marshal it correctly, the types of fields need to be identified.

Security Module

The Security module defines the OMG IDL for security data types common to the other security modules. This module depends on the TimeBase module and must be available with any ORB that claims to be security ready.

Listing 15-3 shows the data types supported by the Security module.

Note: This information is taken from *CORBA services: Common Object Services Specification*, p. 15-193 to 15-195. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Listing 15-3 Security Module OMG IDL Statements

```
module Security {
    typedef sequence<octet>    Opaque;
```

```

// Extensible families for standard data types
struct ExtensibleFamily {
    unsigned short    family_definer;
    unsigned short    family;
};

//security attributes
typedef unsigned long    SecurityAttributeType;

// identity attributes; family = 0
const SecurityAttributeType    AuditId = 1;
const SecurityAttributeType    AccountingId = 2;
const SecurityAttributeType    NonRepudiationId = 3;

// privilege attributes; family = 1
const SecurityAttributeType    Public = 1;
const SecurityAttributeType    AccessId = 2;
const SecurityAttributeType    PrimaryGroupId = 3;
const SecurityAttributeType    GroupId = 4;
const SecurityAttributeType    Role = 5;
const SecurityAttributeType    AttributeSet = 6;
const SecurityAttributeType    Clearance = 7;
const SecurityAttributeType    Capability = 8;

struct AttributeType {
    ExtensibleFamily    attribute_family;
    SecurityAttributeType    attribute_type;
};

typedef sequence <AttributeType>    AttributeTypeLists;
struct SecAttribute {
    AttributeType    attribute_type;
    Opaque    defining_authority;
    Opaque    value;
    // The value of this attribute can be
    // interpreted only with knowledge of type
};

typedef sequence<SecAttribute>    AttributeList;

// Authentication return status
enum AuthenticationStatus {
    SecAuthSuccess,
    SecAuthFailure,
    SecAuthContinue,
    SecAuthExpired
};

```

```
// Authentication method
typedef unsigned long    AuthenticationMethod;

enum CredentialType {
    SecInvocationCredentials;
    SecOwnCredentials;
    SecNRCredentials

// Pick up from TimeBase
typedef TimeBase::UtcT    UtcT;
};
```

Table 15-2 describes the Security module data type.

Table 15-2 Security Module Data Type Definition

Data Type	Definition
sequence<octet>	Data whose representation is known only to the Security Service implementation.

Security Level 1 Module

This section defines those interfaces available to client application objects that use only Level 1 Security functionality. This module depends on the CORBA module and the Security and TimeBase modules. The Current interface is implemented by the ORB.

Listing 15-4 shows the Security Level 1 module OMG IDL statements.

Note: This information is taken from *CORBA services: Common Object Services Specification*, p. 15-198. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Listing 15-4 Security Level 1 Module OMG IDL Statements

```
module SecurityLevel1 {
    interface Current : CORBA::Current { // PIDL
        Security::AttributeList get_attributes(
```

```

        in Security::AttributeTypeList  attributes
    );
};
};

```

Security Level 2 Module

This section defines the additional interfaces available to client application objects that use Level 2 Security functionality. This module depends on the CORBA and Security modules.

Listing 15-5 shows the Security Level 2 module OMG IDL statements.

Note: This information is taken from *CORBA services: Common Object Services Specification*, p. 15-198 to 15-200. Revised Edition: March 31, 1995.
Updated: November 1997. Used with permission by OMG.

Listing 15-5 Security Level 2 Module OMG IDL Statements

```

module SecurityLevel2 {
    // Forward declaration of interfaces
    interface PrincipalAuthenticator;
    interface Credentials;
    interface Current;

    // Interface Principal Authenticator
    interface PrincipalAuthenticator {
        Security::AuthenticationStatus authenticate(
            in Security::AuthenticationMethod  method,
            in string                          security_name,
            in Security::Opaque                auth_data,
            in Security::AttributeTypeList    privileges,
            out Credentials                    creds,
            out Security::Opaque               continuation_data,
            out Security::Opaque               auth_specific_data
        );

        Security::AuthenticationStatus
            continue_authentication(
                in Security::Opaque        response_data,
                inout Credentials           creds,

```

```
        out Security::Opaque      continuation_data,
        out Security::Opaque      auth_specific_data
    );
};

// Interface Credentials
interface Credentials {
    attribute Security::AssociationOptions
        invocation_options_supported;
    attribute Security::AssociationOptions
        invocation_options_required;
    Security::AttributeList get_attributes(
        in Security::AttributeTypeList  attributes
    );
    boolean is_valid(
        out Security::UtcT      expiry_time
    );
};

// Interface Current derived from SecurityLevel1::Current
// providing additional operations on Current at this
// security level. This is implemented by the ORB.
interface Current : SecurityLevel1::Current { // PIDL
    void set_credentials(
        in Security::CredentialType  cred_type,
        in Credentials                cred
    );

    Credentials get_credentials(
        in Security::CredentialType  cred_type
    );
    readonly attribute PrincipalAuthenticator
        principal_authenticator;
};
};
```

Tobj Module

This section defines the Tobj module interfaces.

This module provides the interfaces you use to program the BEA Tuxedo style of authentication.

Listing 15-6 shows the Tobj module OMG IDL statements.

Listing 15-6 Tobj Module OMG IDL Statements

```
//Tobj Specific definitions

//get_auth_type () return values
enum AuthType {
    TOBJ_NOAUTH,
    TOBJ_SYSAUTH,
    TOBJ_APPAUTH
};

typedef sequence<octet>    UserAuthData;

interface PrincipalAuthenticator :
    SecurityLevel2::PrincipalAuthenticator { // PIDL
    AuthType get_auth_type();

    Security::AuthenticationStatus logon(
        in string          user_name,
        in string          client_name,
        in string          system_password,
        in string          user_password,
        in UserAuthData    user_data
    );
    void logoff();

    void build_auth_data(
        in string          user_name,
        in string          client_name,
        in string          system_password,
        in string          user_password,
        in UserAuthData    user_data,
        out Security::Opaque auth_data,
        out Security::AttributeList privileges
    );
};
```

16 C++ Security Reference

This topic contains the C++ method descriptions for the WebLogic Enterprise Security Service.

SecurityLevel1::Current::get_attributes

Synopsis	Returns attributes for the Current interface.
OMG IDL Definition	<pre>Security::AttributeList get_attributes(in Security::AttributeTypeList attributes); };</pre>
Argument	<p>attributes</p> <p>The set of security attributes (privilege attribute types) whose values are desired. If this list is empty, all attributes are returned.</p>
Description	This method gets privilege (and other) attributes from the principal's credentials for the Current interface.
Return Values	The following table describes valid return values.

Return Value	Meaning
Security::Public	Empty (Public is returned when no authentication was performed)
Security::AccessId	Null terminated ASCII string containing the WebLogic Enterprise username
Security::PrimaryGroupId	Null terminated ASCII string containing the WebLogic Enterprise name of the principal

- Note:** The `defining_authority` field is always empty. Depending on the security level defined in the `UBBCONFIG` file not all the values for the `get_attribute` method may be available. Two additional values, `Group Id` and `Role`, are available with the security level is set to `ACL` or `MANDATORY_ACL` in the `UBBCONFIG` file.
- Note:** This information is taken from *CORBA services: Common Object Services Specification*, pp. 15-103, 104. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

SecurityLevel2::Current::authenticate

Synopsis	Authenticates the principal and optionally obtains credentials for the principal.
OMG IDL Definition	<pre>Security::AuthenticationStatus authenticate(in Security::AuthenticationMethod method, in Security::SecurityName security_name, in Security::Opaque auth_data, in Security::AttributeList privileges, out Credentials creds, out Security::Opaque continuation_data, out Security::Opaque auth_specific_data);</pre>
Arguments	<div>method</div> <p>The security mechanism to be used. Valid values are <code>Tobj::TuxedoSecurity</code> and <code>Tobj::CertificateBased</code>.</p> <div>security_name</div> <p>The principal's identification information (for example, logon information). The value must be a pointer to a NULL-terminated string containing the username of the principal. The string is limited to 30 characters, excluding the NULL character.</p> <p>When using certificate-based authentication, this name is used to look up a certificate in the LDAP-enabled directory service. It is also used as the basis for the name of the file in which the private key is stored. For example: <code>milozzi@company.com</code> is the e-mail address used to look up a certificate in the LDAP-enabled directory service and <code>milozzi_company.pem</code> is the name of the private key file.</p> <div>auth_data</div> <p>The principals' authentication, such as their password or private key. If the <code>Tobj::TuxedoSecurity</code> security mechanism is specified, the value of this argument is dependent on the configured level of authentication. If the <code>Tobj::CertificateBased</code> argument is specified, the value of this argument is the pass phrase used to decrypt the private key of the principal.</p> <div>privileges</div> <p>The privilege attributes requested.</p> <div>creds</div> <p>The object reference of the newly created <code>Credentials</code> object. The object reference is not fully initialized; therefore, the object reference cannot be used until the return value of the <code>SecurityLevel2::Current::authenticate</code> method is <code>SecAuthSuccess</code>.</p>

`continuation_data`
If the return value of the `SecurityLevel2::Current::authenticate` method is `SecAuthContinue`, this argument contains the challenge information for the authentication to continue. The value returned will always be empty.

`auth_specific_data`
Information specific to the authentication service being used. The value returned will always be empty.

Description The `SecurityLevel2::Current::authenticate` method is used by the client application to authenticate the principal and optionally request privilege attributes that the principal requires during its session with the WebLogic Enterprise domain.

If the `Tobj::TuxedoSecurity` security mechanism is to be specified, the same functionality can be obtained by calling the `Tobj::PrincipalAuthenticator::logon` operation, which provides the same functionality but is specifically tailored for use with the Tuxedo-style authentication security mechanism.

Return Values The following table describes the valid return values.

Return Value	Meaning
<code>SecAuthSuccess</code>	The object reference of the newly created <code>Credentials</code> object returned as the value of the <code>creds</code> argument is initialized and ready to use.
<code>SecAuthFailure</code>	<p>The authentication process was inconsistent or an error occurred during the process. Therefore, the <code>creds</code> argument does not contain an object reference to a <code>Credentials</code> object.</p> <p>If the <code>Tobj::TuxedoSecurity</code> security mechanism is used, this return value indicates that authentication failed or that the client application was already authenticated and did not call either the <code>Tobj::PrincipalAuthenticator::logoff</code> or <code>Tobj_Bootstrap::destroy_current</code> operation.</p>
<code>SecAuthContinue</code>	Indicates that the authentication procedure uses a challenge/response mechanism. The <code>creds</code> argument contains the object reference of a partially initialized <code>Credentials</code> object. The <code>continuation_data</code> indicates the details of the challenge.

Return Value	Meaning
<code>SecAuthExpired</code>	<p>Indicates that the authentication data contained some information, the validity of which had expired; therefore, the <code>creds</code> argument does not contain an object reference to a <code>Credentials</code> object.</p> <p>If the <code>Tobj::TuxedoSecurity</code> security mechanism is used, this return value is never returned.</p>
<code>CORBA::BAD_PARAM</code>	<p>The <code>CORBA::BAD_PARAM</code> exception occurs if:</p> <ul style="list-style-type: none">■ Values for the <code>security_name</code>, <code>auth_data</code>, or <code>privileges</code> arguments are not specified.■ The length of an input argument exceeds the maximum length of the argument.■ The value of the method argument is <code>Tobj::TuxedoSecurity</code> and the content of the <code>auth_data</code> argument contains a <code>username</code> or a <code>clientname</code> as an empty or a <code>NULL</code> string.

SecurityLevel2::Current::set_credentials

Synopsis Sets credentials type.

OMG IDL Definition

```
void set_credentials(  
    in Security::CredentialType cred_type,  
    in Credentials creds  
);
```

Arguments

`cred_type`
The type of credentials to be set; that is, invocation, own, or non-repudiation.

`creds`
The object reference to the Credentials object, which is to become the default.

Description This method can be used only to set `SecInvocationCredentials`; otherwise, `set_credentials` raises `CORBA::BAD_PARAM`. The credentials must have been obtained from a previous call to `SecurityLevel2::Current::get_credentials` or `SecurityLevel2::PrincipalAuthenticator::authenticate`.

Return Values None.

Note: This information is taken from *CORBA services: Common Object Services Specification*, p. 15-104. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

SecurityLevel2::Current::get_credentials

Synopsis Gets credentials type.

OMG IDL Definition

```
Credentials get_credentials(  
    in Security::CredentialType cred_type  
);
```

Argument `cred_type`
The type of credentials to get.

Description This call can be used only to get `SecInvocationCredentials`; otherwise, `get_credentials` raises `CORBA::BAD_PARAM`. If no credentials are available, `get_credentials` raises `CORBA::BAD_INV_ORDER`.

Return Values Returns the active credentials in the client application only.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 15-105. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

SecurityLevel2::Current::principal_authenticator

Synopsis Returns the `PrincipalAuthenticator`.

OMG IDL Definition `readonly attribute PrincipalAuthenticator
principal_authenticator;`

Description The `PrincipalAuthenticator` returned by the `principal_authenticator` attribute is of actual type `Tobj::PrincipalAuthenticator`. Therefore, it can be used both as a `Tobj::PrincipalAuthenticator` and as a `SecurityLevel2::PrincipalAuthenticator`.

Note: This method raises `CORBA::BAD_INV_ORDER` if it is called on an invalid `SecurityCurrent` object.

Return Values Returns the `PrincipalAuthenticator`.

SecurityLevel2::Credentials

Synopsis Represents a particular principal's credential information that is specific to a process. A Credentials object that supports the SecurityLevel2::Credentials interface is a locality-constrained object. Any attempt to pass a reference to the object outside its locality, or any attempt to externalize the object using the CORBA::ORB::object_to_string() operation, results in a CORBA::Marshall exception.

OMG IDL Definition

```
#ifndef _SECURITY_LEVEL_2_IDL
#define _SECURITY_LEVEL_2_IDL

#include <SecurityLevel1.idl>

#pragma prefix "omg.org"

module SecurityLevel2
{
    interface Credentials
    {
        attribute Security::AssociationOptions
                                invocation_options_supported;
        attribute Security::AssociationOptions
                                invocation_options_required;
        Security::AttributeList
        get_attributes(
            in Security::AttributeTypeList    attributes );

        boolean
        is_valid(
            out Security::UtcT                expiry_time );
    };
};
#endif /* _SECURITY_LEVEL_2_IDL */
```

C++ Declaration

```
class SecurityLevel2
{
public:
    class Credentials;
    typedef Credentials *Credentials_ptr;

    class Credentials : public virtual CORBA::Object
    {
public:
```

```
static Credentials_ptr _duplicate(Credentials_ptr obj);
static Credentials_ptr _narrow(CORBA::Object_ptr obj);
static Credentials_ptr _nil();

virtual Security::AssociationOptions
    invocation_options_supported() = 0;
virtual void
    invocation_options_supported(
        const Security::AssociationOptions options ) = 0;
virtual Security::AssociationOptions
    invocation_options_required() = 0;
virtual void
    invocation_options_required(
        const Security::AssociationOptions options ) = 0;

virtual Security::AttributeList *
    get_attributes(
        const Security::AttributeTypeList & attributes) = 0;

virtual CORBA::Boolean
    is_valid( Security::UtcT_out expiry_time) = 0;

protected:
    Credentials(CORBA::Object_ptr obj = 0);
    virtual ~Credentials() { }

private:
    Credentials( const Credentials&) { }
    void operator=(const Credentials&) { }
}; // class Credentials
}; // class SecurityLevel2
```

SecurityLevel2::Credentials::get_attributes

Synopsis Gets the attribute list attached to the credentials.

OMG IDL Definition

```
Security::AttributeList get_attributes(  
    in AttributeTypeList attributes  
);
```

Argument `attributes`
The set of security attributes (privilege attribute types) whose values are desired. If this list is empty, all attributes are returned.

Description This method returns the attribute list attached to the credentials of the principal. In the list of attribute types, you are required to include only the type value(s) for the attributes you want returned in the `AttributeList`. Attributes are not currently returned based on attribute family or identities. In most cases, this is the same result you would get if you called `SecurityLevel1::Current::get_attributes()`, since there is only one valid set of credentials in the principal at any instance in time. The results could be different if the credentials are not currently in use.

Return Values Returns attribute list.

Note: This is information taken from *CORBA services: Common Object Services Specification*, p. 15-97. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

SecurityLevel2::Credentials::invocation_options_supported

Synopsis Indicates the maximum number of security options that can be used when establishing an SSL connection to make an invocation on an object in the WebLogic Enterprise domain.

OMG IDL Definition

```
attribute Security::AssociationOptions(  
                                invocation_options_supported;
```

Argument None.

Description This method should be used in conjunction with the `SecurityLevel2::Credentials::invocation_options_required` method.

The following security options can be specified:

Security Option	Description
NoProtection	The SSL protocol does not provide message protection.
Integrity	The SSL protocol provides an integrity check of messages. Digital signatures are used to protect the integrity of messages.
Confidentiality	The SSL connection protects the confidentiality of messages. Cryptography is used to protect the confidentiality of messages.
DetectReplay	The SSL protocol provides replay detection. Replay occurs when a message is sent repeatedly with no detection.
DetectMisordering	The SSL protocol provides sequence error detection for requests and request fragments.
EstablishTrustInTarget	Indicates that the target of a request authenticates itself to the initiating principal.
NoDelegation	Indicates that the principal permits an intermediate object to use its privileges for the purpose of access control decisions. However, the principal's privileges are not delegated so the intermediate object cannot use the privileges when invoking the next object in the chain.

Security Option	Description
SimpleDelegation	Indicates that the principal permits an intermediate object to use its privileges for the purpose of access control decisions, and delegates the privileges to the intermediate object. The target object receives only the privileges of the client application and does not know the identity of the intermediate object. When this invocation option is used without restrictions on the target object, the behavior is known as impersonation.
CompositeDelegation	Indicates that the principal permits the intermediate object to use its credentials and delegate them. The privileges of both the principal and the intermediate object can be checked.

Return Values The list of defined security options.

If the `Tobj::TuxedoSecurity` security mechanism is used to create the security association, only the `NoProtection`, `EstablishTrustInClient`, and `SimpleDelegation` security options are returned. The `EstablishTrustInClient` security option appears only if the security level of the WebLogic Enterprise application is defined to require passwords to access the WebLogic Enterprise domain.

Note: A `CORBA::BAD_PARAM` exception is returned if the security options specified are not supported by the security mechanism defined for the WebLogic Enterprise application. This exception can also occur if the security options specified have less capabilities than the security options specified by the `SecurityLevel2::Credentials::invocation_options_required` method.

A `Credentials` object with a security mechanism of `Tobj::TuxedoSecurity` always returns the `CORBA::BAD_PARAM` exception.

SecurityLevel2::Credentials::invocation_options_required

- Synopsis** Specifies the minimum number of security options to be used when establishing an SSL connection to make an invocation on a target object in the WebLogic Enterprise domain.
- OMG IDL Definition**

```
attribute Security::AssociationOptions(  
                                invocation_options_required;
```
- Argument** None.
- Description** Use this method to specify that communication between principals and the WebLogic Enterprise domain should be protected. After using this method, a Credentials object makes an invocation on a target object using the SSL protocol with the defined level of security options. This method should be used in conjunction with the `SecurityLevel2::Credentials::invocation_options_supported` method.

The following security options can be specified:

Security Option	Description
NoProtection	The SSL protocol does not provide message protection.
Integrity	The SSL protocol provides an integrity check of messages. Digital signatures are used to protect the integrity of messages.
Confidentiality	The SSL connection protects the confidentiality of messages. Cryptography is used to protect the confidentiality of messages.
DetectReplay	The SSL protocol provides replay detection. Replay occurs when a message is sent repeatedly with no detection.
DetectMisordering	The SSL protocol provides sequence error detection for requests and request fragments.
EstablishTrustInTarget	Indicates that the target of a request authenticates itself to the initiating principal.
NoDelegation	Indicates that the principal permits an intermediate object to use its privileges for the purpose of access control decisions. However, the principal's privileges are not delegated so the intermediate object cannot use the privileges when invoking the next object in the chain.

Security Option	Description
SimpleDelegation	Indicates that the principal permits an intermediate object to use its privileges for the purpose of access control decisions, and delegates the privileges to the intermediate object. The target object receives only the privileges of the client application and does not know the identity of the intermediate object. When this invocation option is used without restrictions on the target object, the behavior is known as impersonation.
CompositeDelegation	Indicates that the principal permits the intermediate object to use its credentials and delegate them. The privileges of both the principal and the intermediate object can be checked.

Return Values The list of defined security options.

If the `Tobj::TuxedoSecurity` security mechanism is used to create the security association, only the `NoProtection`, `EstablishTrustInClient`, and `SimpleDelegation` security options are returned. The `EstablishTrustInClient` security option appears only if the security level of the WebLogic Enterprise application is defined to require passwords to access the WebLogic Enterprise domain.

Note: A `CORBA::BAD_PARAM` exception is returned if the security options specified are not supported by the security mechanism defined for the WebLogic Enterprise application. This exception can also occur if the security options specified have more capabilities than the security options specified by the `SecurityLevel2::Credentials::invocation_options_supported` method.

A `Credentials` object with a parameter of `Tobj::TuxedoSecurity` always returns the `CORBA::BAD_PARAM` exception.

SecurityLevel2::Credentials::is_valid

Synopsis Checks status of credentials.

OMG IDL Definition

```
boolean is_valid(  
    out Security::UtcT      expiry_time  
);
```

Description This method returns TRUE if the credentials used are active at the time; that is, you did not call `Tobj::PrincipalAuthenticator::logoff` or `Tobj_Bootstrap::destroy_current`. If this method is called after `Tobj::PrincipalAuthenticator::logoff()`, FALSE is returned. If this method is called after `Tobj_Bootstrap::destroy_current()`, the `CORBA::BAD_INV_ORDER` exception is raised.

Return Values The expiration date returned contains the maximum unsigned long long value in C++ and maximum long in Java. Until the unsigned long long datatype is adopted, the ulonglong datatype is substituted. The ulonglong datatype is defined as follows:

```
// interim definition of type ulonglong pending the  
// adoption of the type extension by all client ORBs.  
struct ulonglong {  
    unsigned long    low;  
    unsigned long    high;  
};
```

Note: This information is taken from *CORBA services: Common Object Services Specification*, p. 15-97. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

SecurityLevel2::PrincipalAuthenticator

Synopsis Allows a principal to be authenticated. A Principal Authenticator object that supports the `SecurityLevel2::PrincipalAuthenticator` interface is a locality-constrained object. Any attempt to pass a reference to the object outside its locality, or any attempt to externalize the object using the `CORBA::ORB::object_to_string()` operation, results in a `CORBA::Marshal` exception.

OMG IDL Definition

```
#ifndef _SECURITY_LEVEL_2_IDL
#define _SECURITY_LEVEL_2_IDL

#include <SecurityLevel1.idl>

#pragma prefix "omg.org"

module SecurityLevel2
{
    interface PrincipalAuthenticator
    {
        // Locality Constrained
        Security::AuthenticationStatus authenticate (
            in Security::AuthenticationMethod method,
            in Security::SecurityName security_name,
            in Security::Opaque auth_data,
            in Security::AttributeList privileges,
            out Credentials creds,
            out Security::Opaque continuation_data,
            out Security::Opaque auth_specific_data
        );

        Security::AuthenticationStatus continue_authentication (
            in Security::Opaque response_data,
            in Credentials creds,
            out Security::Opaque continuation_data,
            out Security::Opaque auth_specific_data
        );
    };
};

#endif // SECURITY_LEVEL_2_IDL

#pragma prefix "beasys.com"
module Tobj
{
    const Security::AuthenticationMethod
        TuxedoSecurity = 0x54555800;
```

```
        CertificateBased = 0x43455254;
    };

C++ Declaration  class SecurityLevel2
                  {
                  public:
                    class PrincipalAuthenticator;
                    typedef PrincipalAuthenticator * PrincipalAuthenticator_ptr;

                    class PrincipalAuthenticator : public virtual CORBA::Object
                    {
                    public:
                        static PrincipalAuthenticator_ptr
                            _duplicate(PrincipalAuthenticator_ptr obj);
                        static PrincipalAuthenticator_ptr
                            _narrow(CORBA::Object_ptr obj);
                        static PrincipalAuthenticator_ptr _nil();

                        virtual Security::AuthenticationStatus
                            authenticate (
                                Security::AuthenticationMethod method,
                                const char * security_name,
                                const Security::Opaque & auth_data,
                                const Security::AttributeList & privileges,
                                Credentials_out creds,
                                Security::Opaque_out continuation_data,
                                Security::Opaque_out auth_specific_data) = 0;

                        virtual Security::AuthenticationStatus
                            continue_authentication (
                                const Security::Opaque & response_data,
                                Credentials_ptr & creds,
                                Security::Opaque_out continuation_data,
                                Security::Opaque_out auth_specific_data) = 0;

                    protected:
                        PrincipalAuthenticator(CORBA::Object_ptr obj = 0);
                        virtual ~PrincipalAuthenticator() { }

                    private:
                        PrincipalAuthenticator( const PrincipalAuthenticator&) { }
                        void operator=(const PrincipalAuthenticator&) { }
                    }; // class PrincipalAuthenticator
    };
```

SecurityLevel2::PrincipalAuthenticator::continue_authentication

Synopsis Always fails.

OMG IDL Definition

```
Security::AuthenticationStatus continue_authentication(  
    in Security::Opaque                response_data,  
    inout Credentials                  creds,  
    out Security::Opaque                continuation_data,  
    out Security::Opaque                auth_specific_data  
);
```

Description Because the WebLogic Enterprise software does authentication in one step, this method always fails and returns
`Security::AuthenticationStatus::SecAuthFailure`.

Return Values Always returns `Security::AuthenticationStatus::SecAuthFailure`.

Note: This information is taken from *CORBA services: Common Object Services Specification*, pp. 15-92, 93. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Tobj::PrincipalAuthenticator::get_auth_type

Synopsis Gets the type of authentication expected by the WebLogic Enterprise domain.

OMG IDL `AuthType get_auth_type();`
Definition

Description This method returns the type of authentication expected by the WebLogic Enterprise domain.

Note: This method raises CORBA: :BAD_INV_ORDER if it is called with an invalid SecurityCurrent object.

Return Values A reference to the Tobj_AuthType enumeration. Returns the type of authentication required to access the WebLogic Enterprise domain. The following table describes the valid return values.

Return Value	Meaning
TOBJ_NOAUTH	No authentication is needed; however, the client application can still authenticate itself by specifying a username and a client application name. No password is required. To specify this level of security, specify the NONE value for the SECURITY parameter in the RESOURCES section of the UBBCONFIG file.
TOBJ_SYSAUTH	The client application must authenticate itself to the WebLogic Enterprise domain, and must specify a username, a name, and a password for the client application. To specify this level of security, specify the APP_PW value for the SECURITY parameter in the RESOURCES section of the UBBCONFIG file.
TOBJ_APPAUTH	The client application must provide proof material that authenticates the client application to the WebLogic Enterprise domain.The proof material may be a password or a digital certificate. To specify this level of security, specify the USER_AUTH value for the SECURITY parameter in the RESOURCES section of the UBBCONFIG file.

Tobj::PrincipalAuthenticator::logon

Synopsis Authenticates the principal.

OMG IDL Definition

```
Security::AuthenticationStatus logon(  
    in string          user_name,  
    in string          client_name,  
    in string          system_password,  
    in string          user_password,  
    in UserAuthData    user_data  
);
```

Arguments user_name

The WebLogic Enterprise username. The authentication level is TOBJ_NOAUTH. If user_name is NULL or empty, or exceeds 30 characters, logon raises CORBA::BAD_PARAM.

client_name

The WebLogic Enterprise name of the client application. The authentication level is TOBJ_NOAUTH. If the client_name is NULL or empty, or exceeds 30 characters, logon raises the CORBA::BAD_PARAM exception.

system_password

The WebLogic Enterprise client application password. The authentication level is TOBJ_SYSAUTH. If the client name is NULL or empty, or exceeds 30 characters, logon raises the CORBA::BAD_PARAM exception.

Note: The system_password must not exceed 30 characters.

user_password

The user password (needed for use by the default WebLogic Enterprise authentication service). The authentication level is TOBJ_APPAUTH.

user_data

Data that is specific to the client application (needed for use by a custom WebLogic Enterprise authentication service). The authentication level is TOBJ_APPAUTH.

Note: TOBJ_SYSAUTH includes the requirements of TOBJ_NOAUTH, plus a client application password. TOBJ_APPAUTH includes the requirements of TOBJ_SYSAUTH, plus additional information, such as a user password or user data.

Note: The user_password and user_data arguments are mutually exclusive, depending on the requirements of the authentication service used in the configuration of the WebLogic Enterprise domain. The WebLogic

Enterprise default authentication service expects a user password. A customized authentication service may require user data. The logon call raises the `CORBA::BAD_PARAM` exception if both `user_password` and `user_data` are specified.

Description This method authenticates the principal via the IIOP Listener/Handler so that the principal can access a WebLogic Enterprise domain. This method is functionally equivalent to `SecurityLevel2::PrincipalAuthenticator::authenticate`, but the arguments are oriented to Tuxedo-style authentication.

Note: This method raises `CORBA::BAD_INV_ORDER` if it is called with an invalid `SecurityCurrent` object.

Return Values The following table describes the valid return values.

Return Value	Meaning
<code>Security::AuthenticationStatus::SecAuthSuccess</code>	The authentication succeeded.
<code>Security::AuthenticationStatus::SecAuthFailure</code>	The authentication failed, or the client application was already authenticated and did not call one of the following methods: <code>Tobj::PrincipalAuthenticator::logoff</code> <code>Tobj_Bootstrap::destroy_current</code>
<code>CORBA::INVALID_DOMAIN</code>	The method was used with the <code>corbaloc</code> or <code>corbalocs</code> URL address format.

Tobj::PrincipalAuthenticator::logoff

Synopsis Discards the security context associated with the principal.

OMG IDL Definition

```
void logoff();
```

Description This call discards the security context, but does not close the network connections to the WebLogic Enterprise domain. `Logoff` also invalidates the current credentials. After logging off, invocations using existing object references fail if the authentication type is not `TOBJ_NOAUTH`.

If the principal is currently authenticated to a WebLogic Enterprise domain, calling `Tobj_Bootstrap::destroy_current()` calls `logoff` implicitly.

Note: This method raises `CORBA::BAD_INV_ORDER` if it is called with an invalid `SecurityCurrent` object.

Return Values None.

Tobj::PrincipalAuthenticator::build_auth_data

Synopsis Creates authentication data and attributes for use by
SecurityLevel2::PrincipalAuthenticator::authenticate.

OMG IDL

Definition

```
void build_auth_data(  
    in string          user_name,  
    in string          client_name,  
    in string          system_password,  
    in string          user_password,  
    in UserAuthData    user_data,  
    out Security::Opaque auth_data,  
    out Security::AttributeList privileges  
);
```

Arguments

`user_name`
The WebLogic Enterprise username.

`client_name`
The WebLogic Enterprise client name.

`system_password`
The WebLogic Enterprise client application password.

`user_password`
The user password (default WebLogic Enterprise authentication service).

`user_data`
Client application-specific data (custom WebLogic Enterprise authentication service).

`auth_data`
For use by authenticate.

`privileges`
For use by authenticate.

Note: If `user_name`, `client_name`, or `system_password` is NULL or empty, or exceeds 30 characters, the subsequent `authenticate` method invocation raises the CORBA::BAD_PARAM exception.

Note: The `user_password` and `user_data` parameters are mutually exclusive, depending on the requirements of the authentication service used in the configuration of the WebLogic Enterprise domain. The WebLogic Enterprise default authentication service expects a user password. A customized authentication service may require user data. If both `user_password` and `user_data` are specified, the subsequent authentication call raises the `CORBA::BAD_PARAM` exception.

Description This method is a helper function that creates authentication data and attributes to be used by `SecurityLevel2::PrincipalAuthenticator::authenticate`.

Note: This method raises `CORBA::BAD_INV_ORDER` if it is called with an invalid `SecurityCurrent` object.

Return Values None.

17 Java Security Reference

For information about the security package application programming interface (API), see the [*WebLogic Enterprise Javadoc*](#) on the WebLogic Enterprise documentation CD.

18 Automation Security Reference

This topic contains the Automation method descriptions for the WebLogic Enterprise Security Service. In addition, this topic contains programming examples that illustrate using the Automation methods to implement security in an ActiveX client application.

This topic includes the following sections:

- Method Descriptions
- Programming Example

Note: The Automation security methods do not support certificate-based authentication or the use of the SSL protocol.

Method Descriptions

This section describes the Automation Security Service methods.

DISecurityLevel2_Current

The `DISecurityLevel2_Current` object is a BEA implementation of the CORBA Security model. In this release of the WebLogic Enterprise software, the `get_attributes()`, `set_credentials()`, `get_credentials()`, and `Principal_Authenticator()` methods are supported.

DISecurityLevel2_Current.get_attributes

Synopsis Returns attributes for the Current interface.

MIDL Mapping

```
HRESULT get_attributes(  
    [in] VARIANT attributes,  
    [in,out,optional] VARIANT* exceptionInfo,  
    [out,retval] VARIANT* returnValue);
```

Automation Mapping Function get_attributes(attributes, [exceptionInfo])

Parameters attributes

The set of security attributes (privilege attribute types) whose values are desired. If this list is empty, all attributes are returned.

exceptioninfo

An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client applications, all exception data is returned in the OLE Automation Error Object.

Description This method gets privilege (and other) attributes from the credentials for the client application from the Current interface.

Return Values A variant containing an array of DISecurity_SecAttribute objects. The following table describes the valid return values.

Return Value	Meaning
Security::Public	Empty (Public is returned when no authentication was performed.)
Security::AccessId	Null-terminated ASCII string containing the WebLogic Enterprise username.
Security::PrimaryGroupId	Null-terminated ASCII string containing the WebLogic Enterprise name of the client application.

DISecurityLevel2_Current.set_credentials

Synopsis Sets credentials type.

MIDL Mapping HRESULT set_credentials(
 [in] Security_CredentialType cred_type,
 [in] DISecurityLevel2_Credentials* cred,
 [in,out,optional] VARIANT* exceptionInfo);

Automation Mapping Sub set_credentials(cred_type As Security_CredentialType,
 cred As DISecurityLevel2_Credentials,
 [exceptionInfo])

Description This method can be used only to set SecInvocationCredentials; otherwise, set_credentials raises CORBA::BAD_PARAM. The credentials must have been obtained from a previous call to DISecurityLevel2_Current.get_credentials.

Arguments

cred_type The type of credentials to be set; that is, invocation, own, or nonrepudiation.

cred The object reference to the Credentials object, which is to become the default.

exceptioninfo An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client applications, all exception data is returned in the OLE Automation Error Object.

Return Values None.

DISecurityLevel2_Current.get_credentials

Synopsis Gets credentials type.

MIDL Mapping

```
HRESULT get_credentials(  
    [in] Security_CredentialType cred_type,  
    [in,out,optional] VARIANT* exceptionInfo,  
    [out,retval] DISecurityLevel2_Credentials** returnValue);
```

Automation Mapping

```
Function get_credentials(cred_type As Security_CredentialType,  
    [exceptionInfo]) As DISecurityLevel2_Credentials
```

Description This call can be used only to get SecInvocationCredentials; otherwise, `get_credentials` raises CORBA::BAD_PARAM. If no credentials are available, `get_credentials` raises CORBA::BAD_INV_ORDER.

Arguments

`cred_type`
The type of credentials to get.

`exceptioninfo`
An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client application, all exception data is returned in the OLE Automation Error Object.

Return Values A `DISecurityLevel2_Credentials` object for the active credentials in the client application only.

DISecurityLevel2_Current.principal_authenticator

Synopsis Returns the `PrincipalAuthenticator`.

MIDL Mapping `HRESULT principal_authenticator([out, retval]
DITobj_PrincipalAuthenticator** returnValue);`

Automation Mapping Property `principal_authenticator` As `DITobj_PrincipalAuthenticator`

Description The `PrincipalAuthenticator` returned by the `principal_authenticator` property is of actual type `DITobj_PrincipalAuthenticator`. Therefore, it can be used as a `DISecurityLevel2_PrincipalAuthenticator`.

Note: This method raises `CORBA::BAD_INV_ORDER` if it is called on an invalid `SecurityCurrent` object.

Return Values A `DITobj_PrincipalAuthenticator` object.

DITobj_PrincipalAuthenticator

The `DITobj_PrincipalAuthenticator` object is used to log in to and log out of the WebLogic Enterprise domain. In this release of the WebLogic Enterprise software, the `authenticate`, `build_auth_data()`, `continue_authentication()`, `get_auth_type()`, `logon()`, and `logoff()` methods are implemented.

DITobj_PrincipalAuthenticator.authenticate

Synopsis Authenticates the client application.

MIDL Mapping

```

HRESULT authenticate(
    [in] long                method,
    [in] BSTR                security_name,
    [in] VARIANT             auth_data,
    [in] VARIANT             privileges,
    [out] DISecurityLevel2_Credentials**

                                creds,
    [out] VARIANT*           continuation_data,
    [out] VARIANT*           auth_specific_data,
    [in,out,optional] VARIANT* exceptionInfo,
    [out,retval] Security_AuthenticationStatus* returnValue);

```

Automation Mapping

```

Function authenticate(method As Long, security_name As String,
    auth_data, privileges, creds As DISecurityLevel2_Credentials,
    continuation_data, auth_specific_data,
    [exceptionInfo]) As Security_AuthenticationStatus

```

Arguments

method
 Must be Tobj::TuxedoSecurity. If method is invalid, authenticate raises CORBA::BAD_PARAM.

security_name
 The WebLogic Enterprise username.

auth_data
 As returned by DITobj_PrincipalAuthenticator.build_auth_data. If auth_data is invalid, authenticate raises CORBA::BAD_PARAM.

privileges
 As returned by DITobj_PrincipalAuthenticator.build_auth_data. If privileges is invalid, authenticate raises CORBA::BAD_PARAM.

creds
 Placed into the SecurityCurrent object.

continuation_data
 Always empty.

auth_specific_data
 Always empty.

`exceptioninfo`

An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client application, all exception data is returned in the OLE Automation Error Object.

Description This method authenticates the client application via the IIOP Listener/Handler so that it can access a WebLogic Enterprise domain.

Return Values A `Security_AuthenticationStatus` Enum value. The following table describes the valid return values.

Return Value	Meaning
<code>Security::AuthenticationStatus::SecAuthSuccess</code>	The authentication succeeded.
<code>Security::AuthenticationStatus::SecAuthFailure</code>	The authentication failed, or the client application was already authenticated and did not invoke <code>Tobj::PrincipalAuthenticator::logoff</code> or <code>Tobj_Bootstrap::destroy_current</code> .

DIObj_PrincipalAuthenticator.build_auth_data

Synopsis	Creates authentication data and attributes for use by <code>DITObj_PrincipalAuthenticator.authenticate</code> .
-----------------	---

[illegible]

Automation Mapping	Sub build_auth_data(user_name As String, client_name As String, system_password As String, user_password As String, user_data, auth_data, privileges, [exceptionInfo])
--------------------	--

Arguments	
<code>user_name</code>	The WebLogic Enterprise username.
<code>client_name</code>	A name of the WebLogic Enterprise client application.
<code>system_password</code>	The password for the WebLogic Enterprise client application.
<code>user_password</code>	The user password (default WebLogic Enterprise authentication service).
<code>user_data</code>	Client application-specific data (custom WebLogic Enterprise authentication service).
<code>auth_data</code>	For use by authenticate.
<code>privileges</code>	For use by authenticate.
<code>exceptioninfo</code>	An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client application, all exception data is returned in the OLE Automation Error Object.

Note: If `user_name`, `client_name`, or `system_password` is NULL or empty, or exceeds 30 characters, the subsequent `authenticate` method invocation raises the `CORBA::BAD_PARAM` exception.

Note: The `user_password` and `user_data` parameters are mutually exclusive, depending on the requirements of the authentication service used in the configuration of the WebLogic Enterprise domain. The WebLogic Enterprise default authentication service expects a user password. A customized authentication service may require user data. If both `user_password` and `user_data` are specified, the subsequent authentication call raises the `CORBA::BAD_PARAM` exception.

Description This method is a helper function that creates authentication data and attributes to be used by `DITobj_PrincipalAuthenticator.authenticate`.

Note: This method raises `CORBA::BAD_INV_ORDER` if it is called with an invalid `SecurityCurrent` object.

Return Values None.

DITobj_PrincipalAuthenticator.continue_authentication

Synopsis Always returns `Security::AuthenticationStatus::SecAuthFailure`.

MIDL Mapping

```
HRESULT continue_authentication(  
    [in] VARIANT response_data,  
    [in,out] DISecurityLevel2_Credentials** creds,  
    [out] VARIANT* continuation_data,  
    [out] VARIANT* auth_specific_data,  
    [in,out,optional] VARIANT* exceptionInfo,  
    [out,retval] Security_AuthenticationStatus* returnValue);
```

Automation Mapping

```
Function continue_authentication(response_data,  
    creds As DISecurityLevel2_Credentials, continuation_data,  
    auth_specific_data, [exceptionInfo]) As  
    Security_AuthenticationStatus
```

Description Because the WebLogic Enterprise software does authentication in one step, this method always fails and returns `Security::AuthenticationStatus::SecAuthFailure`.

Return Values Always returns `SecAuthFailure`.

DIObj_PrincipalAuthenticator.get_auth_type

Synopsis Gets the type of authentication expected by the WebLogic Enterprise domain.

MIDL Mapping

```
HRESULT get_auth_type(  
    [in, out, optional] VARIANT* exceptionInfo,  
    [out, retval] Tobj_AuthType* returnValue);
```

Automation Mapping

```
Function get_auth_type([exceptionInfo]) As Tobj_AuthType
```

Argument `exceptioninfo`
An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client application, all exception data is returned in the OLE Automation Error Object.

Description This method returns the type of authentication expected by the WebLogic Enterprise domain.

Note: This method raises `CORBA::BAD_INV_ORDER` if it is called with an invalid `SecurityCurrent` object.

Returned Values A reference to the `Tobj_AuthType` enumeration. The following table describes the valid return values.

Return Value	Meaning
TOBJ_NOAUTH	No authentication is needed; however, the client application can still authenticate itself by specifying a username and a client application name. No password is required. To specify this level of security, specify the <code>NONE</code> value for the <code>SECURITY</code> parameter in the <code>RESOURCES</code> section of the <code>UBBCONFIG</code> file.

Return Value	Meaning
TOBJ_SYSAUTH	<p>The client application must authenticate itself to the WebLogic Enterprise domain, and must specify a username, a name, and a password for the client application.</p> <p>To specify this level of security, specify the APP_PW value for the SECURITY parameter in the RESOURCES section of the UBBCONFIG file.</p>
TOBJ_APPAUTH	<p>The client application must provide proof material that authenticates the client application to the WebLogic Enterprise domain. The proof material may be a password or a digital certificate.</p> <p>To specify this level of security, specify the USER_AUTH value for the SECURITY parameter in the RESOURCES section of the UBBCONFIG file.</p>

DITobj_PrincipalAuthenticator.logon

Synopsis Logs in to the WebLogic Enterprise domain. The correct input parameters depend on the authentication level.

MIDL Mapping

```
HRESULT logon(
    [in] BSTR                user_name,
    [in] BSTR                client_name,
    [in] BSTR                system_password,
    [in] BSTR                user_password,
    [in] VARIANT             user_data,
    [in,out,optional] VARIANT* exceptionInfo,
    [out,retval] Security_AuthenticationStatus*
    returnValue);
```

Automation Mapping

```
Function logon(user_name As String, client_name As String,
    system_password As String, user_password As String,
    user_data, [exceptionInfo]) As Security_AuthenticationStatus
```

Description For remote WebLogic Enterprise client applications, this method authenticates the client application via the IIOP Listener/Handler so that the remote client application can access a WebLogic Enterprise domain. This method is functionally equivalent to `DITobj_PrincipalAuthenticator.authenticate`, but the parameters are oriented to WebLogic Enterprise security.

Arguments

`user_name`
The WebLogic Enterprise username. This parameter is required for `TOBJ_NOAUTH`, `TOBJ_SYSAUTH`, and `TOBJ_APPAUTH` authentication levels.

`client_name`
The name of the WebLogic Enterprise client application. This parameter is required for `TOBJ_NOAUTH`, `TOBJ_SYSAUTH`, and `TOBJ_APPAUTH` authentication levels.

`system_password`
A password for the WebLogic Enterprise client application. This parameter is required for `TOBJ_SYSAUTH` and `TOBJ_APPAUTH` authentication levels.

`user_password`
The user password (default WebLogic Enterprise authentication service). This parameter is required for the `TOBJ_APPAUTH` authentication level.

`user_data`

Application-specific data (custom authentication service). This parameter is required for the TOBJ_APPAUTH authentication level.

Note: If `user_name`, `client_name`, or `system_password` is NULL or empty, or exceeds 30 characters, the subsequent `authenticate` method invocation raises the `CORBA::BAD_PARAM` exception.

Note: If the authorization level is `TOBJ_APPAUTH`, only one of `user_password` or `user_data` may be supplied.

`exceptioninfo`
An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client application, all exception data is returned in the OLE Automation Error Object.

Return Values The following table describes the valid return values.

Return Value	Meaning
<code>Security::AuthenticationStatus::SecAuthSuccess</code>	The authentication succeeded.
<code>Security::AuthenticationStatus::SecAuthFailure</code>	The authentication failed, or the client application was already authenticated and did not call one of the following methods: <code>Tobj::PrincipalAuthenticator::logoff</code> <code>Tobj_Bootstrap::destroy_current</code>

DIObj_PrincipalAuthenticator.logoff

Synopsis	Discards the current security context associated with the WebLogic Enterprise client application.
MIDL Mapping	<code>HRESULT logoff([in, out, optional] VARIANT* exceptionInfo);</code>
Automation Mapping	<code>Sub logoff([exceptionInfo])</code>
Description	<p>This call discards the context associated with the WebLogic Enterprise client application, but does not close the network connections to the WebLogic Enterprise domain. <code>Logoff</code> also invalidates the current credentials. After logging off, calls using existing object references fail if the authentication type is not <code>TOBJ_NOAUTH</code>.</p> <p>If the client application is currently authenticated to a WebLogic Enterprise domain, calling <code>Tobj_Bootstrap.destroy_current()</code> calls <code>logoff</code> implicitly.</p>
Argument	<p><code>exceptioninfo</code></p> <p>An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client applications, all exception data is returned in the OLE Automation Error Object.</p>
Return Values	None.

DISecurityLevel2_Credentials

The `DISecurityLevel2_Credentials` object is a BEA implementation of the CORBA Security model. In this release of the WebLogic Enterprise software, the `get_attributes()` and `is_valid()` methods are supported.

DISecurityLevel2_Credentials.get_attributes

Synopsis Gets the attribute list attached to the credentials.

MIDL Mapping

```
HRESULT get_attributes(  
    [in] VARIANT attributes,  
    [in,out,optional] VARIANT* exceptionInfo,  
    [out,retval] VARIANT* returnValue);
```

Automation Mapping Function get_attributes(attributes, [exceptionInfo])

Arguments

attributes
The set of security attributes (privilege attribute types) whose values are desired. If this list is empty, all attributes are returned.

exceptioninfo
An optional input argument that allows the client application to get additional exception data if an error occurs. For the ActiveX client application, all exception data is returned in the OLE Automation Error Object.

Description This method returns the attribute list attached to the credentials of the client application. In the list of attribute types, you are required to include only the type value(s) for the attributes you want returned in the `AttributeList`. Attributes are not currently returned based on attribute family or identities. In most cases, this is the same result you would get if you called `DISecurityLevel2.Current::get_attributes()`, since there is only one valid set of credentials in the client application at any instance in time. The results could be different if the credentials are not currently in use.

Return Values A variant containing an array of `DISecurity_SecAttribute` objects.

DISecurityLevel2_Credentials.is_valid

Synopsis Checks the status of credentials.

MIDL Mapping

```
HRESULT is_valid(  
    [out] IDispatch** expiry_time,  
    [in,out,optional] VARIANT* exceptionInfo,  
    [out,retval] VARIANT_BOOL* returnValue
```

Automation Mapping

```
Function is_valid(expiry_time As Object,  
    [exceptionInfo]) As Boolean
```

Description This method returns `TRUE` if the credentials used are active at the time; that is, you did not call `DITobj_PrincipalAuthenticator.logoff` or `destroy_current`. If this method is called after `DITobj_PrincipalAuthenticator.logoff()`, `FALSE` is returned. If this method is called after `destroy_current()`, the `CORBA::BAD_INV_ORDER` exception is raised.

Return Values The output `expiry_time` as a `DITimeBase_UtcT` object set to `max`.

Programming Example

This section contains the portions of an ActiveX client application that implement the following:

- Using the Bootstrap object to obtain the SecurityCurrent object
- Getting the Principal Authenticator object from the SecurityCurrent object
- Using Tuxedo-style authentication
- Logging off the WebLogic Enterprise domain

Listing 18-1 ActiveX Client Application That Uses Tuxedo-Style Authentication

```
Set objSecurityCurrent = objBootstrap.CreateObject("Tobj.SecurityCurrent")
Set objPrincipalAuthenticator = objSecurityCurrent.principal_authenticator

    AuthorityType = objPrincipalAuthenticator.get_auth_type
    If AuthorityType = TOBJ_APPAUTH Then logonStatus =
        oPrincipalAuthenticator.Logon(
            UserName, _
            ClientName, _
            SystemPassword, _
            UserPassword
            User Data)
End If

    objPrincipalAuthenticator.logoff()
```

Index

A

- administration steps
 - certificate-based authentication 3-12
 - link-level encryption 3-23
 - the SSL protocol 3-26
 - username/password authentication 3-8
- authentication
 - certificate-based 3-9
 - username/password 3-5
- authorized users
 - defining 7-3
- AUTHSRV
 - code example 7-2
 - configuring 7-2
 - described 3-5
 - use with username/password authentication 3-8

B

- building
 - Secure Simpapp sample application 11-2
 - Security sample application 11-2

C

- certificate authorities
 - defined 4-9
 - obtaining a digital certificate for 4-9
- certificate-based authentication
 - administration steps 3-12

- C++ code example 10-17
- configuration illustrated 3-14
- defining JNDI environment
 - properties 12-6
- described 3-9
- development process 3-12
- how it works 3-10
- illustrated 3-9
- Java code example 10-19
- programming steps 3-12
- requirements 3-11
- sample UBBCONFIG file 7-13
- writing the client application 10-17
- cipher suites
 - supported by the WLE product 2-10
- compiling
 - client applications
 - Secure Simpapp sample application 11-10
 - server applications
 - Secure Simpapp sample application 11-10
- concepts
 - AUTHSRV 3-5
 - certificate-based authentication 3-9
 - digital certificates 3-23
 - link-level encryption 3-19
 - SSL protocol 3-23
 - username/password authentication 3-5
- configuring
 - a port for SSL communications 6-2

- host matching 6-3
- setting session renegotiation 6-7
- setting the encryption strength 6-4
- the SSL protocol
 - CORBA C++ ORB 6-2
 - CORBA Java ORB 6-2
 - IIOP Listener/Handler 6-2
- CORBA C++ client applications
 - starting
 - Secure Simpapp sample application 11-10
- CORBA C++ ORB
 - defining a port for SSL
 - communications 6-2
 - enabling host matching 6-3
 - setting the encryption strength 6-4
- CORBA Java client applications
 - starting
 - Secure Simpapp sample application 11-10
- CORBA Java ORB
 - defining a port for SSL
 - communications 6-2
 - enabling host matching 6-3
 - example of configuring the SSL
 - protocol 6-9
- CORBA module
 - described 15-2
- CORBA Module IDL 15-2
- corbaloc URL Address format
 - described 10-5
- corbalocs URL Address format
 - described 10-5
- Credentials object
 - described 14-7
- customer support contact information xiii

D

- Data types
 - security module 15-4

I-2 Using Security

- deployment descriptor
 - specifying security roles 12-5
- development process
 - certificate-based authentication 3-12
 - for security in EJBs 12-2
 - the SSL protocol 3-26
 - username/password authentication 3-7
- digital certificates
 - certificate-based authentication 3-9
 - for principals 4-7
 - obtaining 4-6
 - publishing in LDAP 4-6
 - SSL protocol 3-23
 - troubleshooting 13-9
- directory location of source files
 - Secure Simpapp sample application 11-3, 11-5
- documentation, where to find it xii

E

- EJBs
 - assigning security roles to methods 12-3
 - code example
 - certificate-based
 - authentication 12-11
 - username/password
 - authentication 12-11
 - depoymnt descriptor 12-5
 - description of security 12-2
 - how authentication works 12-2
 - property keys for security 12-8
 - specifying certificate-based
 - authentication 12-8
 - specifying username/password
 - authentication 12-8
 - steps for adding security to 12-2
 - using URL Address formats 12-7
- encryption
 - setting encryption strength 6-4
 - values 6-5

environment variables

APPDIR 11-8

JAVA_HOME 11-7

JDKDIR 11-8

Secure Simpapp sample application 11-7

TOBJADDR 11-8

TUXCONFIG 11-8

TUXDIR 11-7

F

file protections

Secure Simpapp sample application 11-7

H

host matching

enabling 6-3

values 6-4

I

IIOP Listener/Handler

configuring session renegotiation 6-7

defining a port for SSL communications
6-2

enabling host matching 6-3

SEC_PRINCIPAL_LOCATION
parameter 6-8

SEC_PRINCIPAL_NAME parameter 6-
8

SEC_PRINCIPAL_PASSVAR
parameter 6-8

setting security parameters 6-7

setting the encryption strength 6-4

use with certificate-based
authentication 3-9

use with the SSL protocol 3-23

invocation_options_required method

C++ code example 10-21

described 10-20

Java code example 10-22

ISL command

configuring session renegotiation 6-7

enabling host matching 6-4

example 6-8

setting the encryption strength 6-4

specifying a port for SSL

communications 6-2

J

JAVA_HOME parameter

Secure Simpapp sample application 11-7

JDKDIR parameter

setenv file 11-8

JNDI environment properties

for certificate-based authentication 12-6

for username/password authentication
12-6

WLEContext.INITIAL_CONTEXT_
FACTORY 12-6

WLEContext.PROVIDER_URL 12-7

WLEContext.SECURITY_
AUTHENTICATION 12-8

JNDI factory

use in authentication 12-2

joint client/server applications

using the SSL protocol 6-3

L

LDAP directory service

directory structure 4-4

search filter file 4-5

use with certificate-based
authentication 3-11

use with the SSL protocol 3-26

use with WLE security 4-3

LDAP Search Filter file

modifying 4-5

stanzas used by SSL protocol 4-6

- stanzas used for certificate-based authentication 4-6
- link-level encryption
 - administration steps 3-23
 - described 3-19
 - development process 3-23

M

- makefile
 - Secure Simpapp sample application 11-6

O

- OMG IDL
 - CORBA module 15-2
 - Security Level 2 module 15-7
 - Security module 15-4
 - SecurityLevel 1 module 15-6
 - TimeBase module 15-2
 - Tobj module 15-7

P

- Peer Rules file
 - described 4-11
 - elements 4-11
 - example 4-11
 - syntax 4-12
- PrincipalAuthenticator object
 - certificate-based authentication 14-5
 - described 14-4
 - using in client applications 10-6
 - WLE extensions 14-6
- printing product documentation xiii
- private keys
 - example 4-8
 - for principals 4-7
 - format 4-8
 - location 4-7

- protocols
 - link-level encryption 3-19
 - SSL 3-23

R

- related information xiii
- runme command
 - description 11-10
 - files generated by 11-11, 11-12

S

- SEC_PRINCIPAL_LOCTION parameter
 - defined 6-8
- SEC_PRINCIPAL_NAME parameter
 - defined 6-8
- SEC_PRINCIPAL_PASSVAR parameter
 - defined 6-8
- Secure Simpapp sample application
 - building 11-2
 - changing protection on files 11-7
 - compiling the Java client
 - application 11-10
 - compiling the Java server
 - application 11-10
 - description 10-16
 - development process 10-16
 - illustrated 10-16
 - loading the UBBCONFIG file 11-10
 - locations of files 11-3
 - required environment variables 11-7
 - runme command 11-10
 - setting up the work directory 11-3
 - source files 11-3, 11-5
 - starting the Java client application 11-15
 - starting the Java server application 11-15
 - using the client applications 11-15
- Security Level 2 module
 - described 15-7
- Security module

- described 15-4
- SECURITY parameter
 - defining in UBBCONFIG file 7-6
 - setting for username/password authentication 3-8
 - values for 7-6
- security roles
 - assigning to EJB methods 12-3
 - defining in deployment descriptor 12-5
 - syntax rules 12-5
- Security sample application
 - description 10-6
 - illustrated 10-7
 - location of files 10-7
 - PrincipalAuthenticator object 10-6
 - SecurityCurrent object 10-6
- SecurityCurrent object
 - described 14-8
 - using in client applications 10-6
- SecurityLevel 1 module
 - described 15-6
- source files
 - Secure Simpapp sample application 11-3, 11-5
- SSL parameters
 - SEC_PRINCIPAL_LOCATION 3-27
 - SEC_PRINCIPAL_NAME 3-27
 - SEC_PRINCIPAL_PASSVAR 3-27
- SSL protocol
 - administration steps 3-26
 - configuration illustrated 3-27
 - described 3-23
 - development process 3-26
 - how it works 3-24
 - illustrated 3-23
 - requirements 3-26
- support
 - technical xiv

T

- TimeBase module
 - described 15-2
- TimeBase Module IDL 15-2
- tmboot command
 - Secure Simpapp sample application 11-15
- tmloadcf command
 - Secure Simpapp sample application 11-10
- Tobj module
 - described 15-7
- tpgrpadd command
 - defining security groups 3-8, 7-3
- tpusradd command
 - defining users for security 3-8, 7-3
- troubleshooting
 - bootstrapping problems 13-6
 - callback objects 13-9
 - certificate-based authentication problems 13-5
 - configuration problems 13-8
 - digital certificates 13-9
 - IIOP Listener/Handler startup problems 13-7
 - ORB initialization problems 13-3
 - tracing 13-1
 - Ulog file 13-1
 - username/password authentication problems 13-4
- Trusted Certificate Authority file
 - described 4-9
 - example 4-9
- TUXCONFIG parameter
 - setenv file 11-8
- TUXDIR parameter
 - Secure Simpapp sample application 11-7

U

UBBCONFIG 3-12

UBBCONFIG file

- configuring the authentication server 7-2
- defining a security level 7-5
- defining link-level encryption 3-23
- defining security parameters for the IIOP Listener/Handler 6-8
- example of certificate-based authentication 7-13
- example of username/password authentication 7-9
- link-level encryption 3-23
- Secure Simpapp sample application 11-10
- username/password authentication 3-8

URL Address formats

- certificate-based authentication 3-10
- corbaloc 10-2, 10-5
- corbalocs 10-2, 10-5
- described 10-2
- Host and Port 10-4
- syntax 10-3
- the SSL protocol 3-24
- username/password authentication 3-34
- using with EJBs 12-7

username/password authentication

- administration steps 3-7
- application password 3-5
- C++ example
 - SecurityLevel2
 - PrincipalAuthenticator 10-9
 - Tobj PrincipalAuthenticator 10-12
- defining JNDI environment properties 12-6
- defining users and groups 3-8
- described 3-5
- development process 3-7
- how it works 3-5

illustrated 3-5

interfaces explained 10-8

Java example

SecurityLevel2

PrincipalAuthenticator
10-10

Tobj PrincipalAuthenticator 10-14

programming steps 3-7

sample UBBCONFIG file 7-9

system authentication 3-5

writing the client application 10-7

W

WLE domain

adding security to 10-6

WLE Security model

- accessing objects 14-3
- administrative control 14-3
- authenticating principals 14-2
- components 14-3
 - Credentials object 14-7
 - PrincipalAuthenticator object 14-4
 - SecurityCurrent object 14-8
- described 14-2

WLE Security Pack

- described 4-2
- use with certificate-based authentication 3-11
- use with SSL protocol 3-26

WLEContext.INITIAL_CONTEXT
_FACTORY property 12-6

WLEContext.PROVIDER_URL property
12-7

WLEContext.SECURITY_
AUTHENTICATION property 12-
8