BEA Tuxedo

Using Security in CORBA Applications

BEA Tuxedo 8.0
Document Edition 8.0
June 2001

Using Security in CORBA Applications

Document Edition	Date	Software Version
8.0	June 2001	BEA Tuxedo 8.0

Contents

About This Document

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

Note: Release 8.0 of the BEA Tuxedo product includes environments that allow you to build both Application-to-Transaction Monitor Interfaces (ATMI) and CORBA applications. This topic explains how to implement security in a CORBA application. For information about implementing security in an ATMI application, see *Using Security in ATMI Applications*.

This document includes the following topics:

- Chapter 1, “Overview of the CORBA Security Features,” presents an overview of the security features for CORBA in the BEA Tuxedo product.
- Chapter 2, “Introduction to the SSL Technology,” introduces the concepts associated with a Public Key Infrastructure (PKI).
- Chapter 3, “Fundamentals of CORBA Security,” presents an indepth discussion of the features in the CORBA 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 CORBA applications that use the Secure Sockets Layer (SSL) protocol and certificate 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 or the CORBA C++ ORB so that it can be used with the Secure Sockets Layer (SSL) protocol and certificate authentication.

- Chapter 7, “Configuring Authentication,” explains the configuration tasks required when using authentication in a CORBA application.
- Chapter 8, “Configuring Single Sign-on,” explains the configuration tasks required when using trusted connection pools in a CORBA application.
- Chapter 9, “Configuring Security Plug-ins,” explains how to register Security Plug-Ins in the CORBA environment.
- Chapter 10, “Writing a CORBA Application that Implements Security,” explains how the bootstrapping options work and describes implementing password authentication and certificate authentication in 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, “Troubleshooting,” provides troubleshooting tips that can be used when solving problems that occur with the security portion of a CORBA application.
- Chapter 13, “CORBA Security APIs,” introduces the security model in CORBA applications and the functional components of the security model.
- Chapter 14, “Security Modules,” includes the Object Management Group (OMG) Interface Definition Language (IDL) for the modules used by the CORBA Security service.
- Chapter 15, “C++ Security Reference,” includes the C++ method descriptions.
- Chapter 16, “Java Security Reference,” includes the Java method descriptions.
- Chapter 17, “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 CORBA applications and system administrators who are responsible for setting up and maintaining the security infrastructure in an enterprise.

e-docs Web Site

The BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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, BEA Tuxedo, distributed object computing, transaction processing, C++ and Java programming, see the *CORBA Bibliography* in the BEA Tuxedo online documentation.

Contact Us!

Your feedback on the BEA Tuxedo 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 BEA Tuxedo documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA Tuxedo 8.0 release.

If you have any questions about this version of BEA Tuxedo, or if you have problems installing and running BEA Tuxedo, 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.

Convention	Item
italics	Indicates emphasis or book titles.

`monospace text`	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.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	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.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

Convention	Item
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ 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.
	Example:
	`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 CORBA Security Features

This topic includes the following sections:

- The CORBA Security Features
- The CORBA Security Environment
- Single Sign-on in the CORBA Security Environment
- BEA Tuxedo Security SPIs

Note: Release 8.0 of the BEA Tuxedo product includes environments that allow you to build both Application-to-Transaction Monitor Interfaces (ATMI) and CORBA applications. This topic explains how to implement security in a CORBA application. For information about implementing security in an ATMI application, see *Using Security in ATMI Applications*.

The CORBA Security Features

Security refers to techniques for ensuring that data stored in a computer or passed between computers is not compromised. Most security measures involve proof material and data encryption, 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.

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 CORBA security features of the BEA Tuxedo product lets you establish secure connections between client and server applications. It has the following features:

- Authentication of CORBA C++ and Java client applications to the BEA Tuxedo domain. Authentication can be accomplished using a standard username/password combination or 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++ and Java client applications can establish SSL sessions with a BEA Tuxedo domain. BEA Tuxedo client applications can use LLE to protect network traffic between bridges and domains.
- A single sign-on environment between the BEA WebLogic Server™ and CORBA environments using WebLogic Enterprise Connectivity. This feature allows the propagation of security information about the requesting WebLogic Server User to the BEA Tuxedo domain over network connections that are part of a trusted 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 CORBA 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 security features of the CORBA environment, you need to install a license that enable the use of the SSL protocol, LLE, and PKI. For information about installing the license for the security features, see the *Installing the BEA Tuxedo System*.

Note: *Using Security in CORBA Applications* describes the security features of the CORBA environment in the BEA Tuxedo product. For a complete description of using the security features in the ATMI environment in the BEA Tuxedo product, see *Using Security in ATMI Applications*.

Table 1-1 summarizes the features in the CORBA security features in the BEA Tuxedo product.

Table 1-1 CORBA Security Features

Security Features	**Description**	**Service Provider Interface (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 certificate authentication.
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 CORBA application.	N/A	RC4 symmetric key encryption.

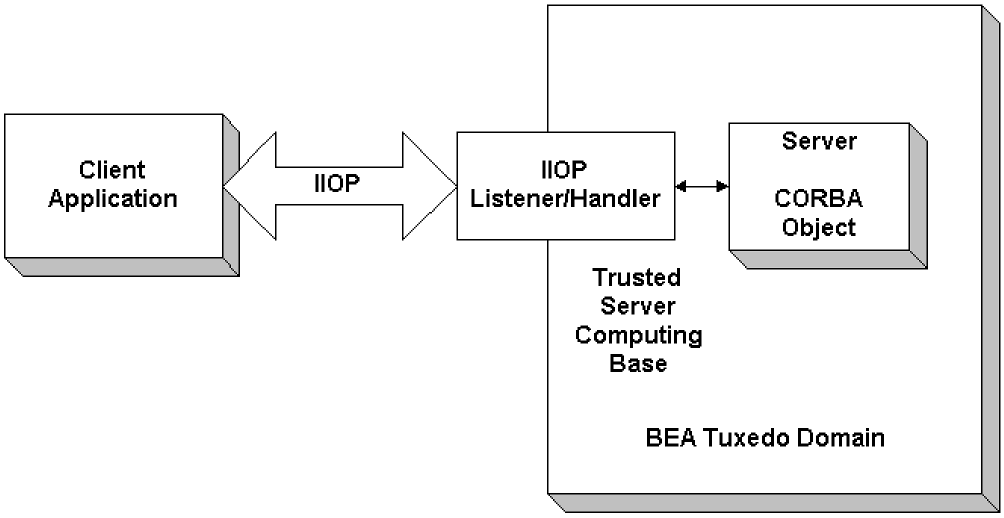
Table 1-1 CORBA Security Features (Continued)

Security Features	Description	Service Provider Interface (SPI)	Default Implementation
The Secure Sockets Layer (SSL) protocol	Uses asymmetric encryption to establish data privacy for messages moving over network links between BEA Tuxedo domains.	N/A	The SSL version 3.0 protocol.
Single Sign-On	Propagates the security identity of a WebLogic Server User identity to a BEA Tuxedo domain.	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 3.0 allowing mutual authentication based on X.509 digital certificates.	Implemented as the following interfaces: ■ Public key initialization ■ Key management ■ Certificate lookup ■ Certificate parsing ■ Certificate validation ■ Proof material mapping	Default public key security supports the following algorithms: ■ RSA for key exchange. ■ DES and its variants RC2 and RC4 for bulk encryption. ■ MD5 and SHA for message digests.

The CORBA Security Environment

Direct end-to-end mutual authentication in a distributed enterprise middleware environment such as the BEA Tuxedo CORBA 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 processing each service request. Instead, CORBA applications in a BEA Tuxedo product implements a delegated trust authentication model as shown in Figure 1-1.

Figure 1-1 Delegated Trust Model



In a delegated trust model, principals (generally users of client applications) authenticate to a trusted system gateway process. In the case of the CORBA applications, 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 BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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 CORBA application. The default behavior of the BEA Tuxedo product is to encrypt the authentication information but not to protect the message sent between the client application and the BEA Tuxedo 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 BEA Tuxedo system interrogates the forwarded tokens of the requesting principal to determine if the request should be processed or denied. The CORBA security features will, based on the decision of the authorization implementation, deny the processing of any request on an object for which the requesting principal has no permission to access.

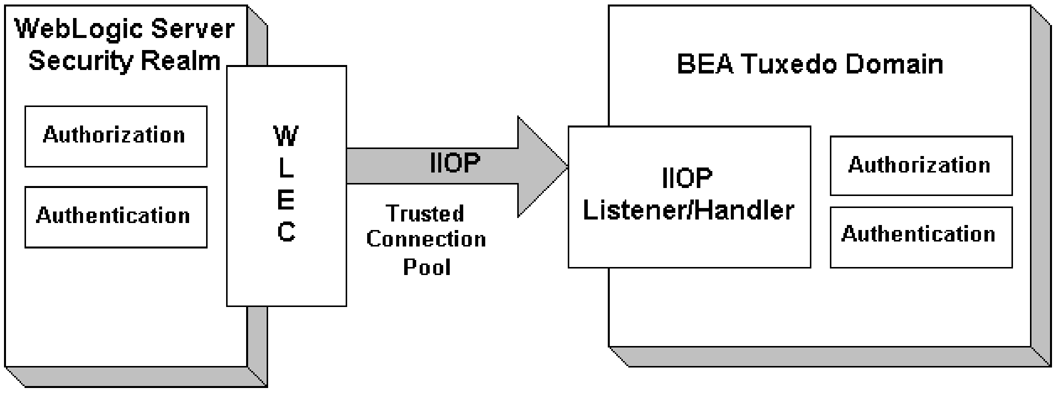
Single Sign-on in the CORBA Security Environment

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

Note: The single sign-on functionality in the CORBA security environment of the BEA Tuxedo product is unidirectional. You can only propagate a principal’s identity from the WebLogic Server security realm to the BEA Tuxedo domain.

Figure 1-2 illustrates how single sign-on works in the CORBA security environment.

Figure 1-2 Single Sign-on in the CORBA Security Environment



When using single sign-on, the security identity of a WebLogic Server User is propagated as part of the service context of a IIOP request sent to a CORBA object in a BEA Tuxedo 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 password and certificate authentication can be used to establish a trusted connection pool.

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

To facilitate the mapping of principal identities, the IIOP 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 CORBA object. The target CORBA object can then use these tokens to determine information about the initiator of the request, including the identity of the principal.

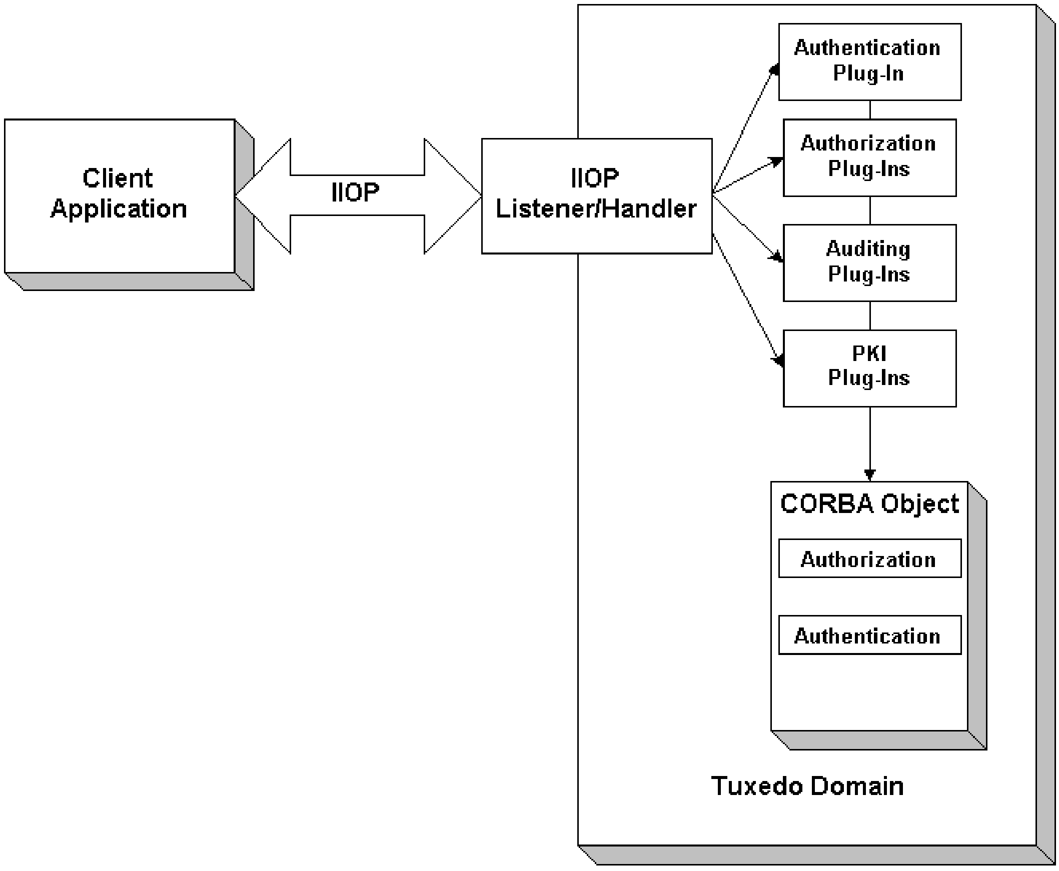
The SSL protocol can be used to protect the confidentiality and integrity of the request from the WebLogic Server realm. SSL encryption is provided for IIOP requests to CORBA objects in the BEA Tuxedo domain. In order to protect the request, both WebLogic Connectivity and the CORBA application must be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

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

BEA Tuxedo Security SPIs

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

Figure 1-3 Architecture for the BEA Tuxedo Security Service Provider Interfaces



The BEA Tuxedo product provides interfaces for the types of security plug-ins listed in Table 1-2.

Table 1-2 The BEA Tuxedo Security Plug-Ins

Plug-In	Description
Authentication	Allows communicating processes to mutually prove identification.
Authorization	Allows system administrators to control access to CORBA applications. Specifically, an administrator can use authorization to allow or disallow principals to use resources or services provided by a CORBA 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 CORBA 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 BEA Tuxedo 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.

CHAPTER

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 CORBA 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

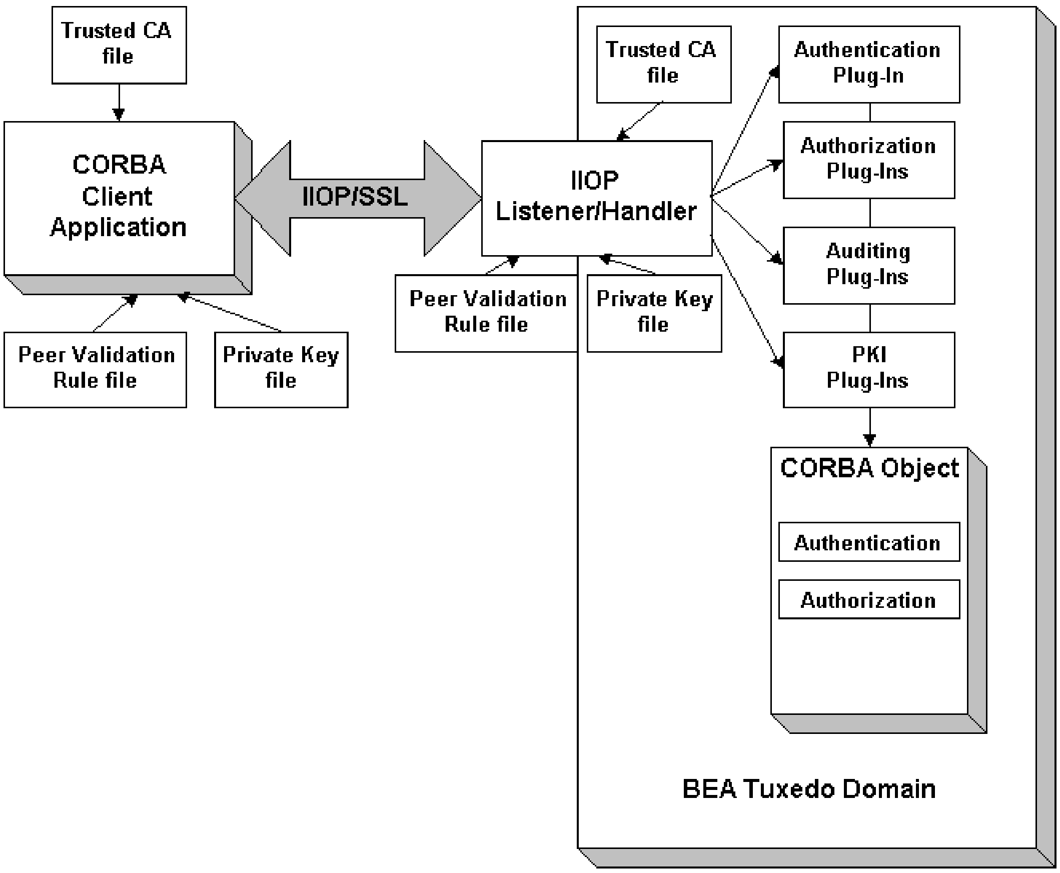
  Authentication is the ability to ascertain with whom you are speaking. By using digital certificates and public key security, CORBA client and server applications can each be authenticated to the other. This allows the two parties to be certain they are communicating with someone they trust. The SSL protocol provides a mechanism that can be used to authenticate principals to a BEA Tuxedo domain using X.509 digital certificates. The use of certificate authentication can be used as an alternative to 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 pair of asymmetric keys 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. Conversely, data encrypted with the private key can be decrypted only with the public 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 CORBA security environment.

Figure 2-1 The SSL Protocol in the CORBA Security Environment



When using the SSL protocol in the CORBA 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, principals present 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 and the infrastructure needed to use digital certificates is available in the BEA Tuxedo product by installing a license available in the product installation. For more information, see *Installing the BEA Tuxedo System*.

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 (CA), 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 CORBA security environment 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 CORBA security environment, 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.

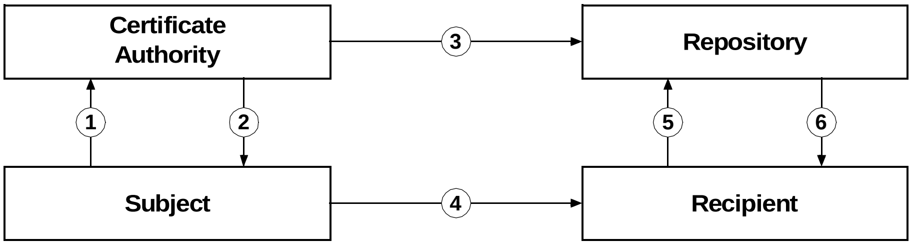
There is no single standard public key infrastructure today, though efforts are underway to define one. It is not yet clear whether a standard will be established or multiple independent PKIs will evolve with varying degrees of interoperability. In this sense, the state of PKI technology today can be 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 BEA Tuxedo 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 BEA Tuxedo customers to use a PKI security solution with the PKI software from their vendor of choice. See "Single Sign-on" on page 3-24 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 BEA Tuxedo 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 CORBA security environment also supports digital signature algorithms. Digital signature algorithms are simply public key algorithms used to provide digital signatures.

The BEA Tuxedo 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 CORBA security environment 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 the public key security in the CORBA security environment restricts the key length to 128 bits.

Customers of the BEA Tuxedo product cannot expand or modify this list of algorithms.

Supported Message Digest Algorithms

The CORBA security environment 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 CORBA security environment supports the cipher suites described in Table 2-1.

Table 2-1 SSL Cipher Suites Supported by the CORBA Security Environment

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 CORBA security environment 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.

CHAPTER

3 Fundamentals of CORBA Security

This topic includes the following sections:

- Link-Level Encryption
- Password Authentication
- The SSL Protocol
- Certificate Authentication
- Using an Authentication Plug-in
- Authorization
- Auditing
- Single Sign-on
- PKI Plug-ins
- Commonly Asked Questions About the CORBA Security Features

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 BEA Tuxedo system messages or CORBA 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 BEA Tuxedo 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 to encrypt communication between machines and/or domains in a CORBA application..

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

There are three levels of LLE security: 0-bit (no encryption), 56-bit (Export), and 128-bit (Domestic). The Export LLE version allows 0-bit and 56-bit encryption. The Domestic 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.

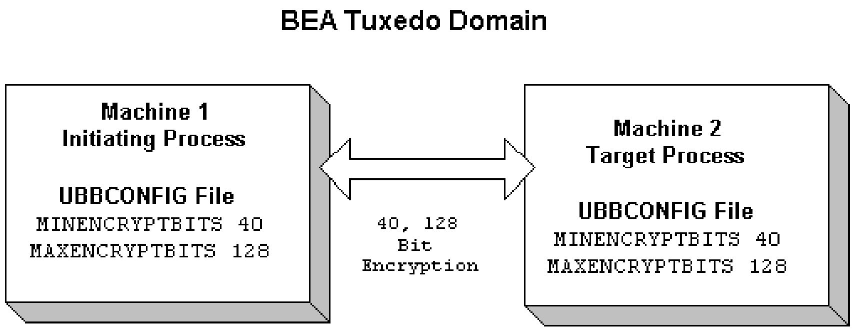
   For convenience, the two parameters are denoted as (`min`, `max`). For example, the values (56, 128) for a process mean that the process accepts at least 56-bit encryption but can support up to 128-bit encryption.
2. An initiator process begins the communication session.
3. A target process receives 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 negotiated encryption level.

Figure 3-1 illustrates these steps.

Figure 3-1 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 BEA Tuxedo 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 BEA Tuxedo 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 Export 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 Domestic 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 3-1 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 3-1 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 3-1 Interprocess Negotiation Results (Continued)

	(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

To use LLE in a CORBA application, you need to install a license that enables the use of LLE. For information about installing the license, see *Installing the BEA Tuxedo System*.

The implementation of LLE is an administrative task. The system administrators for each CORBA application set *`min-max`* values in the `UBBCONFIG` file that control encryption strength. When the two CORBA 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.

Password Authentication

The CORBA security environment supports a password mechanism to provide authentication to existing CORBA applications and to new CORBA applications that are not prepared to deploy a full Public Key Infrastructure (PKI). When using password authentication, the applications that initiate invocations on CORBA objects authenticate themselves to the BEA Tuxedo domain using a defined username and password.

The following levels of password authentication are provided:

- None—indicates that no password or access checking is performed in the CORBA application.
- Application Password—indicates that users are required to supply a domain password in order to access the CORBA application.
- User Authentication—indicates that users are required to supply an application password as well as the domain password in order to access the CORBA application.
- ACL—indicates that authorization is used in the CORBA application and access control checks are performed on interfaces, queue names, and event names. If an associated ALC is not found for a user, it is assumed that access is granted.
- Mandatory ACL—indicates that authorization is used in the CORBA application and access control checks are performed on interfaces, queue names, and event names. The value of Mandatory ACL is similar to ACL, but permission is denied if an associated ACL is not found for the user.

When using Password authentication, you have the option of using the `Tobj::PrincipalAuthenticator::logon()` or the `SecurityLevel2::PrincipalAuthenticator::authenticate()` methods in your client application.

If you use password authentication, the SSL protocol can be used to provide confidentiality and integrity to communication between applications. For more information, see “The SSL Protocol” on page 3-10.

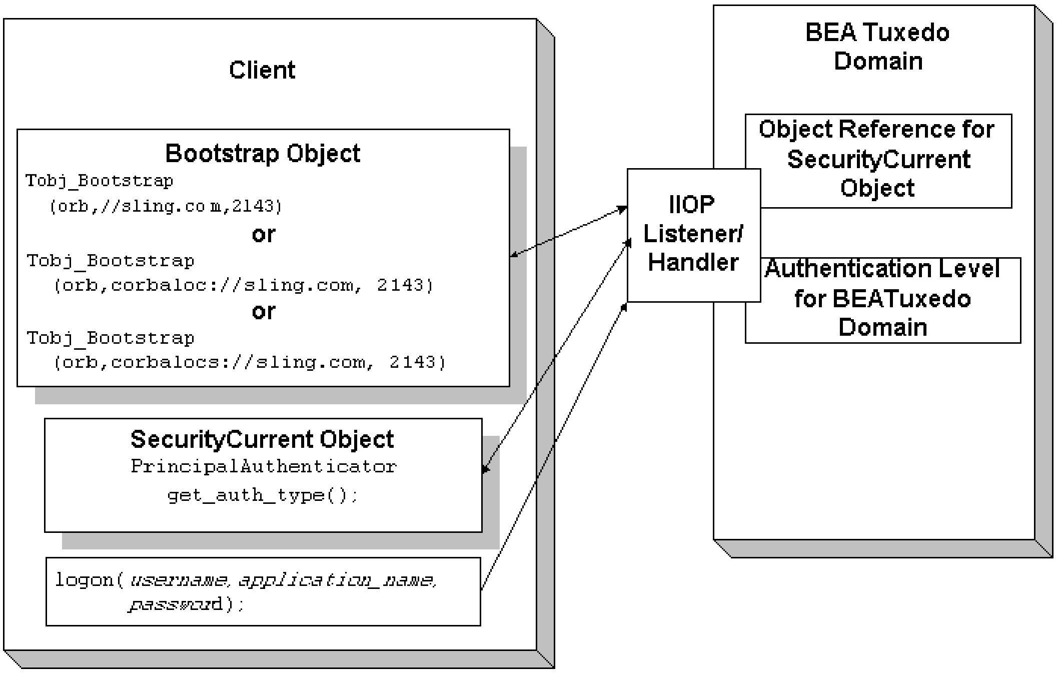
How Password Authentication Works

Password authentication works in the following way:

1. The initiating application accesses the BEA Tuxedo domain in one of the following ways:
   - Through the CORBA Interoperable Naming Service (INS) Bootstrapping mechanism. Use this mechanism if you are using a client ORB from another vendor. For more information about using CORBA INS, see the *CORBA Programming Reference* in the BEA Tuxedo online documentation
   - The BEA Bootstrapping mechanism. Use this mechanism if you are using BEA CORBA client applications.
2. The initiating application obtains credentials for the user. The initiating application must provide proof material to be used by the BEA Tuxedo 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 BEA Tuxedo 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 CORBA object in the BEA Tuxedo 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.
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 CORBA object.

Figure 3-2 illustrates these steps.

Figure 3-2 How Password Authentication Works


Client
Bootstrap Object
Tobj_Bootstrap (orb,//sling.co m,2143)
or
Tobj_Bootstrap (orb,corbaloc://sling.com, 2143)
or
Tobj_Bootstrap (orb,corbalocs://sling.com, 2143)
SecurityCurrent Object
PrincipalAuthenticator
get_auth_type();
logon(username, application_name, password);
IIOP Listener/ Handler
BEA Tuxedo Domain
Object Reference for SecurityCurrent Object
Authentication Level for BEATuxedo Domain

Development Process for Password Authentication

Defining password authentication for a CORBA application includes administration and programming steps. Table 3-2 and Table 3-3 list the administration and programming steps for password authentication. For a detailed description of the administration steps for password authentication, see "Configuring Authentication" on page 7-1. For a complete description of the programming steps, see "Writing a CORBA Application That Implements Security" on page 10-1.

Table 3-2 Administration Steps for Password Authentication

Step	Description
1	Set the `SECURITY` parameter in the `UBBCONFIG` file to `APP_PW`, `USER_AUTH`, `ACL`, or `MANDATORY_ACL`.

Table 3-2 Administration Steps for Password Authentication (Continued)

Step	Description
2	If you defined the `SECURITY` parameter as `USER_AUTH, ACL`, or `MANDATORY_ACL`, configure the authentication server (`AUTHSRV`) in the `UBBCONFIG` file.
3	Use the `tpusradd` and `tpgrpadd` commands to define lists of authorized users and groups including the IIOP Listener/Handler.
4	Use the `tmloadcf` command to load the `UBBCONFIG` file. When the `UBBCONFIG` file is loaded, the system administrator is prompted for a password. The password entered at this time becomes the password for the CORBA application.

Table 3-3 Programming Steps for Password Authentication

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

The SSL Protocol

The BEA Tuxedo 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 CORBA security environment 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 BEA Tuxedo domain.

Like LLE, the SSL protocol can be used with password authentication to provide confidentiality and integrity to communication between the client application and the BEA Tuxedo domain. When using the SSL protocol with password authentication, you are prompted for the password of the IIOP Listener/Handler defined by the `SEC_PRINCIPAL_NAME` parameter when you enter the `tmloadcf` command.

How the SSL Protocol Works

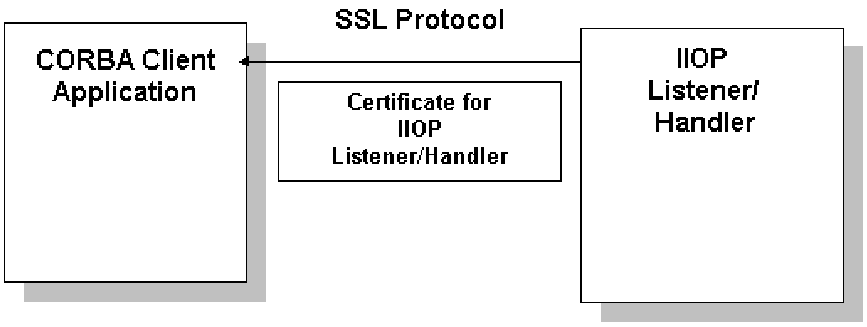
The SSL protocol works in the following manner:

1. The IIOP Listener/Handler presents its digital certificate to the initiating application.
2. The initiating application compares the digital certificate of the IIOP Listener/Handler against its list of trusted certificate authorities.
3. If the initiating application validates the digital certificate of the IIOP Listener/Handler, the application and the IIOP Listener/Handler establish an SSL connection.

   The initiating application can then use either password or certificate authentication to authenticate itself to the BEA Tuxedo domain.

Figure 3-3 illustrates how the SSL protocol works.

Figure 3-3 How the SSL Protocol Works in a CORBA Application



Requirements for Using the SSL Protocol

To use the SSL protocol in a CORBA application, you need to install a license that enables the use of the SSL protocol and PKI. For information about installing the license for the security features, see *Installing the BEA Tuxedo System*.

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

Development Process for the SSL Protocol

Using the SSL protocol in a CORBA application is primarily an administration process. Table 3-5 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 password authentication or certificate authentication in your CORBA application. For more information, see "Writing a CORBA Application That Implements Security" on page 10-1.

Note: If you are using the BEA CORBA C++ 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 3-4 Administration Steps for the SSL Protocol

Step	Description
1	Set up an LDAP-enabled directory service. You will be prompted for the name of the LDAP server during the installation of the BEA Tuxedo product.
2	Install the license for the SSL protocol.
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	Set the `SECURITY` parameter in the `UBBCONFIG` file to `NONE`.
7	Define a port for secure communication on the IIOP Listener/Handler using the `-S` option of the ISL command.
8	Create a Trusted Certificate Authority file (`trust_ca.cer`) that defines the certificate authorities trusted by the IIOP Listener/Handler.

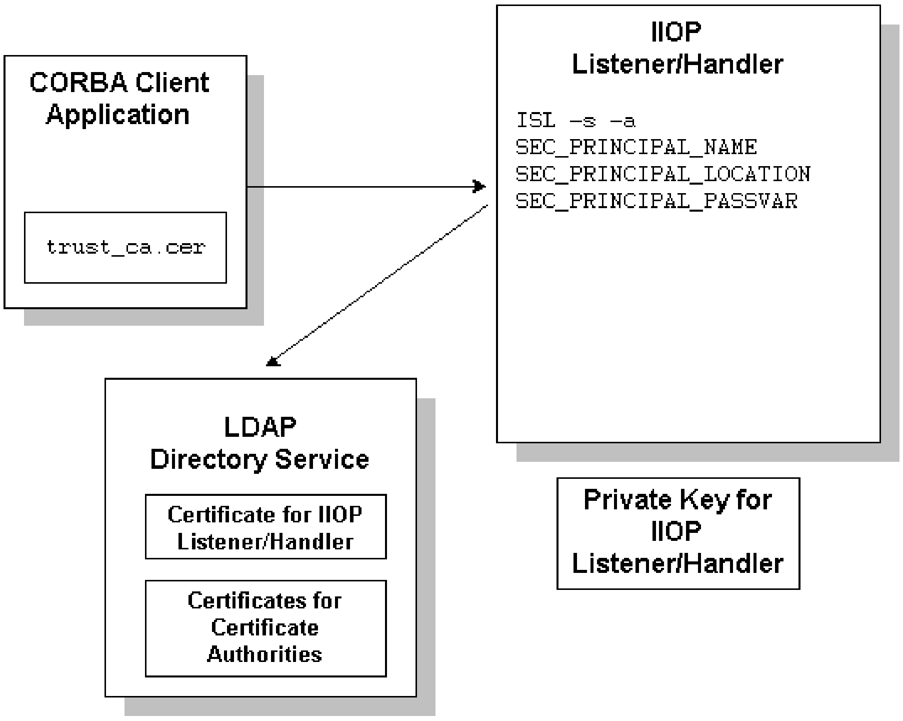
Table 3-4 Administration Steps for the SSL Protocol (Continued)

Step	Description
9	Use the `tmloadcf` command to load the `UBBCONFIG` file.
10	Optionally, create a Peer Rules file (`peer_val.rul`) for the IIOP Listener/Handler.
11	Optionally, modify the LDAP Search filter file to reflect the directory hierarchy in place in your enterprise.

If you use the SSL protocol with password authentication, you need to set the `SECURITY` parameter in the `UBBCONFIG` file to desired level of authentication and if appropriate, configure the Authentication Server (`AUTHSRV`). For information about the administration steps for password authentication, see “Password Authentication” on page 3-6.

Figure 3-4 illustrates the configuration of a CORBA application that uses the SSL protocol.

Figure 3-4 Configuration for Using the SSL Protocol in a CORBA Application



Certificate Authentication

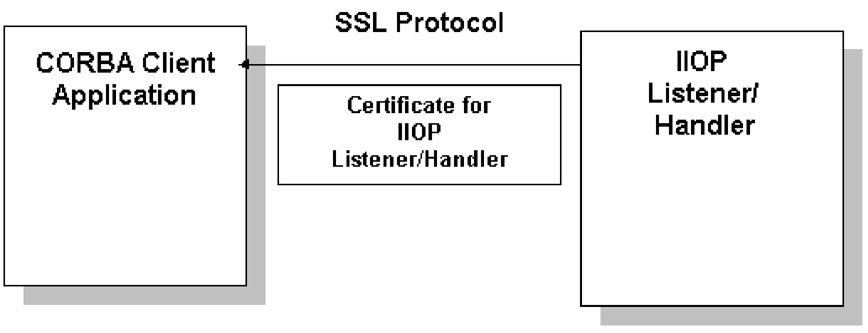
Certificate authentication requires that each side of an SSL connection proves its identity to the other side of the connection. In the CORBA security environment, the IIOP Listener/Handler presents its digital certificate to the principal who initiated the SSL connection. The initiator then provides 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 CORBA security environment 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 BEA Tuxedo 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 3-5 provides a conceptual overview of the certificate authentication.

Figure 3-5 Certificate Authentication



How Certificate Authentication Works

Certificate authentication works in the following manner:

1. The initiating application accesses the BEA Tuxedo domain in one of the following ways:
   - Through the CORBA INS Bootstrapping mechanism. Use this mechanism if you are using a client ORB from another vendor. For more information about

using CORBA INS, see *CORBA Programming Reference* in the BEA Tuxedo online documentation.

- The BEA Bootstrapping mechanism. Use this mechanism if you are using the BEA client ORB.

2. 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 authentication is to be used.

3. 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.

   The IIOP Listener/Handler receives and validates the application's digital certificate as part of the authentication process.

4. If the verification succeeds, the BEA Tuxedo system constructs a `Credentials` object. The `Credentials` object for the principal represents the security context for the current thread of execution.

5. The initiating application invokes a CORBA object in the BEA Tuxedo domain using an object reference.

6. 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.

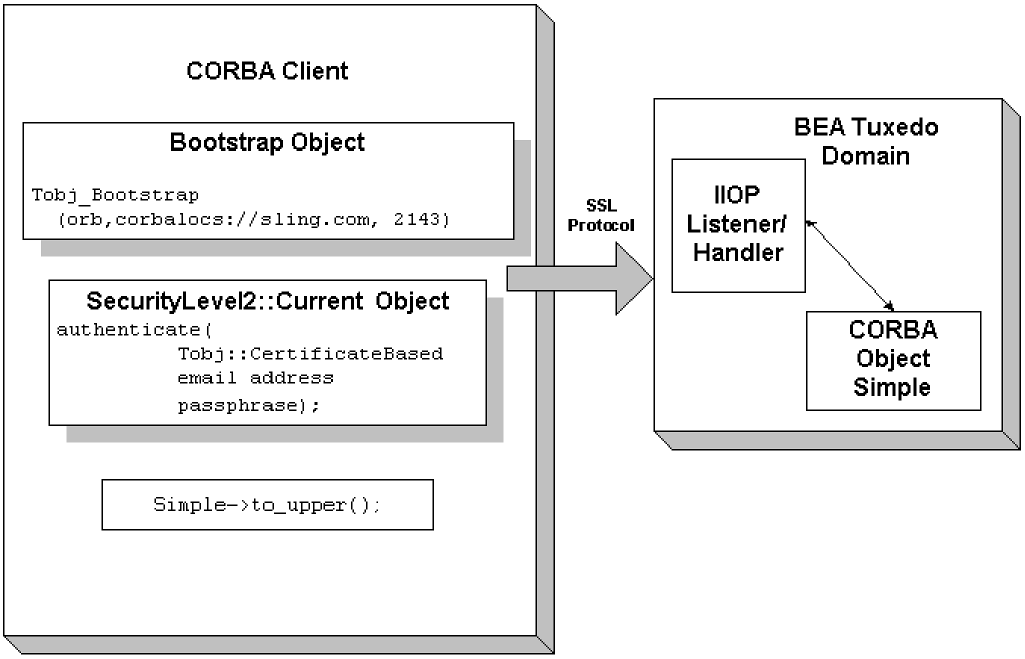
7. The request is digitally signed and encrypted before it is sent to the IIOP Listener/Handler. The BEA Tuxedo system performs the signing and sealing of requests.

8. The IIOP Listener/Handler receives the request from the initiating application. The request is decrypted.

9. The IIOP Listener/Handler retrieves the e-mail component of the subjectDN of the principal's and uses that as the identity of the user.

10. The IIOP Listener/Handler forwards the request, along with the associated tokens of the principal, to the appropriate CORBA object.

Figure 3-6 How Certificate Authentication Works



Development Process for Certificate Authentication

To use certificate authentication in a CORBA application, you need to install a license that enables the use of the SSL protocol and PKI. For information about installing the license, see *Installing the BEA Tuxedo System*.

Using certificate authentication in a CORBA application includes administration and programming steps. Table 3-5 and Table 3-6 list the administration and programming steps for certificate 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 3-5 Administration Steps for Certificate Authentication

Step	Description
1	Set up an LDAP-enabled directory service. You will be prompted for the name of the LDAP server during the installation of the BEA Tuxedo product.
2	Install the license for the SSL protocol.
3	Obtain a digital certificate and private key for the IIOP Listener/Handler from a certificate authority.
4	Obtain digital certificates and private keys for the CORBA client applications from a certificate authority.
5	Store the private key files for the CORBA client applications and the IIOP Listener/Handler in the Home directory of the user or in `$TUXDIR/udataobj/security/keys`.
6	Publish the digital certificates for the IIOP Listener/Handler, the CORBA applications, and the certificate authority in the LDAP-enabled directory service.
7	Define the `SEC_PRINCIPAL_NAME`, `SEC_PRINCIPAL_LOCATION`, and `SEC_PRINCIPAL_PASSVAR` for the ISL server process in the `UBBCONFIG` file.
8	Set the `SECURITY` parameter in the `UBBCONFIG` file to `USER_AUTH`, `ACL`, or `MANDATORY_ACL`.
9	Configure the Authentication Server (`AUTHSRV`) in the `UBBCONFIG` file.
10	Use the `tpusradd` and `tpgrpadd` commands to define the authorized Users and Groups of your CORBA application.
11	Define a port for SSL communication on the IIOP Listener/Handler using the `-S` option of the ISL command.
12	Enable certificate authentication in the IIOP Listener/Handler using the `-a` option of the ISL command.

Table 3-5 Administration Steps for Certificate Authentication (Continued)

Step	Description
13	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 CORBA client application.
13	Use the `tmloadcf` command to load the `UBBCONFIG` file. You will be prompted for the password of the IIOP Listener/Handler defined in the `SEC_PRINCIPAL_NAME` parameter.
14	Optionally, create a Peer Rules file (`peer_val.rul`) for both the CORBA 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 3-7 illustrates the configuration of a CORBA application that uses certificate authentication.

Figure 3-7 Configuration for Using Certificate Authentication in a CORBA Application

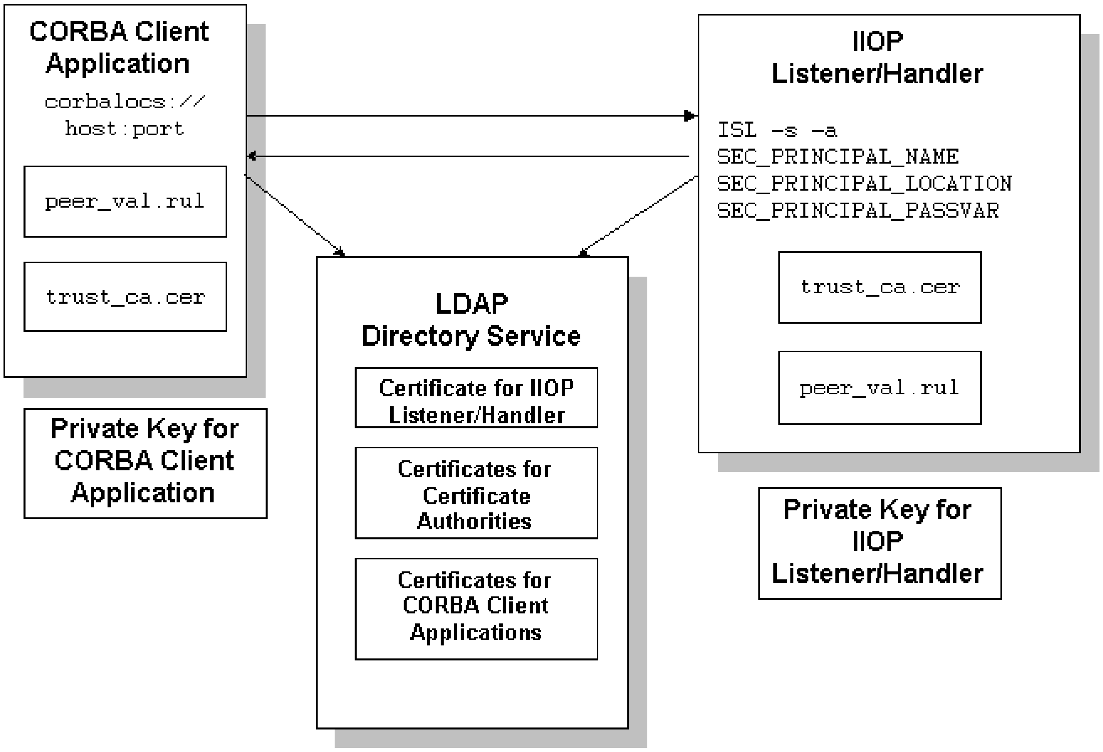


Table 3-6 lists the programming steps for using certificate authentication in a CORBA application. For more information, see "Writing a CORBA Application That Implements Security" on page 10-1.

Table 3-6 Programming Steps for Certificate Authentication

Step	Description
1	Write application code that uses the `corbaloc` or `corbalocs` URL address formats of the Bootstrap object. Note that the CommonName 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 "Using the Bootstrapping Mechanism" on page 10-1. You can also use the CORBA INS bootstrap mechanism to object a reference to a PrincipalAuthenticator object in the BEA Tuxedo domain. For more information about using CORBA INS, see the *CORBA Programming Reference*.
2	Write application code that uses the `authenticate()` method of the `SecurityLevel2::PrincipalAuthenticator` interface to perform authentication. Specify `Tobj::CertificateBased` for the method argument and the pass phrase for the private key as the `auth_data` argument for `Security::Opaque`.

Using an Authentication Plug-in

The BEA Tuxedo product allows the integration of authentication plug-ins into a CORBA application. The BEA Tuxedo 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 BEA Tuxedo 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 CORBA applications. Specifically, an administrator can use authorization to allow or disallow principals to use resources or services provided by a CORBA application.

The CORBA security environment 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.

If you chose to use an authorization plug-in, you must configure the authorization plug-in the registry of the BEA Tuxedo 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 CORBA 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 `ULOG` 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 `ULOG` file, see *Setting Up a BEA Tuxedo Application*.

You can enhance the auditing capabilities of your CORBA application by using an auditing plug-in. The BEA Tuxedo 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 CORBA 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 BEA Tuxedo 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 BEA Tuxedo 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 the registry of the BEA Tuxedo 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.

Single Sign-on

Single sign-on allows authenticated WebLogic Server Users in a WebLogic Server security realm to make secure requests on CORBA objects in a BEA Tuxedo 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 CORBA environment. The trust relationship of the pool can be established in one of the following ways:

- With password authentication. In this scenario, the WebLogic Server User is authenticated but the request between the WebLogic Server realm and the BEA Tuxedo domain is unprotected.
- With 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 authentication. This is the most secure scenario, however, it requires that both WebLogic Server and the CORBA 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 BEA Tuxedo product provides a PKI environment which includes the SSL protocol and the infrastructure needed to use digital certificates in a CORBA application. 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 CORBA 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.
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.

Table 3-7 PKI Interfaces (Continued)

PKI Interface	Description
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 BEA Tuxedo 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 CORBA Security Features

The following sections answer some of the commonly asked questions about the CORBA security features.

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

The answer is no. If you are using security interfaces from previous versions of the WebLogic Enterprise product in your CORBA application there is no requirement for you to change your CORBA application. You can leave your current security scheme in place and your existing CORBA application will work with CORBA applications built with the BEA Tuxedo 8.0 product.

For example, if your CORBA 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 CORBA 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 additional security features.

Can I Use the SSL Protocol in an Existing CORBA Application?

The answer is yes. You may want to take advantage of the extra security protection provided by the SSL protocol in your existing CORBA application. For example, if you have a CORBA 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 CORBA server application and that they are not being routed to a fake CORBA 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 CORBA 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 bits 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 CORBA 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 CORBA application uses password authentication, you can use that code with the SSL protocol. If your CORBA C++ 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 “Single Sign-on” on page 3-24.

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 Authentication?

You might be ready to migrate your existing CORBA application to use Internet connections between the CORBA application and Web browsers and commercial Web servers. For example, users of your CORBA 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 CORBA 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 accidently or maliciously become a $5000 order.

In these situations, the SSL protocol and certificate authentication offer CORBA applications the maximum level of protection. In addition to the benefits achieved through the use of the SSL protocol, certificate authentication offers CORBA 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 BEA Tuxedo 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 CORBA applications to have an identity in native CORBA server applications.

For more information, see “Single Sign-on” on page 3-24.

Part II Security Adminstration

CHAPTER

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 authentication in your CORBA application.

Requirements for Using Public Key Security

To use the SSL protocol and public key security to protect communication between principals and the BEA Tuxedo domain, you need to install a special license. For information about installing the license, see *Installing the BEA Tuxedo System*.

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

Who Needs Digital Certificates and Private/Private Key Pairs?

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

Any digital certificate that is obtained and used must be issued from a trusted certificate authority defined in the trusted CA file. For more information, see “Defining the Trusted Certificate Authorities” on page 4-7.

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
`*key*.der`	The private key file.
`*request*.pem`	The CSR file which you submit to the certificate authority. It contains the same data as the `.dem` 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 through the Web.

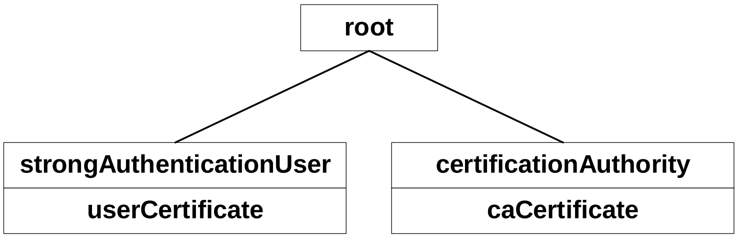
Publishing Certificates in the LDAP Directory Service

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 CORBA security environment in the BEA Tuxedo 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 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 `certificateAuthority` 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 BEA Tuxedo product requires that the digital certificates be stored in the directory service in Privacy Enhanced Mail (PEM) format.

Refer to *Installing the BEA Tuxedo System* for information about integrating an LDAP directory service into the CORBA security environment.

Editing the LDAP Search Filter File

When configuring a CORBA application to use the SSL protocol or certificate 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 BEA Tuxedo product 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 BEA Tuxedo product. For more information, see *Installing the BEA Tuxedo System*.

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 BEA Tuxedo product for the SSL protocol and certificate 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 BEA Tuxedo 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 BEA Tuxedo software looks in the following directories for private key files:

Window 2000

`%HOMEDRIVE%\%HOMEPATH%`

UNIX

`$HOME`

The BEA Tuxedo software also looks in the following directory for private key files:

`$TUXDIR/udataobj/security/keys`

The `$TUXDIR/udataobj/security/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-----
MIICoDAaBgkqhkiG9w0BBQMwDQQItSFrtYcfKygCAQUEggKAEgrMxo8gYB/MOSXG
...
-----END ENCRYPTED PRIVATE KEY-----
```

Defining the Trusted Certificate Authorities

When establishing an SSL connection, the CORBA processes (client applications and the IIOP Listener/Handler) check the identity of the certificate authority and certificates from the peer’s digital certificate chain against a list of trusted certificate authorities to ensure the certificate authority is trusted by the organization. This check is similar to the check done in Web browsers. If the comparison fails, the initiator of the SSL connection refuses to authenticate the target and drops the SSL connection. It is typically the job of the system administrator to define a list of trusted certificate authorities.

Retrieve from the LDAP directory service the digital certificates for the certificate authorities that are to be trusted. Cut and paste the PEM formatted digital certificates into a file named `trust_ca.cer` which is stored in `$TUXDIR/udataobj/security/certs`. The `trust_ca.cer` can be edited with any text editor.

The `trust_ca.cer` 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.

Listing 4-2 provides an example of a Trusted Certificate Authority file.

Listing 4-2 Example of Trusted Certificate Authority File

```
-----BEGIN CERTIFICATE----

MIIEuzCCBCSgAwIBAgIQKtZuM5AOzS9dZaIATJxIuDANBgkqhkiG9w0BAQQFADCB
zDEXMBUGA1UEChMOVmVyaVNpZ24sIEluYy4xHzAdBgNVBAsTFlZlcmlTaWduIFRy
dXN0IE5ldHdvcmsxRjBEBgNVBAsTPXd3dy52ZXJpc2lnbi5jb20vcmVwb3NpdG9y
eS9SUEEgSW5jb3JwLiBCeSBSZWYuLExJQUIuTFREKGMpOTgxSDBGBgNVBAMTP1Zl
cmlTaWduIENsYXNzIDEgQ0EgSW5kaXZpZHVhbCBTdWJzY3JpYmVyLVBlcnNvbmEg
...
-----END CERTIFICATE-----

-----BEGIN CERTIFICATE----

MIIEuzCCBCSgAwIBAgIQKtZuM5AOzS9dZaIATJxIuDANBgkqhkiG9w0BAQQFADCB
zDEXMBUGA1UEChMOVmVyaVNpZ24sIEluYy4xHzAdBgNVBAsTFlZlcmlTaWduIFRy
dXN0IE5ldHdvcmsxRjBEBgNVBAsTPXd3dy52ZXJpc2lnbi5jb20vcmVwb3NpdG9y
...
-----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 BEA Tuxedo 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 BEA Tuxedo 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

Table 4-1 Supported Keys for Peer Rules File (Continued)

Key	Attribute
`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 semicolon (`;`) 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 BEA Tuxedo product verifies peer information against the Peer Rules file. If you do not want to perform this check, create an empty Peer Rules file.

CHAPTER

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 CORBA Application Links

Understanding min and max Values

Before you can configure LLE for your CORBA 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 Domestic 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 CORBA 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 BEA Tuxedo 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 Domestic 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
```

BEA Tuxedo license files are located in the following directories:

Windows 2000

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

UNIX

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

Configuring LLE on CORBA Application Links

To configure LLE in CORBA applications, you need to set the `MINENCRYPTBITS` and `MAXENCRYPTBITS` parameters in the `UBBCONFIG` file for each CORBA 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.

CHAPTER

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 authentication with the IIOP Listener/Handler or the CORBA C++ object request broker (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 or the command-line options of the CORBA C++ 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++ ORB, use the `-ORBsecurePort` command-line option 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 when using callback objects or the CORBA 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 license for the SSL protocol to be installed. If the `-S` option or the `-ORBsecurePort` command-line option is executed and a license to enable the use of the SSL protocol does not exist, the IIOP Listener/Handler or CORBA C++ 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 in enabled in the IIOP Listener/Handler and the CORBA C++ ORB. 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++ ORB, specify the `–ORBpeerValidate` command-line option.

The values for the `-v` option and the `-ORBpeerValidate` command-line option 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 BEA Tuxedo 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 on the ORB to set the encryption strength in the CORBA C++ ORB.

The `-z` option and the `-ORBminCrypto` command-line option set the minimum level of encryption used when an application establishes an SSL connection with the IIOP Listener/Handler or the CORBA C++ ORB. The valid values are 0, 40, 56, and 128. A value of 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 set the maximum level of encryption used when an application establishes an SSL connection with the IIOP Listener/Handler or the CORBA C++ 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 are available only if the license for the SSL protocol is installed.

To change the strength of encryption currently used in a CORBA 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

`-z` `-ORBminCrypto`	`-Z` `-ORBmaxCrypto`	**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 `ORBminCrypto` and `ORBmaxCrypto`, 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.
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.

Table 6-1 Combinations of Encryption Values (Continued)

`-z` `-ORBminCrypto`	`-Z` `-ORBmaxCrypto`	**Description**
No value specified	40, 56, 12	Privacy protection can be negotiated to the value specified by the `-Z` option as long as it is less than the maximum allowed by the SSL license. The `-z` option defaults to 40.
40, 56, 128	40, 56, 128	Privacy protection can be negotiated between the values specified by the `-z` option up to the value specified by the `-Z` 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 CORBA security environment, 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 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_LOCATION
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 security parameters and reboot the IIOP 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 BEA Tuxedo 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 CORBA application, and get authorization and auditing tokens for remote client applications.

Because the IIOP Listener/Handler must authenticate its own identity to the BEA Tuxedo 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 7-2 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 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
```

CHAPTER

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 Password Authentication
- Sample UBBCONFIG File for Password Authentication
- Configuring Certificate Authentication
- Sample UBBCONFIG File for Certificate 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 BEA Tuxedo system uses a default authentication server called AUTHSRV 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 CORBA 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 CORBA 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 CORBA 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 BEA Tuxedo 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"
```

Once you configure the default authentication server, the identity of the IIOP Listener/Handler (as specified in the `SEC_PRINCIPAL_NAME` parameter in the `UBBCONFIG` file) must be specified in the `tpusr` file. In addition, all the users of the CORBA application must be specified in the `tpusr` file. For more information, see “Defining Authorized Users” on page 7-3.

Defining Authorized Users

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

Authorized users can be defined in the following ways:

- When using password authentication, authorized users are defined using a username and an associated password.

- When using certificate 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 CORBA application.

You use the `tpusradd` command to create files containing lists of authorized principals. The `tpusradd` command adds a new principal entry to the BEA Tuxedo 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 CORBA 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 CORBA 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.

- `-g` *`gid`*

  The group identification number. The GID can be an integer identifier or character-string name. This option defines the new user's group membership. It defaults to the `other` group (identifier 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 CORBA application.

- `usrname`

  A string of printable characters that specifies the new login name of the user. The name may not contain a colon (:). pound sign (#), or a newline (\n). The user name must be unique within the list of existing users for the CORBA application

If you are using the default authentication server, the identity of the IIOP Listener/Handler (as specified in the `SEC_PRINCIPAL_NAME` parameter in the `UBBCONFIG` file) must be specified in the `tpusr` file. In addition, all the users of the CORBA application must be specified in the `tpusr` file.

If you are using a custom authentication service, define the IIOP Listener/Handler and the users of the CORBA application in the user registry of the custom authentication service. In addition, no file called `tpusr` should appear in `$APPDIR`. If a file by that name exists, a `CORBA/NO_PERMISSION` exception will be raised.

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

Listing 7-2 Sample tpusr File

```
Usrname   Cltname   Password Entry   Uid    GID

milozzi   “bar”              2        100    0
smart     “ “                1        1      0
pat       “tpsysadmin”       3        0      8192
butler     “tpsysadmin”       3        N/A    8192
```

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

In addition to the `tpusradd` and `tpgrpadd` commands, the BEA Tuxedo 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 *BEA Tuxedo Command Reference* in the BEA Tuxedo 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 CORBA application, but only after converting them to the format required by the BEA Tuxedo system. To convert your files, run the `tpaclcvt` 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:

   ```
   tpaclcvt -u /etc/password
   ```

   This command creates the `tpusr` file and stores the converted data in it. If the `tpusr` file already exists, `tpaclcvt` 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 BEA Tuxedo system, enter the following command:

   ```
   tpaclcvt -g /etc/group
   ```

   This command creates the `tpgrp` file and stores the converted data in it. If the `tpgrp` file already exists, `tpaclcvt` 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 CORBA 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 CORBA application. Tobj::PrincipalAuthenticator::get_auth_type() returns a value of TOBJ_NOAUTH.
APP_PW	Indicates that client applications are required to supply an application password to access the BEA Tuxedo domain. The tmloadcf command prompts for an application password. Tobj::PrincipalAuthenticator::get_auth_type() returns a value of TOBJ_SYSAUTH.
USER_AUTH	Indicates that client applications and the IIOP Listener/Handler are required to authenticate themselves to the BEA Tuxedo domain using a password. The value USER_AUTH is similar to APP_PW but, in addition, indicates that user authentication will be done during client initialization. The tmloadcf command prompts for an application password. Tobj::PrincipalAuthenticator::get_auth_type() returns a value of TOBJ_APPAUTH. No access control checking is performed at this security level.
ACL	Indicates that authentication is used in the CORBA 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 tmloadcf command prompts for an application password. Tobj::PrincipalAuthenticator::get_auth_type returns a value of TOBJ_APPAUTH.
MANDATORY_ACL	Indicates that authentication is used in the CORBA application and access control checks are performed on interfaces, services, queue names, and event names. The value MANDATORY_ACL is similar to ACL, but permission is denied if an associated ACL is not found for the name.The tmloadcf command prompts for an application password. Tobj::PrincipalAuthenticator::get_auth_type returns a value of TOBJ_APPAUTH.

Note: If the IIOP Listener/Handler is configured for using certificate 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 offline means such as telephone or letter.

Configuring Password Authentication

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 CORBA 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 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 offline means such as telephone or letter.

Sample UBBCONFIG File for Password Authentication

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

Listing 7-3 Sample UBBCONFIG File for Password Authentication

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

*MACHINES
    "ICEAXE"
    LMID        = SITE1
    APPDIR      = "D:\TUXDIR\samples\corba\SECURAPP"
    TUXCONFIG   = "D:\TUXDIR\samples\corba\SECURAPP\results
                     \tuxconfig"
    TUXDIR      = "D:\Tux8"
    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 Authentication

Certificate authentication uses the SSL protocol so you need to install the license for the SSL protocol and configure the SSL protocol before you can use certificate authentication. Information about installing the license for the SSL protocol can be found in *Installing the BEA Tuxedo System*. 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 authentication in a CORBA 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 CORBA application. For more information, see “Managing Public Key Security” on page 4-1.

To enable certificate authentication, complete the following steps:

1. Install the license for the SSL protocol.

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 CORBA application from a certificate authority.

5. Store the private keys for the CORBA application in the `Home` directory of the user or in the following directories:

   Windows 2000

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

   UNIX

   ```
   $TUXDIR/udataobj/security/keys
   ```

6. Publish the certificates for the IIOP Listener/Handler, the CORBA 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 CORBA 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 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 CORBA application. For more information, see "Defining the Trusted Certificate Authorities" on page 4-7.

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 CORBA 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 authentication, complete one of the following:

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

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

Sample UBBCONFIG File for Certificate Authentication

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

Listing 7-4 Sample UBBCONFIG File for Certificate Authentication

```
*RESOURCES
    IPCKEY    55432
    DOMAINID  simpapp
    MASTER    SITE1
    MODEL     SHM
    LDBAL     N
    SECURITY  USER_AUTH
    AUTHSVR “AUTHSVC”


*MACHINES
    "ICEAXE"
    LMID        = SITE1
    APPDIR      = "D:\TUXDIR\samples\corba\SIMPAP~1"
    TUXCONFIG   = "D:\TUXDIR\samples\corba\SIMPAP~1
                  \results\tuxconfig"
    TUXDIR      = "D:\TUX8"
    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 -- -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 the CORBA security environment 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.

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 CORBA 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 offline 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 CORBA 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 offline means such as telephone or letter.

Setting ACL Policy Between CORBA Applications

As the administrator, you use the following configuration parameters to set and control the access control list (ACL) policy between CORBA applications that reside in different BEA Tuxedo 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 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 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 Listing 7-5, 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.

Listing 7-5 Sample DMCONFIG File for ACL Policy

```
*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 erver applications in a BEA Tuxedo 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 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 on the CORBA object request broker (ORB).

By setting the `CLOPT -t` option or specifying the `-ORBinterOP` command-line option, you are lowering the effective level of security for a CORBA 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 7-6 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++ ORB, specify the `-ORBinterOp` command-line option 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.

CHAPTER

8 Configuring Single Sign-on

This topic includes the following sections:

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

Single Sign-on with Password Authentication

The steps for implementing single sign-on with 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 BEA Tuxedo domain used with your CORBA application. For more information about the different address formats supported in CORBA applications, see "Writing a CORBA Application that Implements Security" on page 10-1.

- `username` as the name of the WebLogic Server User.
- `userpassword` as the password for the WebLogic Server User
- `apppassword` as the password of the CORBA application you want to access.
- `securitycontext` as `Yes`. Yes indicates that you want the security context of the WebLogic Server User passed to the BEA Tuxedo 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 online documentation.

2. Use the `tpusradd` command to define the WebLogic Server User as an authorized user in the BEA Tuxedo domain. The username and password for the WebLogic Server User 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 security 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 BEA Tuxedo domain.
4. Set the `SECURITY` parameter in the `UBBCONFIG` file to `USER_AUTH` or higher.

Single Sign-on with Password Authentication and the SSL Protocol

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

1. Configure the SSL protocol in the WebLogic Server and the BEA Tuxedo CORBA environments.

   For information about configuring the SSL protocol in the WebLogic Server environment, see *Managing Security* in the WebLogic Server online documentation.

   For information about configuring the SSL protocol in the CORBA environment, see “Single Sign-on” on page 3-24.

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 BEA Tuxedo domain you want to access. For more information about the different address formats supported in CORBA applications, see “Using the Bootstapping Mechanism” on page 10-1.

   - `username` as the name of the WebLogic Server User.

   - `userpassword` as the password for the WebLogic Server User.

   - `apppassword` as the password of the CORBA application you want to access.

   - `securitycontext` as `Yes`. Yes indicates that you want the security context of the WebLogic Server User passed to the BEA Tuxedo 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 BEA Tuxedo CORBA 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 online documentation.

3. Use the `tpusradd` command to define the WebLogic Server User as an authorized user in the BEA Tuxedo domain. The username and password for the WebLogic Server User 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 security 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 BEA Tuxedo domain.

5. Set the `SECURITY` parameter in the `UBBCONFIG` file to `USER_AUTH` or higher.

Single Sign-on with the SSL Protocol and Certificate Authentication

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

1. Configure the SSL protocol in the WebLogic Server and the BEA Tuxedo CORBA environments.

   For information about configuring the SSL protocol in the WebLogic Server environment, see *Managing Security* in the WebLogic Server online documentation.

   For information about configuring the SSL protocol in the BEA Tuxedo CORBA environment, see “Single Sign-on” on page 3-24.

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 BEA Tuxedo domain you want to access.

   - `username` as the e-mail address of the subject of the digital certificate.

- `userpassword` as the private key of the digital certificate.
- `apppassword` as the password of the CORBA application you want to access.
- `securitycontext` as `Yes`. Yes indicates that you want the security context of the WebLogic Server User passed to the BEA Tuxedo domain.
- `minencryptionlevel` and `maxecrptionlevel`. 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 BEA Tuxedo CORBA environments.
- `certificatebasedauth` as `Yes`. Yes indicates that certificate 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 online documentation.

3. Use the `tpusradd` command to define the WebLogic Server User as an authorized user in the BEA Tuxedo domain. The username and password for the WebLogic Server User 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 security 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 BEA Tuxedo domain.
5. Set the -a option of the ISL command to configure the IIOP Listener/Handler to enable certificate authentication.The ISL command for the IIOP Listener/Handler is defined for the CLOPT parameter in the `UBBCONFIG` file for the BEA Tuxedo domain.
6. Set the `SECURITY` parameter in the `UBBCONFIG` file to `USER_AUTH` or higher.

Using certificate authentication between the WebLogic Server environment and the BEA Tuxedo CORBA environment implies performing a new SSL handshake to establish a connection from the WebLogic Server environment to a CORBA object in the BEA Tuxedo CORBA environment. In order to support multiple client requests over the same SSL network connection, certificate authentication must be set up as follows:

- Obtain a digital certificate for the WebLogic Enterprise Connectivity process. This digital certificate is presented to the BEA Tuxedo CORBA 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 BEA Tuxedo environment remains.
- When a client request is made from the WebLogic Server environment on a CORBA object in the BEA Tuxedo CORBA environment, digital certificates are exchanged between the 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 the BEA Tuxedo CORBA environment that has the sessions keys that were created when the SSL connection was established between the environments. The WebLogic Enterprise Connectivity process then forwards the client request using the established SSL connection to the BEA Tuxedo environment.

CHAPTER

9 Configuring Security Plug-ins

This topic includes the Registering the Security Plug-ins (SPIs) section.

Registering the Security Plug-ins (SPIs)

The CORBA and ATMI environments in the BEA Tuxedo product use a common transaction processing (TP) infrastructure that consists of a set of core services, such as security. The TP infrastructure is available to CORBA 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 BEA Tuxedo system. The registry of the BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo installation. Every client application, server application, and server machine in a particular CORBA application must use the same set of security plug-ins.

The registry is located in the following directory:

Windows 2000

```
$TUXDIR\udataobj
```

UNIX

```
$TUXDIR/udataobj
```

The system administrator of a CORBA application in which custom security plug-ins are used is responsible for registering those plug-ins. A system administer can register security plug-ins in the registry of the BEA Tuxedo 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:

- `epifreg`—for registering a security plug-in
- `epifunreg`—for unregistering a security plug-in
- `epifregedt`—for editing registry information

Instructions for using these commands are available in *Developing Security Services for ATMI and CORBA Environments*. (This document contains the specifications for the Security SPIs, and describes the BEA Tuxedo 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 BEA Tuxedo system in order to access the customer security plug-ins.

Part III Security Programming

CHAPTER

10 Writing a CORBA Application That Implements Security

This topic includes the following sections:

- Using the Bootstrapping Mechanism
- Using Password Authentication
- Using Certificate Authentication
- Using the Interoperable Naming Service Mechanism
- Using the Invocations_Options_Required() Method

Using the Bootstrapping Mechanism

Note: This mechanism should be used with the BEA CORBA client applications.

The Bootstrap object in the BEA Tuxedo CORBA environment 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 password authentication 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 `authenticate()` method of the `SecurityLevel2::PrincipalAuthenticator` interface and the `invocation_options_required()` method of the `SecurityLeve12::Credentials` interface to specify that certificate authentication is to be used.
`corbalocs`	Invocations on the IIOP Listener/Handler are protected and the IIOP Listener/Handler or the CORBA C++ 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 BEA Tuxedo 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 BEA Tuxedo 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 CORBA 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 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 CORBA 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 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 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 but not configured for certificate authentication and the `corbaloc` URL address format is used, the invocation on the specified CORBA object results in an `INVALID_DOMAIN` exception.

BEA recommends that existing CORBA 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 or the CORBA C++ 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 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 or the CORBA C++ ORB must be configured to enable the use of the SSL protocol. For more information about configuring the IIOP Listener/Handler or the CORBA C++ ORB for the SSL protocol, see “Configuring the SSL Protocol” on page 6-1.

Using Password Authentication

This section describes implementing password authentication in a CORBA applications.

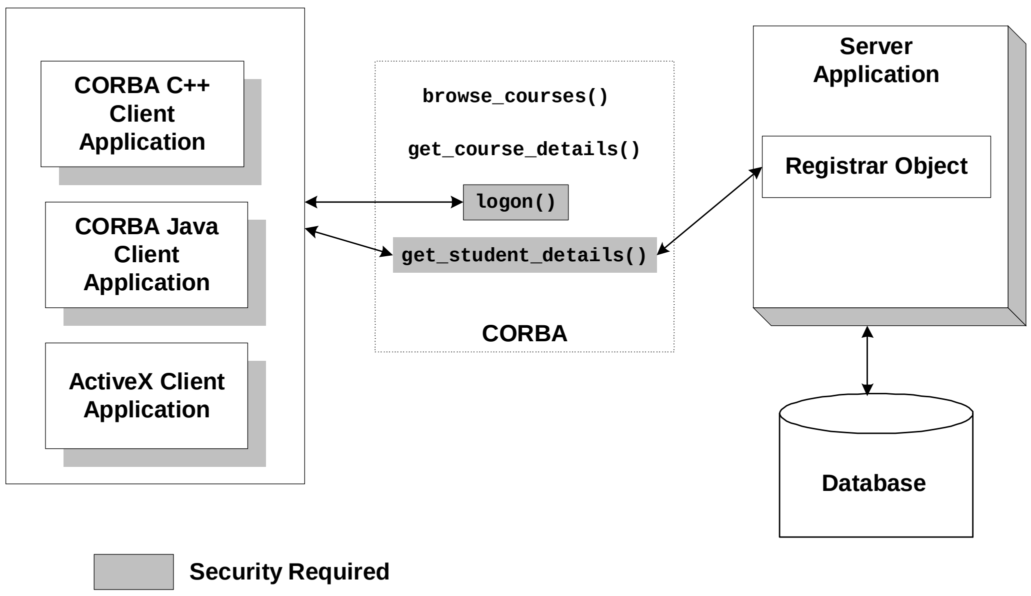
The Security Sample Application

The Security sample application demonstrates 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 BEA Tuxedo software. For information about building and running the Security sample application, see the *Guide to the CORBA University Sample Applications*.

Writing the Client Application

When using 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 BEA Tuxedo 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 CORBA environment extends the PrincipalAuthenticator object with functionality to support similar security to that found in the ATMI environment in the BEA Tuxedo product. 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 BEA Tuxedo extensions when developing a CORBA 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, CORBA 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.

Note: The user data specified as part of the logon cannot contain embedded NULLs.

The following sections contain C++ and Java code examples that illustrate implementing password authentication. For a Visual Basic code example, see "Automation Security Reference" on page 17-1.

C++ Code Example That Uses the SecurityLevel2::PrincipalAuthenticator::authenticate() Method

Listing 10-3 contains C++ code that performs 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 =
     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’};

Tobj::PrincipalAuthenticator_ptr var_bea_principal_authenticator =
   Tobj::PrincipalAuthenticator::_narrow(var_bea_principal_authenticator.in());

//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 That Uses the SecurityLevel2.PrincipalAuthenticator.authenticate() Method

Listing 10-4 contains Java code that performs 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 That Uses the Tobj::PrincipalAuthenticator::logon() Method

Listing 10-5 contains C++ code that performs 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 =
     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;

//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 That Uses the Tobj.PrincipalAuthenticator.logon() Method

Listing 10-6 contains Java code that performs 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 Authentication

This section describes implementing certificate authentication in CORBA 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 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.

The Simpapp sample application was modified in the following ways to support certificate 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 IIOP Listener/Handler.
- The code for the CORBA client application uses the `corbalocs` URL address format.
- The code for the CORBA 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++ Secure Simpapp sample application are located in the `\samples\corba\simpappSSL` directory of the BEA Tuxedo 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 CORBA Client Application

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

1. Uses the Bootstrap object to obtain a reference to the SecurityCurrent object for the specific BEA Tuxedo 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 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 authentication.

C++ Code Example of Certificate Authentication

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

Listing 10-7 CORBA C++ Client Application That Uses Certificate 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_varprivileges;
        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 =
                           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 Authentication

Listing 10-8 illustrates using certificate authentication in a CORBA Java client application.

Listing 10-8 CORBA Java Client Application That Uses Certificate 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 ocurr =
         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 Interoperable Naming Service Mechanism

Note: This mechanism should be used with third-party client ORBs.

To use the Interoperable Naming Service mechanism to access the BEA Tuxedo domain with the proper credentials, perform the following steps:

1. Use the `ORB::resolve_initial_references()` operation to get a `SecurityLevel2::PrincipalAuthenticator` object for the BEA Tuxedo domain. The `SecurityLevel2::PrincipalAuthenticator` object adheres to the standard CORBAservices Security Service instead of the proprietary BEA delegated interfaces and contains methods for the purpose of authenticating principals.

2. Use the `authenticate()` method of the `SecurityLevel2::PrincipalAuthenticator` object to log on to the BEA Tuxedo domain and authenticate the client ORB to the BEA Tuxedo domain. If security credentials are required to access the BEA Tuxedo domain, the `authenticate()` method will return a status indicating that continued authentication is required.

3. Use the `continue_authentication()` method of the `SecurityLevel2::PrincipalAuthenticator` object to pass encyrpted logon and credential information to the BEA Tuxedo domain.

For more information about using the CORBA Interoperable Naming Service (INS) mechanism, see the *CORBA Bootstrap Object Programming Reference* for the `SecurityLevel2::PrincipalAuthenticator` interface.

Using the Invocations_Options_Required() Method

When using certificate 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 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 15-1 and "Java Security Reference" on page 16-1 for a complete list of security options.

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

Listing 10-9 C++ Example That Uses 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 =
                           seccur->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::EstalishTrustInClient|
                               Security::SimpleDelegation);

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

```
                    }
        ...
```

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

Listing 10-10 Java Example That Uses 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");

      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 ocurr =
         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);
         }
       ...
```

CHAPTER

11 Building and Running the CORBA Sample Applications

The topic includes 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 password authentication. For instructions for building and running the Security sample application, see the *Guide to the CORBA University Sample Applications*.

Building and Running the Secure Simpapp Sample Application

The Secure Simpapp sample application demonstrates using the SSL protocol and certificate authentication to protect communications between client applications and the BEA Tuxedo domain.

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 files for the Secure Simpapp sample application are located in the following directories:

Windows 2000

`drive:\TUXdir\samples\corba\simpappSSL`

UNIX

`/usr/local/TUXdir/samples/corba/simpappSSL`

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

Table 11-1 Files Included in the Secure Simpapp Sample Application

File	Description
`Simple.idl`	The OMG IDL code that declares the `Simple` and `SimpleFactory` interfaces.
`Simples.cpp`	The C++ source code that overrides the default `Server::initialize` and `Server::release` methods.
`Simplec.cpp`	The source code for the CORBA C++ client application in the Secure Simpapp sample application.
`Simple_i.cpp`	The C++ source code that implements the `Simple` and `SimpleFactory` methods.
`Simple_i.h`	The C++ header file that defines the implementation of the `Simple` and `SimpleFactory` methods.
`SimpleClient.java`	The Java source code for the client application in the Secure Simpapp sample application.

Table 11-1 Files Included in the Secure Simpapp Sample Application

File	Description
`Readme.html`	This file provides the latest information about building and running the Secure Simpapp sample application.
`runme.cmd`	The Windows 2000 batch file that builds and runs the Secure Simpapp sample application.
`runme.ksh`	The UNIX Korn shell script that builds and executes the Secure Simpapp sample application.
`makefile.mk`	The makefile for 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 `Readme.html` file for information about manually building the Secure Simpapp sample application. The UNIX `make` command needs to be in the path of your machine.
`makefiles.nt`	The makefile for the Secure Simpapp sample application on the Windows 2000 operating system. This makefile can be used directly by the Visual C++ `nmake` command. This file is used to manually build the Secure Simpapp sample application. Refer to the `Readme.html` file for information about manually building the Secure Simpapp sample application. The Windows 2000 `nmake` 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 BEA Tuxedo 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 2000

```
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 11-2 lists the environment variables required to run the Secure Simpapp sample application.

Table 11-2 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 2000
	`APPDIR=c:\`*`work`*`\simpappSSL`
	UNIX
	`APPDIR=/usr/`*`work`*`/simpappSSL`

`TUXCONFIG`	The directory path and name of the configuration file. For example:
	Windows 2000
	`TUXCONFIG=c:\work\simpappSSL\tuxconfig`
	UNIX
	`TUXCONFIG=/usr/`*`work`*`/simpappSSL/tuxconfig`

Table 11-2 Required Environment Variables for the Secure Simpapp Sample Application

Environment Variable	Description
`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 2000** `TOBJADDR=trixie::1111` **UNIX** `TOBJADDR=trixie::1111`

`JAVA_HOME`	The directory path where you installed the JDK software. For example:
	Windows 2000
	`JAVA_HOME=c:\JDK1.2`
	UNIX
	`JAVA_HOME=/usr/local/JDK1.2`
	If `JAVA_HOME` is not defined the sample only uses CORBA C++ client application.

`RESULTSDIR`	A subdirectory of `APPDIR` where files that are created as a result of executing the `runme` command are stored. For example:
	Windows 2000
	`RESULTSDIR=c:\workdirectory\`
	UNIX
	`RESULTSDIR=/usr/local/workdirectory/`

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

Windows 2000

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` (for the CORBA Java client application)

To change the settings, perform the following steps:

Windows 2000

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`* (for the CORBA Java client application)

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 2000

```
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 11-3 lists the C++ files in the work directory generated by the `runme` command.

Table 11-3 C++ Files Generated by the runme Command

File	Description
`Simple_c.cpp`	Generated by the `idl` command, this file contains the client stubs for the `SimpleFactory` and `Simple` interfaces.

Table 11-3 C++ Files Generated by the runme Command (Continued)

File	Description
`Simple_c.h`	Generated by the `idl` command, this file contains the client definitions of the `SimpleFactory` and `Simple` interfaces.
`Simple_s.cpp`	Generated by the `idl` command, this file contains the server skeletons for the `SimpleFactory` and `Simple` interfaces.
`Simple_s.h`	Generated by the `idl` command, this file contains the server definition for the `SimpleFactory` and `Simple` interfaces.

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

Table 11-4 Java Files Generated by the runme Command

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

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

File	Description
`Simple.java`	Generated by the `idltojava` command for the `Simple` interface. The `Simple` interface contains the Java version of the OMG IDL interface. It extends `org.omg.CORBA.Object`.
`SimpleHolder.java`	Generated by the `idltojava` command for the `Simple` interface.This class holds a public instance member of type `Simple`. The class provides operations for `out` and `inout` arguments that CORBA has but that do not match exactly to Java.
`SimpleHelper.java`	Generated by the `idltojava` command for the `Simple` interface. This class provides auxiliary functionality, notably the `narrow` method.
`_SimpleStub.java`	Generated by the `idltojava` command for the `Simple` interface. This class is the client stub that implements the `Simple.java` interface.

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

Table 11-5 Files in the results Directory Generated by the runme Command

File	Description
`input`	Contains the input that the `runme` command provides to the CORBA client application.
`output`	Contains the output produced when the `runme` command executes the CORBA client application.
`expected_output`	Contains the output that is expected when the CORBA client application is executed by the `runme` command. The data in the `output` file is compared to the data in the `expected_output` file to determine whether or not the test passed or failed.

Table 11-5 Files in the results Directory Generated by the runme Command

File	Description
`log`	Contains the output generated by the `runme` command. If the `runme` command fails, check this file for errors.
`setenv.cmd`	Contains the commands to set the environment variables needed to build and run the Secure Simpapp sample application on the Windows 2000 operating system platform.
`stderr`	Generated by the `tmboot` command, which is executed by the `runme` command.
`stdout`	Generated by the `tmboot` command, which is executed by the `runme` command.
`tmsysevt.dat`	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the `tmboot` command in the `runme` command.
`tuxconfig`	A binary version of the `UBBCONFIG` file.
`ULOG.<date>`	A log file that contains messages generated by the `tmboot` command.

Using the Secure Simpapp Sample Application

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

Windows 2000

```
prompt>tmboot -y
```

UNIX

```
ksh prompt>tmboot -y
```

Run the CORBA C++ client application in the Secure Simpapp sample application as follows:

Windows 2000

```
prompt> set TOBJADDR=corbalocs://host:port
prompt> simple_client -ORBid BEA_IIOP -ORBpeerValidate none
String?
Hello World
HELLO WORLD
hello world
```

UNIX

```
ksh prompt>export TOBJADDR=corbalocs://host:port
ksh prompt>simple_client -ORBid BEA_IIOP -ORBpeerValidate none
String?
Hello World
HELLO WORLD
hello world
```

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

Windows 2000

```
prompt> set CLASSPATH=%TUXDIR%\udataobj\java\jdk\m3envobj.jar;
%TUXDIR%\udataobj\java\jdk\wleclient.jar;.;%CLASSPATH%
java -DTOBJADDR=%TOBJADDR% -Dorg.omg.CORBA.ORBpeerValidate=none
classpath %CLASSPATH% SimpleClient
String?
Hello World
HELLO WORLD
hello world
```

UNIX

```
ksh prompt>export
CLASSPATH=${TUXDIR}/udataobj/java/jdk/m3envobj.jar;
${TUXDIR}/udataobj/java/jdk/wleclient.jar:.:${CLASSPATH}
java -DTOBJADDR=${TOBJADDR} -Dorg.omg.CORBA.ORBpeerValidate=none
-classpath ${CLASSPATH} SimpleClient
String?
Hello World
HELLO WORLD
hello world
```

Note: The CORBA Java client application in the Secure Simpapp sample CORBA Java client application uses the client-only JAR files `m3envobj.jar` and `wleclient.jar`.

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 2000

```
prompt>tmshutdown -y
```

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

UNIX

```
ksh prompt>tmshutdown -y
```

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

CHAPTER

12 Troubleshooting

This topic includes the following sections:

- Using ULOGS and ORB Tracing
- CORBA::ORB_init Problems
- Password Authentication Problems
- Certificate 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 authentication with 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 2000, 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 2000

```
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.

CORBA::ORB_init Problems

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 security license is enabled. Also, verify that the specified level of encryption does not exceed the encryption level supported by the security license.

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.

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. The values for the properties must be within the range appropriate for the license.

- 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.

Password Authentication Problems

If the client application fails when using the `corbalocs` URL address format with 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-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 `ULOG` file will indicate that a non-GIOP message was received if the IIOP Listener/Handler was configured for the SSL protocol but the CORBA client application used a `TOBJADDR` object without the `corbalocs` prefix to indicate a secure connection.

Certificate Authentication Problems

If the client application fails when using the `corbalocs` URL address format with certificate 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” on page 12-6 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 combination. Verify that the security level for the CORBA application is set to `USER_AUTH` and that the specified 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 12-9 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/Hander 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 security:

- The ORB `-ORBpeerValidate` command-line option 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 CORBA application, you cannot 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 CORBA 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 password authentication with the `corbalocs` URL address format to communicate with the IIOP Listener/Handler. If you try to use certificate 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 BEA 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 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. A problem can occur if any certificate authority in the `trust_ca.cer` file is invalid.
- 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, Microsoft 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 BEA Tuxedo 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

13 CORBA Security APIs

This topic includes the following sections:

- The CORBA Security Model
- Functional Components of the CORBA Security Environment
- The Principal Authenticator Object
- The Credentials Object
- The SecurityCurrent Object

For the C++, Java, and Automation method descriptions for the CORBA Security APIs, see the following topics:

- “C++ Security Reference” on page 15-1
- “Java Security Reference” on page 16-1
- “Automation Security Reference” on page 17-1

The CORBA Security Model

The security model in the CORBA environment of the BEA Tuxedo product defines only a framework for security. The BEA Tuxedo 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 CORBA application.

The security model in the CORBA environment defines:

- Under what conditions client applications may access objects in a BEA Tuxedo domain
- What type of proof material principals are required to authenticate themselves to the BEA Tuxedo domain

The security model in the CORBA environment is a combination of the security model defined in the CORBAservices Security Service specification and the value-added extensions that provide a focused, simplified form of the security model found in the ATMI environment of the BEA Tuxedo product.

The following sections describe the general characteristics of the CORBA 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 CORBA security model provides a simple framework through which a security officer can limit access to the BEA Tuxedo 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 CORBA application. The BEA Tuxedo product provides a set of configuration parameters and utilities. Using the configuration parameters and utilities, a system administrator can configure the CORBA application to force the principals to be authenticated to access a system on which BEA Tuxedo software is installed. To enforce the configuration parameters, the system administrator uses the `tmloadcf` command to update the configuration file for a particular CORBA application.

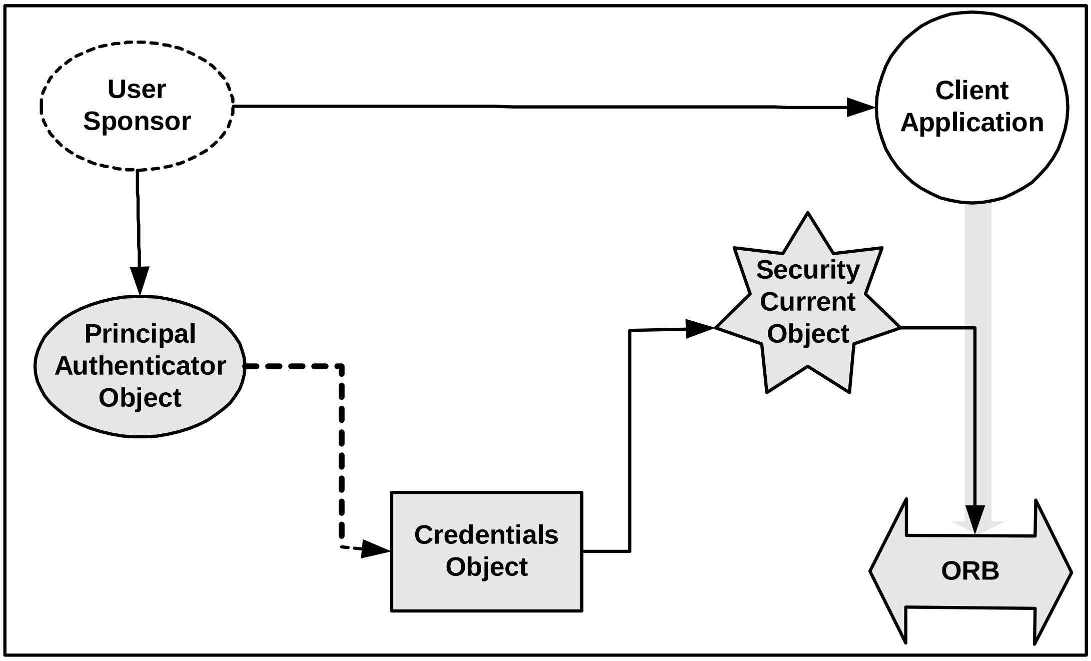
For more information about configuring security for your CORBA application, see “Configuring the SSL Protocol” on page 6-1 and “Configuring Authentication” on page 7-1.

Functional Components of the CORBA Security Environment

The CORBA security model is based on the process of authenticating principals to the BEA Tuxedo domain. The objects in the CORBA security environment 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 13-1 illustrates the authentication process used in the CORBA security model.

Figure 13-1 Authentication Process in the CORBA Security Model



The following sections describe the objects in the CORBA 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 Authentication

The Principal Authenticator object has been enhanced to support certificate authentication. The use of certificate authentication is controlled by specifying the `Security::AuthenticationMethod` value of `Tobj::CertificateBased` as a parameter to the `PrincipalAuthenticator::authenticate` operation. When certificate 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.

BEA Tuxedo Extensions to the Principal Authenticator Object

The CORBA environment in the BEA Tuxedo product extends the Principal Authenticator object to support a security mechanism similar to the security in the ATMI environment in the BEA Tuxedo product. The enhanced functionality is provided by defining the `Tobj::PrincipalAuthenticator` interface. This interface contains methods to provide similar capability to that available from the ATMI environment 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 CORBAservices 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.

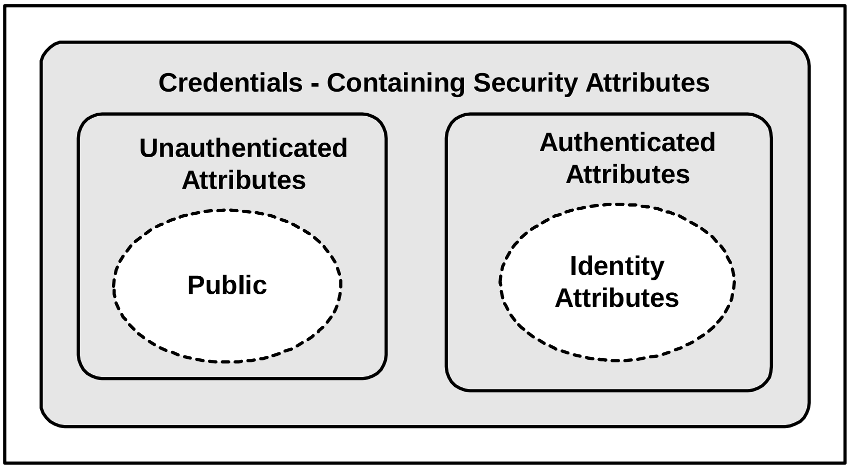
The Credentials Object

A Credentials object (as shown in Figure 13-2) 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 13-2 The Credentials Object


Credentials - Containing Security Attributes
Unauthenticated Attributes
Public
Authenticated Attributes
Identity Attributes

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.

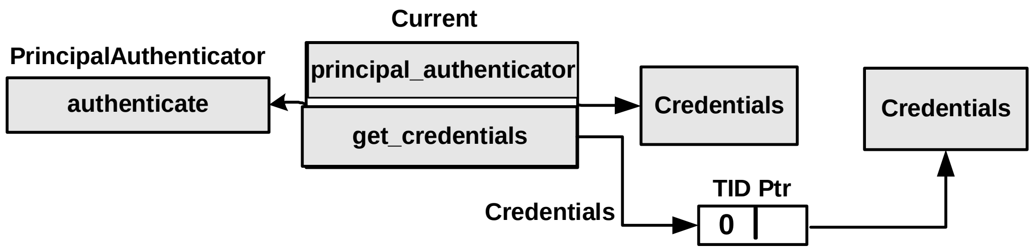
- The `invocation_options_supported` attribute indicates which security options are allowed when establishing a secure connection.

- 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 13-3) 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 the state associated with the thread of execution and the process in which the thread is executing.

Figure 13-3 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 CORBAservices 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.

CHAPTER

14 Security Modules

This topic contains the Object Management Group (OMG) Interface Definition Language (IDL) definitions for the following modules that are used in the CORBA security model:

- 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 14-1 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 14-1 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 14-2 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 14-2   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             TdfT;
       struct UtcT {
              TimeT              time;     // 8 octets
              unsigned long      inacclo;  // 4 octets
              unsigned short     inacchi;  // 2 octets
              TdfT              tdf;       // 2 octets
                                          // total 16 octets
       };
};
```

Table 14-1 defines the TimeBase module data types.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 14-6. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Table 14-1 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 14-1 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 14-3 shows the data types supported by the Security module.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 15-193 to 15-195. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Listing 14-3 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 14-2 describes the Security module data type.

Table 14-2 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 14-4 shows the Security Level 1 module OMG IDL statements.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 15-198. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Listing 14-4 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 14-5 shows the Security Level 2 module OMG IDL statements.

Note: This information is taken from *CORBAservices: Common Object Services Specification*, p. 15-198 to 15-200. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

Listing 14-5 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::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,
                     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 ATMI-style of authentication.

Listing 14-6 shows the Tobj module OMG IDL statements.

Listing 14-6 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
              );
        };
};
```

CHAPTER

15 C++ Security Reference

This topic contains the C++ method descriptions for CORBA security.

SecurityLevel1::Current::get_attributes

Synopsis

Returns attributes for the Current interface.

OMG IDL Definition

```
Security::AttributeList get_attributes(
       in Security::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 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 BEA Tuxedo username.
`Security::PrimaryGroupId`	Null terminated ASCII string containing the BEA Tuxedo 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 *CORBAservices: Common Object Services Specification*, pp. 15-103, 104. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

SecurityLevel2::PrincipalAuthenticator::authenticate

Synopsis

Authenticates the principal and optionally obtains credentials for the principal.

OMG IDL Definition

```
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 );
```

Arguments

`method`

The security mechanism to be used. Valid values are `Tobj::TuxedoSecurity` and `Tobj::CertificateBased`.

`security_name`

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.

When using certificate 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: `milozzi@company.com` is the e-mail address used to look up a certificate in the LDAP-enabled directory service and `milozzi_company.pem` is the name of the private key file.

`auth_data`

The principals' authentication, such as their password or private key. If the `Tobj:TuxedoSecurity` security mechanism is specified, the value of this argument is dependent on the configured level of authentication. If the `Tobj::CertificateBased` argument is specified, the value of this argument is the pass phrase used to decrypt the private key of the principal.

`privileges`

The privilege attributes requested.

`creds`

The object reference of the newly created Credentials object.The object reference is not fully initialized; therefore, the object reference cannot be used until the return value of the `SecurityLevel2::Current::authenticate` method is `SecAuthSuccess.`

`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 BEA Tuxedo 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 ATMI authentication security mechanism.

Return Values

The following table describes the valid return values.

Return Value	**Meaning**
`SecAuthSuccess`	The object reference of the newly created Credentials object returned as the value of the `creds` argument is initialized and ready to use.
`SecAuthFailure`	The authentication process was inconsistent or an error occurred during the process. Therefore, the `creds` argument does not contain an object reference to a Credentials object. If the `Tobj::TuxedoSecurity` 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 `Tobj::PrincipalAuthenticator::logoff` or the `Tobj_Bootstrap::destroy_current` operation.
`SecAuthContinue`	Indicates that the authentication procedure uses a challenge/response mechanism. The `creds` argument contains the object reference of a partially initialized Credentials object. The `continuation_data` indicates the details of the challenge.

Return Value	Meaning
`SecAuthExpired`	Indicates that the authentication data contained some information, the validity of which had expired; therefore, the `creds` argument does not contain an object reference to a Credentials object. If the `Tobj::TuxedoSecurity` security mechanism is used, this return value is never returned.
`CORBA::BAD_PARAM`	The `CORBA::BAD_PARAM` exception occurs if: ■ Values for the `security_name`, `auth_data`, or `privileges` arguments are not specified. ■ The length of an input argument exceeds the maximum length of the argument. ■ The value of the `method` argument is `Tobj::TuxedoSecurity` and the content of the `auth_data` argument contains a `username` or a `clientname` as an empty or a NULL 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 *CORBAservices: 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:
    classCredentials;
    typedefCredentials *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 *CORBAservices: 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 BEA Tuxedo 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. Crytography 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.
`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.

Security Option	Description
`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 CORBA application is defined to require passwords to access the BEA Tuxedo domain.

Note: A `CORBA::NO_PERMISSION` exception is returned if the security options specified are not supported by the security mechanism defined for the CORBA 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.

The `invocation_options_supported` attribute has `set()` and `get()` methods. You cannot use the `set()` method when using the `Tobj::TuxedoSecurity` security mechanism to get a Credentials object. If you do use the `set()` method with the `Tobj::TuxedoSecurity` security mechanism, a `CORBA::NO_PERMISSION` exception is returned.

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 BEA Tuxedo domain.

OMG IDL Definition

```
attribute Security::AssociationOptions
                                        invocation_options_required;
```

Argument	None.
Description	Use this method to specify that communication between principals and the BEA Tuxedo 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. Crytography 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 CORBA application is defined to require passwords to access the BEA Tuxedo domain.

Note: A `CORBA::NO_PERMISSION` exception is returned if the security options specified are not supported by the security mechanism defined for the CORBA 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.

The `invocation_options_required` attribute has `set()` and `get()` methods. You cannot use the `set()` method when using the `Tobj::TuxedoSecurity` security mechanism to get a Credentials object. If you do use the `set()` method with the `Tobj::TuxedoSecurity` security mechanism, a `CORBA::NO_PERMISSION` exception is returned.

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 *CORBAservices: 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::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 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:
     classPrincipalAuthenticator;
     typedefPrincipalAuthenticator * 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,
       in Credentials                  creds,
       out Security::Opaque            continuation_data,
       out Security::Opaque            auth_specific_data
);
```

Description

Because the BEA Tuxedo 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 *CORBAservices: 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 BEA Tuxedo domain.

OMG IDL Definition

```
AuthType get_auth_type();
```

Description

This method returns the type of authentication expected by the BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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 CORBA 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 BEA Tuxedo authentication service). The authentication level is `TOBJ_APPAUTH`. The password must not exceed 30 characters.

`user_data`

Data that is specific to the client application (needed for use by a custom BEA Tuxedo 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 BEA Tuxedo domain. The BEA Tuxedo 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 BEA Tuxedo domain. This method is functionally equivalent to `SecurityLevel2::PrincipalAuthenticator::authenticate`, but the arguments are oriented to ATMI 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**
`Security::AuthenticationStatus::SecAuthSuccess`	The authentication succeeded.

`Security::AuthenticationStatus::SecAuthFailure`	The authentication failed, or the client application was already authenticated and did not call one of the following methods:
	`Tobj::PrincipalAuthenticator:logoff`
	`Tobj_Bootstrap::destroy_current`

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 BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo username.

`client_name`
The CORBA client name.

`system_password`
The CORBA client application password.

`user_password`
The user password (default BEA Tuxedo authentication service).

`user_data`
Client application-specific data (custom BEA Tuxedo 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 BEA Tuxedo domain. The BEA Tuxedo 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.

CHAPTER

16 Java Security Reference

For information about the security application programming interface (API), see the *CORBA Javadoc* in the BEA Tuxedo online documentation.

17 Automation Security Reference

This topic contains the Automation method descriptions for CORBA security. 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 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 BEA Tuxedo 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 BEA Tuxedo username.
`Security::PrimaryGroupId`	Null-terminated ASCII string containing the BEA Tuxedo 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 BEA Tuxedo domain. In this release of the BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo domain.

Return Values

A `Security_AuthenticationStatus` Enum value. The following table describes the valid return values.

Return Value	**Meaning**
`Security::Authentication Status:: SecAuthSuccess`	The authentication succeeded.
`Security::Authentication Status:: SecAuthFailure`	The authentication failed, or the client application was already authenticated and did not invoke `Tobj::PrincipalAuthenticator:logoff` or `Tobj_Bootstrap::destroy_current`.

DITobj_PrincipalAuthenticator.build_auth_data

Synopsis

Creates authentication data and attributes for use by `DITobj_PrincipalAuthenticator.authenticate`.

MIDL Mapping

```
HRESULT build_auth_data(
  [in] BSTR                 user_name,
  [in] BSTR                 client_name,
  [in] BSTR                 system_password,
  [in] BSTR                 user_password,
  [in] VARIANT              user_data,
  [out] VARIANT*            auth_data,
  [out] VARIANT*            privileges,
  [in,out,optional] VARIANT* exceptionInfo);
```

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

`user_name`
: The BEA Tuxedo username.

`client_name`
: A name of the CORBA client application.

`system_password`
: The password for the CORBA client application.

`user_password`
: The user password (for default authentication service).

`user_data`
: Client application-specific data (custom authentication service).

`auth_data`
: For use by `authenticate`.

`privileges`
: For use by `authenticate`.

`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.

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 BEA Tuxedo domain. The 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 BEA Tuxedo software does authentication in one step, this method always fails and returns `Security::AuthenticationStatus::SecAuthFailure`.

Return Values

Always returns `SecAuthFailure`.

DITobj_PrincipalAuthenticator.get_auth_type

Synopsis

Gets the type of authentication expected by the BEA Tuxedo 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 BEA Tuxedo 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 `NONE` value for the `SECURITY` parameter in the `RESOURCES` section of the `UBBCONFIG` file.
`TOBJ_SYSAUTH`	The client application must authenticate itself to the BEA Tuxedo 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.

Return Value	Meaning
`TOBJ_APPAUTH`	The client application must provide proof material that authenticates the client application to the BEA Tuxedo 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.

DITobj_PrincipalAuthenticator.logon

Synopsis

Logs in to the BEA Tuxedo 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 CORBA client applications, this method authenticates the client application via the IIOP Listener/Handler so that the remote client application can access a BEA Tuxedo domain. This method is functionally equivalent to `DITobj_PrincipalAuthenticator.authenticate`, but the parameters are oriented to security.

Arguments

`user_name`
: The BEA Tuxedo username. This parameter is required for `TOBJ_NOAUTH`, `TOBJ_SYSAUTH`, and `TOBJ_APPAUTH` authentication levels.

`client_name`
: The name of the CORBA client application. This parameter is required for `TOBJ_NOAUTH`, `TOBJ_SYSAUTH`, and `TOBJ_APPAUTH` authentication levels.

`system_password`
: A password for the CORBA client application. This parameter is required for `TOBJ_SYSAUTH` and `TOBJ_APPAUTH` authentication levels.

`user_password`
: The user password (default 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**
`Security::AuthenticationStatus::SecAuthSuccess`	The authentication succeeded.

`Security::AuthenticationStatus::SecAuthFailure`	The authentication failed, or the client application was already authenticated and did not call one of the following methods:
	`Tobj::PrincipalAuthenticator:logoff`
	`Tobj_Bootstrap::destroy_current`

DITobj_PrincipalAuthenticator.logoff

Synopsis

Discards the current security context associated with the CORBA client application.

MIDL Mapping

```
HRESULT logoff([in, out, optional] VARIANT* exceptionInfo);
```

Automation Mapping

```
Sub logoff([exceptionInfo])
```

Description

This call discards the context associated with the CORBA client application, but does not close the network connections to the BEA Tuxedo domain. `Logoff` also invalidates the current credentials. After logging off, calls using existing object references fail if the authentication type is not `TOBJ_NOAUTH`.

If the client application is currently authenticated to a BEA Tuxedo domain, calling `Tobj_Bootstrap.destroy_current()` calls `logoff` implicitly.

Argument

`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_Credentials

The `DISecurityLevel2_Credentials` object is a BEA implementation of the CORBA Security model. In this release of the BEA Tuxedo 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 BEA Tuxedo domain

Listing 17-1 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