Oracle® Tuxedo

Using Security in CORBA Applications

12*c* Release 2 (12.2.2)

April 2016

ORACLE®

Contents

Overview of the CORBA Security Features

This topic includes the following sections:

- The CORBA Security Features
- The CORBA Security Environment
- Oracle Tuxedo Security SPIs

Notes: The Oracle 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 Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

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 Oracle Tuxedo product lets you establish secure connections between client and server applications. It has the following features:

- Authentication of CORBA C++ applications to the Oracle 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++ applications can establish SSL sessions with an Oracle Tuxedo domain. Oracle Tuxedo client applications can use LLE or SSL to protect network traffic between bridges and domains.
- 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 Oracle Tuxedo System*.

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

Table 1-1 summarizes the features in the CORBA security features in the Oracle 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.

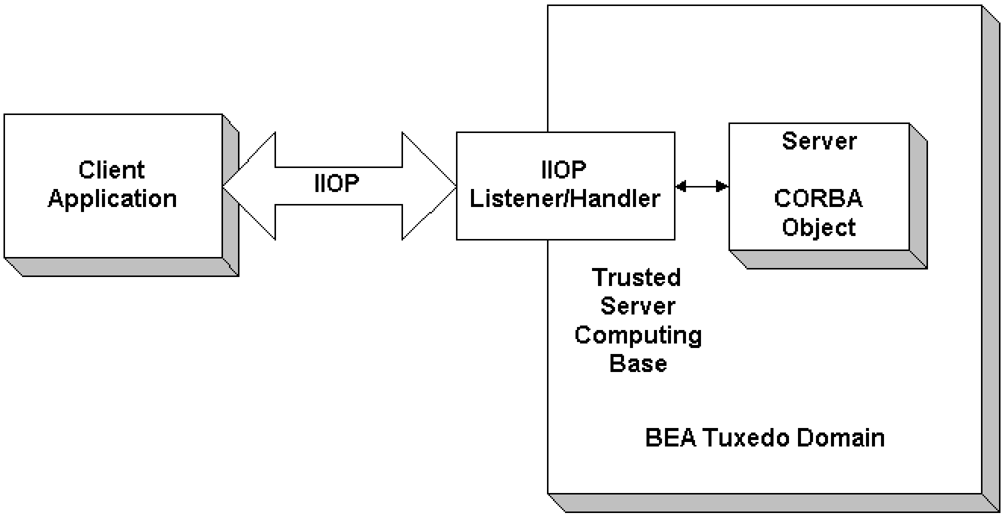
Table 1-1 CORBA Security Features (Continued)

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 Oracle Tuxedo domains.	N/A	The SSL version 3.0 protocol.
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. • AES or 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 Oracle 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 an Oracle 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 Oracle 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 an Oracle 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 Oracle 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 Oracle 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 Oracle Tuxedo product is to encrypt the authentication information but not to protect the message sent between the client application and the Oracle 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.

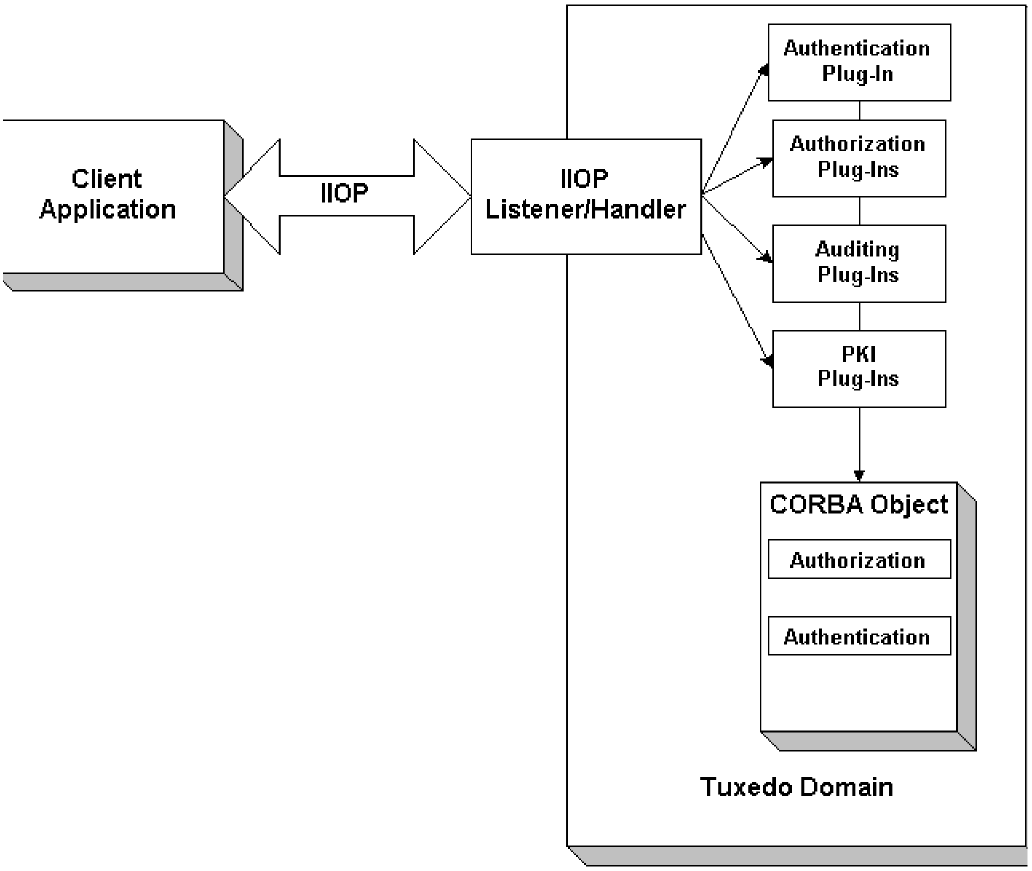
5. The IIOP Listener/Handler forwards the request along with the authorization and auditing tokens of the initiating principal to the appropriate server application.

6. When the request is received by the server application, the Oracle 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.

Oracle Tuxedo Security SPIs

As shown in Figure 1-2, the authentication, authorization, auditing, and public key security features available with the Oracle 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 1-2 Architecture for the Oracle Tuxedo Security Service Provider Interfaces



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

Table 1-2 The Oracle 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 Oracle 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 Oracle Systems, Inc. Customers who want to customize a security feature must contact one of these vendors or Oracle Professional Services. For example, an Oracle 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 Oracle Professional Services.

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

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

Notes: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

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

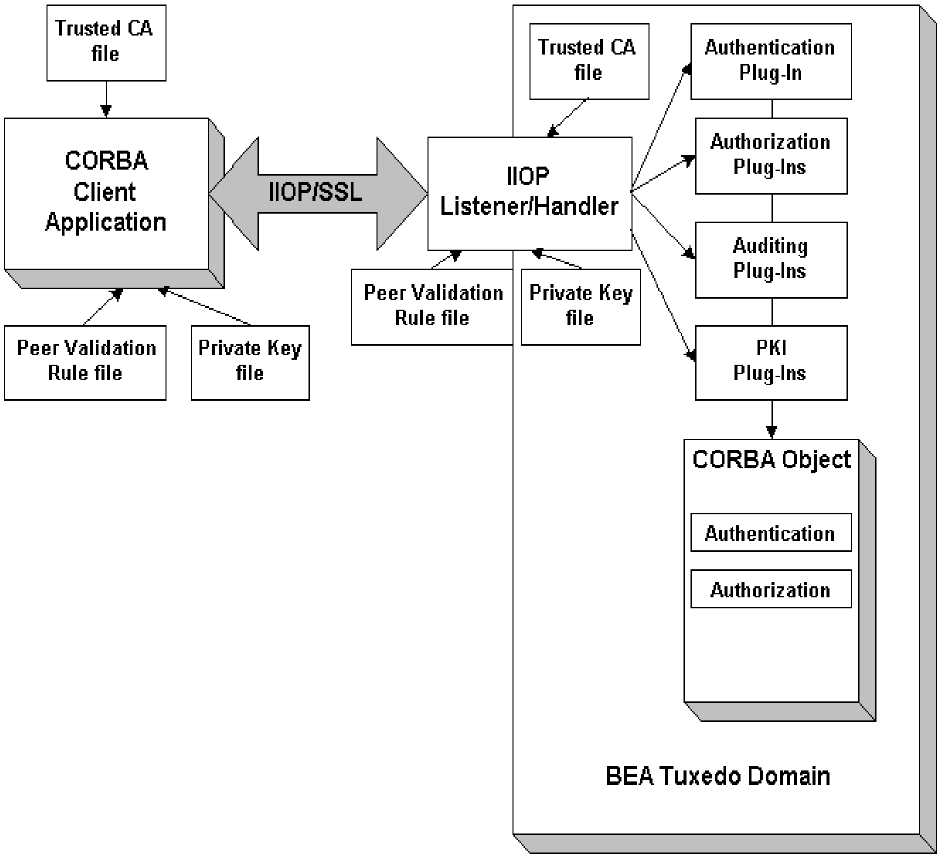
  Authentication is the ability to ascertain with whom you are speaking. By using digital certificates and public key security, CORBA client and server applications can each be authenticated to the other. This allows the two parties to be certain they are communicating with someone they trust. The SSL protocol provides a mechanism that can be used to authenticate principals to an Oracle 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 Oracle Tuxedo product.

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. Oracle 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.

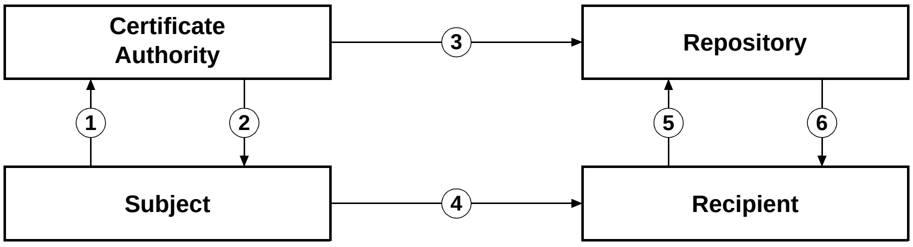
There is no single standard public key infrastructure today, though efforts are underway to define one. It is not yet clear whether a standard will be established or multiple independent PKIs will evolve with varying degrees of interoperability. In this sense, the state of PKI technology today can be 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 Oracle Tuxedo product does not provide the tools necessary to be a certificate authority. Oracle Systems, Inc. recommends using a third-party certificate authority such as VeriSign or Entrust. By offering a Public Key SPI, Oracle Systems, Inc. extends the opportunity to all Oracle Tuxedo customers to use a PKI security solution with the PKI software from their vendor of choice. See "PKI Plug-ins" on page 3-22 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 Oracle 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 Oracle 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.

- AES-256-CBC (Advanced Encryption Standard for Cipher Block Chaining)

  AES-256-CBC is a 128-bit block cipher run in Cipher Block Chaining (CBC) mode. It provides 256-bits keys

Customers of the Oracle 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

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

Cipher Suite	Key Exchange Type	Symmetric Key Strength
`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. Oracle recommends obtaining certificates from a certificate authority such as Verisign or Entrust.

Fundamentals of CORBA Security

This topic includes the following sections:

Notes: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

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 Oracle 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 Oracle 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.

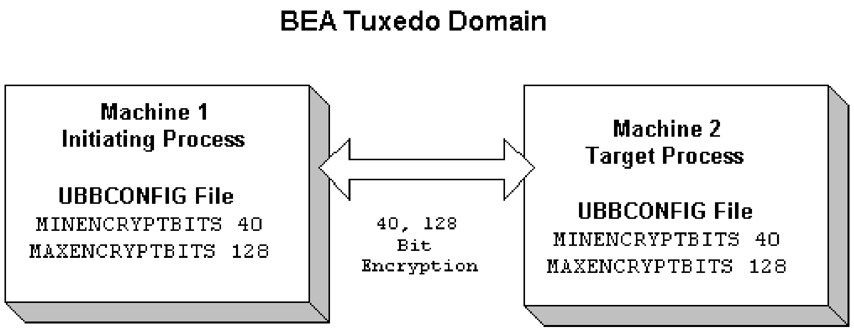
2. An initiator process begins the communication session.

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 Oracle 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 Oracle 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
(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 Oracle 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 Oracle 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 3-9.

How Password Authentication Works

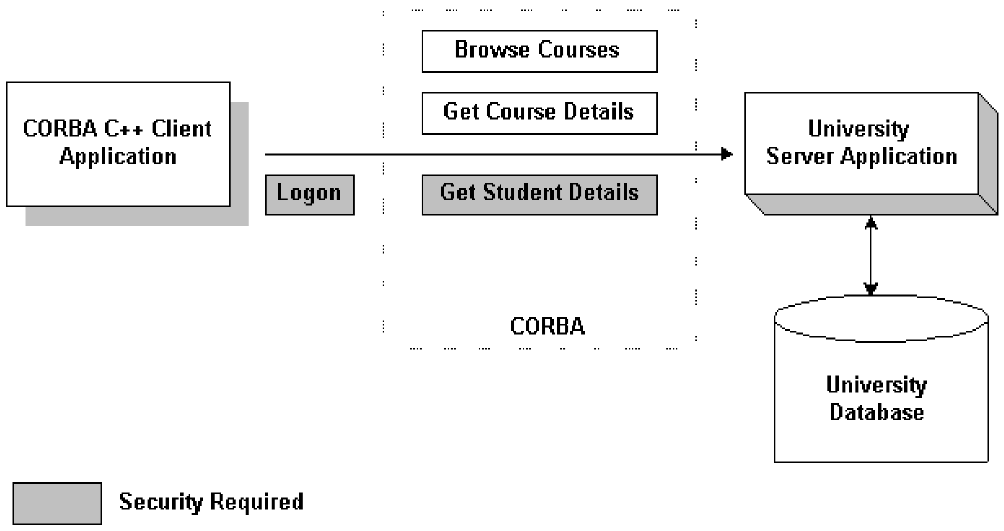
Password authentication works in the following way:

1. The initiating application accesses the Oracle 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 Oracle Tuxedo online documentation
   - The Oracle Bootstrapping mechanism. Use this mechanism if you are using Oracle CORBA client applications.
2. The initiating application obtains credentials for the user. The initiating application must provide proof material to be used by the Oracle 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 Oracle 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 Oracle 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



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`.
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 Oracle 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 Oracle Tuxedo domain.
4	Write application code that prompts the user for the password defined when the `UBBCONFIG` file is loaded.

The SSL Protocol

The Oracle 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 Oracle 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 Oracle 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

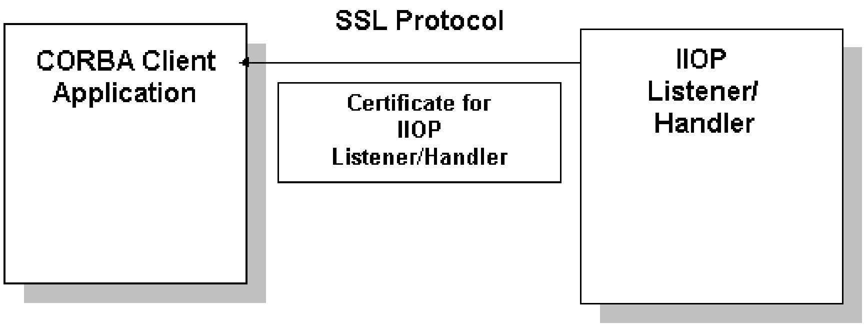
The SSL protocol works in the following manner:

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 Oracle 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 Oracle Tuxedo System*.

The implementation of the SSL protocol is flexible enough to fit into most public key infrastructures. The Oracle 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 3-4 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 Oracle 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.

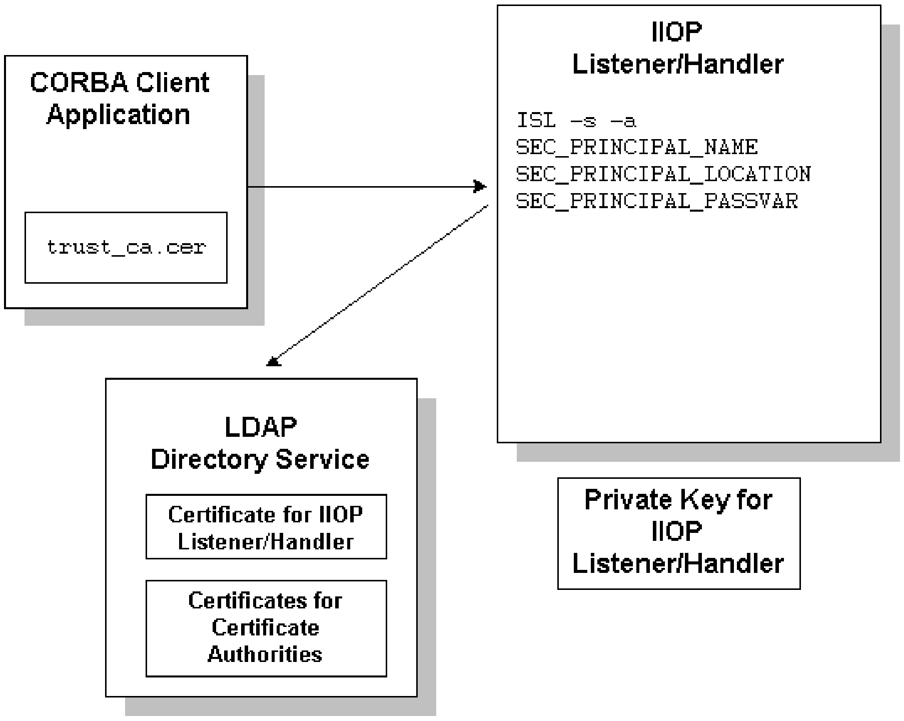
Table 3-4 Administration Steps for the SSL Protocol

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 Oracle 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.
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 3-5.

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

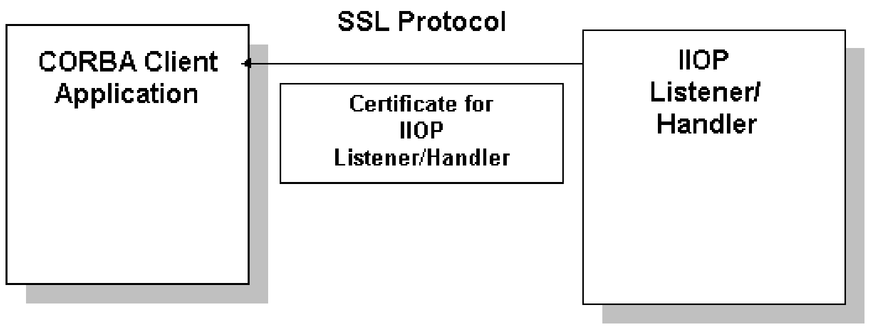
Certificate authentication requires that each side of an SSL connection proves its identity to the other side of the connection. In the CORBA security environment, the IIOP Listener/Handler presents its digital certificate to the principal who initiated the SSL connection. The initiator then provides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



Commonly, X.509 V3 CA certificates are required to contain the Basic Constraints extension, marked as being from a Certificate Authority (CA), and marked as a critical extension (see IETF RFC 2459). Ensuring that V3 CA certificates protects against non-CA certificates from masquerading as intermediate CA certificates.

For more information, please refer to the following URL:

`http://www.ietf.org/rfc/rfc2459.txt`

Note: This default behavior will not check Basic Constraints on X.509 V1 and V2 certificates, as these versions of X.509 certificates do not support certificate extensions.

There is a mechanism provided to control the level of enforcement that will be performed in order to avoid problems with some customer's applications:

The mechanism is used by setting the value of the environment variable TUX_SSL_ENFORCECONSTRAINTS. The levels of enforcement are as follows:

1

This level is the default. No checking is performed on V1 or V2 certificates in the certificate chain. The Basic Constraints for V3 CA certificates are checked and the certificates are verified to be CA certificates.

```
TUX_SSL_ENFORCECONSTRAINTS=1
```

2

This level does the same checking as level 1, and additionally enforces two more requirements:

- All CA certificates in the certificate chain must be V3 certificates.
- The Basic Constraints extensions of the CA certificates must be marked as "critical" in accordance with IETF RFC 2459.

This is not the default setting because a number of current commercially available V3 CA certificates do not mark the Basic Constraints as critical.

```
TUX_SSL_ENFORCECONSTRAINTS=2
```

Note: In versions of Tuxedo prior to Tuxedo 12.1.1 a value of 0 was also allowed, which disabled Basic Constraints enforcement entirely. This option was provided for compatibility with older certificates back when Basic Constraints were still a fairly recent feature in the X.509 standard. Since this is no longer the case, the `TUX_SSL_ENFORCECONSTRAINTS=0` value is no longer supported in Tuxedo 12.1.1 and later releases.

How Certificate Authentication Works

Certificate authentication works in the following manner:

1. The initiating application accesses the Oracle 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 Oracle Tuxedo online documentation.
   - The Oracle Bootstrapping mechanism. Use this mechanism if you are using the Oracle 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 Oracle 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 Oracle 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.

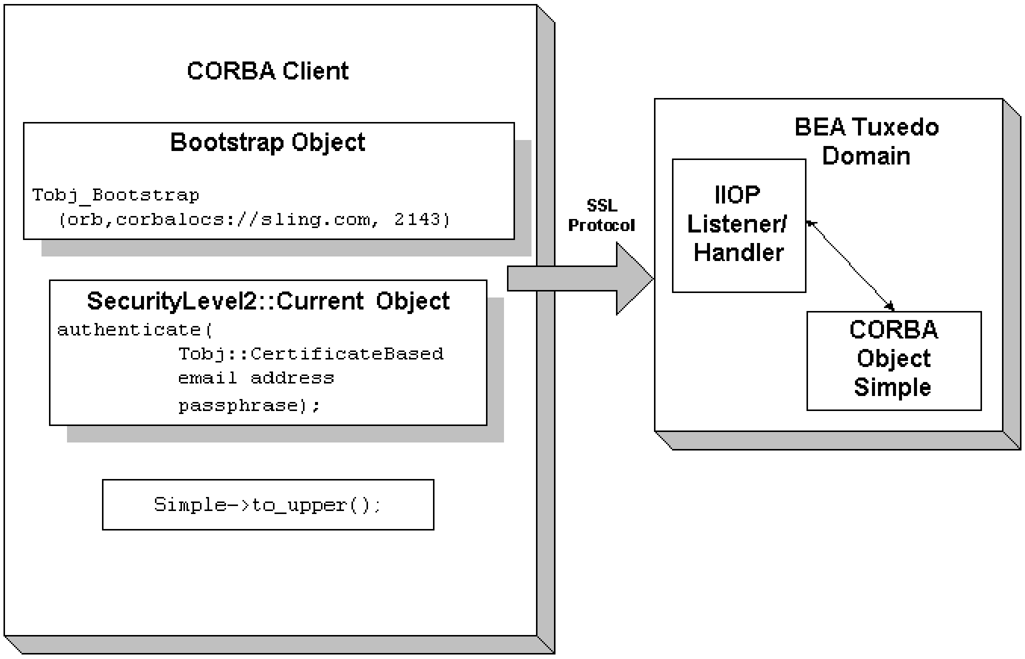
7. The request is digitally signed and encrypted before it is sent to the IIOP Listener/Handler. The Oracle Tuxedo system performs the signing and sealing of requests.

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



```
Tobj_Bootstrap
   (orb,corbalocs://sling.com, 2143)
```

```
authenticate(
             Tobj::CertificateBased
             email address
             passphrase);
```

```
Simple->to_upper();
```

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 Oracle 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 Oracle 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.
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.

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

Step	Description
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

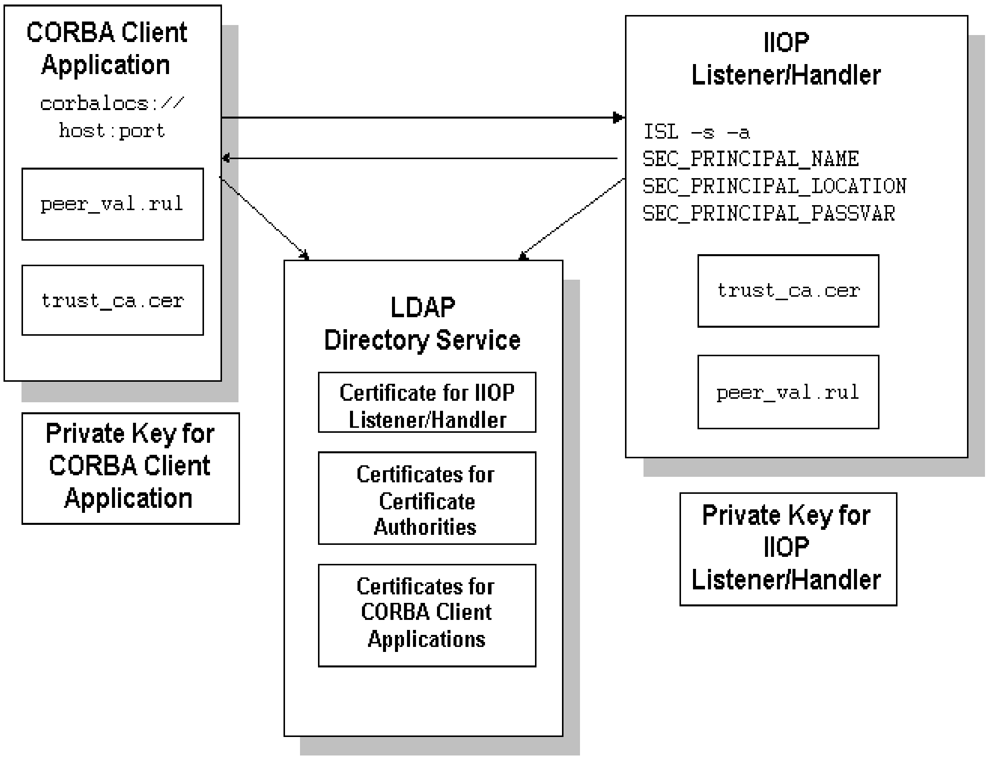


Table 3-6 lists the programming steps for using certificate authentication in a CORBA application. For more information, see "Writing a CORBA Application That Implements Security" on page 10-1.

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 Oracle 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 Oracle Tuxedo product allows the integration of authentication plug-ins into a CORBA application. The Oracle 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 Oracle Tuxedo system. For more detail about the registry, see "Configuring Security Plug-ins" on page 8-1.

For more information about an authentication plug-ins, including installation and configuration procedures, see your Oracle 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 Oracle Tuxedo system. For more detail about the registry, see "Configuring Security Plug-ins" on page 8-1**.**

For more information about authorization plug-ins, including installation and configuration procedures, see your Oracle 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 an Oracle Tuxedo Application*.

You can enhance the auditing capabilities of your CORBA application by using an auditing plug-in. The Oracle 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, an Oracle 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 Oracle 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 Oracle Tuxedo system. For more detail about the registry, see "Configuring Security Plug-ins" on page 8-1**.**

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

PKI Plug-ins

The Oracle 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.
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 Oracle Tuxedo system. For more detail about the registry, see "Configuring Security Plug-ins" on page 8-1.

For more information about PKI plug-ins, including installation and configuration procedures, see your Oracle 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 Oracle Tuxedo 8.0 or later.

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 “PKI Plug-ins” on page 3-22.

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 Oracle Tuxedo domain, the client application can still have an Oracle Tuxedo username and password. The IIOP Listener/Handler maps the identity defined in a digital certificate to an Oracle Tuxedo username and password thus allowing existing CORBA applications to have an identity in native CORBA server applications.

For more information, see "PKI Plug-ins" on page 3-22.

CHAPTER 10

Writing a CORBA Application That Implements Security

This topic includes the following sections:

Notes: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Using the Bootstrapping Mechanism

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

The Bootstrap object in the Oracle 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 Oracle 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|corbalocs://55
5last.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 Oracle 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:111
1
```

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.

Oracle 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.

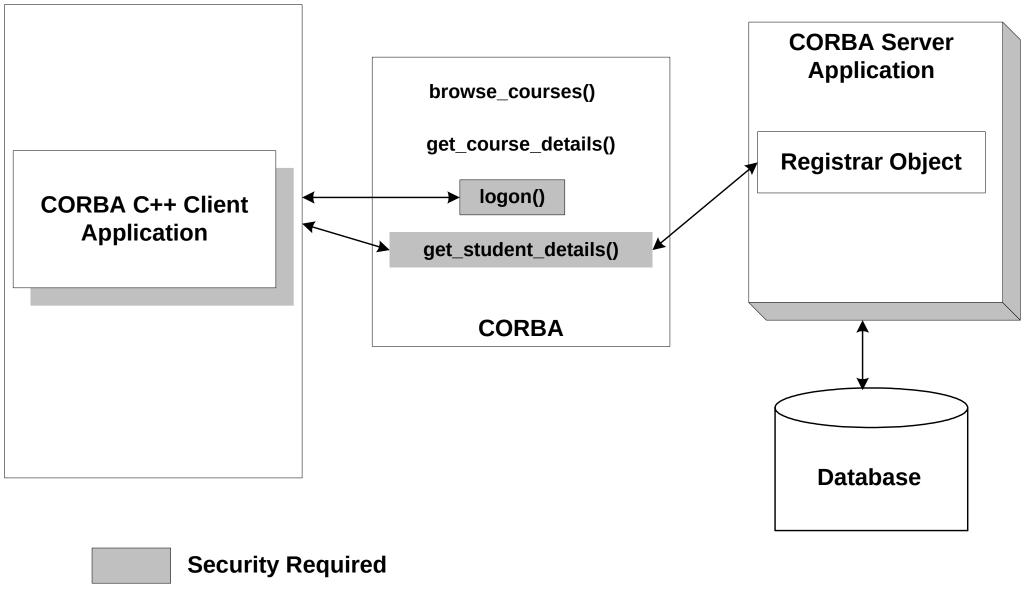
The Security Sample Application

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 Oracle 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 Oracle 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`
   - C++—`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 Oracle 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 Oracle 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++ code examples that illustrate implementing password authentication. For a Visual Basic code example, see "Automation Security Reference" on page 16-1.

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
...
```

C++ Code Example That Uses the Tobj::PrincipalAuthenticator::logon() Method

Listing 10-4 contains C++ code that performs password authentication using the `Tobj::PrincipalAuthenticator::logon()`method.

Listing 10-4 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();
         }
...
```

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 Oracle Tuxedo software. For instructions for building and running the Secure Simpapp sample application, see "Building and Running the CORBA Sample Applications" on page 9-1.

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 Oracle 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++ code examples that illustrate implementing certificate authentication.

C++ Code Example of Certificate Authentication

Listing 10-5 illustrates using certificate authentication in a CORBA C++ client application.

Listing 10-5 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_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);
              while(auth_status == Security::SecAuthContinue) {
              auth_status = PA->continue_authentication(
                                      response_data,
                                      the_creds,
                                      continuation_data,
                                        auth_specific_data);
              }
...
```

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 Oracle 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 Oracle Tuxedo domain. The `SecurityLevel2::PrincipalAuthenticator` object adheres to the standard CORBAservices Security Service instead of the proprietary Oracle 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 Oracle Tuxedo domain and authenticate the client ORB to the Oracle Tuxedo domain. If security credentials are required to access the Oracle 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 Oracle 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.

Protecting the Client Credentials

The following information provides a sample that protects the client credentials before performing the step of continuing authentication.

The following example assumes a Java client using J2SE v 1.4, accessing an Oracle Tuxedo application.

1. Add `$TUXDIR/udataobj/java/jdk/tuxsecenv.jar` to your `CLASSPATH`.
2. In your client code, call `com.bea.protectLogonData()` before you call the `PrincipalAuthenticator continue_authentication()` method.
3. The following is sample code that shows a `protectLogonData()` call. This code depends on Java classes that are generated from these IDL files in `$TUXDIR/include: security.idl, lcs.idl, ns.idl, tobj.idl`.

Listing 10-6 Sample Client Code Using CORBA INS

```
try {
    // Initialize the ORB.

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

    // Authentication

    org.omg.CORBA.Object sec_obj =
      orb.resolve_initial_references("PrincipalAuthenticator");

    org.omg.SecurityLevel2.PrincipalAuthenticator pa =
    org.omg.SecurityLevel2.PrincipalAuthenticatorHelper.narrow(sec_obj);

    String userName = "geni";
    String clientName  = "SimpleClient";

    org.omg.Security.SecAttribute[] privilege =
    new org.omg.Security.SecAttribute[1];

  org.omg.SecurityLevel2.CredentialsHolder myCreds =
    new org.omg.SecurityLevel2.CredentialsHolder();

  org.omg.Security.OpaqueHolder cont_data =        // continuation data
    new org.omg.Security.OpaqueHolder();

  org.omg.Security.OpaqueHolder auth_data =       // auth specific data
    new org.omg.Security.OpaqueHolder();

  org.omg.Security.AuthenticationStatus status = pa.authenticate(
      1,
     userName,
     clientName.getBytes(),
     privilege,
     myCreds,
     cont_data,
     auth_data
     );

  if (status.value() == 2) {

    // further authentication required

    org.omg.SecurityLevel2.Credentials creds = myCreds.value;
    String secUid = new String(cont_data.value);

  org.omg.Security.OpaqueHolder cont_data_2 =
    new org.omg.Security.OpaqueHolder();
  org.omg.Security.OpaqueHolder auth_data_2 =
    new org.omg.Security.OpaqueHolder();
  org.omg.Security.OpaqueHolder opqholder =
    new org.omg.Security.OpaqueHolder();
  byte[] ba0 = new byte[0];

    String userPasswd = new String("abc123");
    String domainPasswd = new String("abc123");
```

```
        //  encrypt the logon data
        com.bea.LogonData td = new com.bea.LogonData();
         int rc = td.protectLogonData(
            userName,
            clientName,
            domainPasswd,
            userPasswd,
            secUid,
            ba0,
            opqholder
            );

         // continue authentication
         status = pa.continue_authentication(
            opqholder.value,
            creds,
            cont_data_2,
            auth_data_2
            );
       }
       else {
         System.out.println("No security required");
       }
     .
     .
     .
```

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 14-1 and "Java Security Reference" on page 15-1 for a complete list of security options.

Listing 10-7 provides a C++ example that uses the `invocation_options_required()` method.

Listing 10-7 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);
              }
```

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

Notes: The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

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 13-1 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 13-1 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 13-2 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 13-2 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 13-1 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 13-1 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.
`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 13-3 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 13-3 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 13-2 describes the Security module data type.

Table 13-2 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 13-4 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 13-4 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 13-5 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 13-5 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 13-6 shows the Tobj module OMG IDL statements.

Listing 13-6 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
              );
        };
};
```

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 Oracle Tuxedo username.
`Security::PrimaryGroupId`	Null terminated ASCII string containing the Oracle 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 Oracle 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 Oracle 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.
`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 Oracle 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 Oracle 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 Oracle 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.
`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 Oracle 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++. 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 Oracle 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 Oracle Tuxedo domain.

OMG IDL Definition

```
AuthType get_auth_type();
```

Description

This method returns the type of authentication expected by the Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle 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 Oracle Tuxedo domain. The Oracle 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 an Oracle 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 Oracle 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 an Oracle 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 Oracle Tuxedo username.

`client_name`
: The CORBA client name.

`system_password`
: The CORBA client application password.

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

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

Automation Security Reference

This topic contains the Automation method descriptions for CORBA security. This topic includes the following section:

- Method Descriptions

Notes: The Automation security methods do not support certificate authentication or the use of the SSL protocol.

The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Method Descriptions

This section describes the Automation Security Service methods.

DISecurityLevel2_Current

The `DISecurityLevel2_Current` object is an Oracle implementation of the CORBA Security model. In this release of the Oracle 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.

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 Oracle Tuxedo username.
`Security::PrimaryGroupId`	Null-terminated ASCII string containing the Oracle 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.

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.

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 Oracle Tuxedo domain. In this release of the Oracle 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 Oracle 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.

Description

This method authenticates the client application via the IIOP Listener/Handler so that it can access an Oracle 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 Oracle 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.

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 Oracle 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 Oracle 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 Oracle 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.

Description

This method returns the type of authentication expected by the Oracle 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 Oracle 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 Oracle 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 Oracle 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 an Oracle Tuxedo domain. This method is functionally equivalent to `DITobj_PrincipalAuthenticator.authenticate`, but the parameters are oriented to security.

Arguments

`user_name`
: The Oracle 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.

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 Oracle 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 an Oracle 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.

Return Values

None.

DISecurityLevel2_Credentials

The `DISecurityLevel2_Credentials` object is an Oracle implementation of the CORBA Security model. In this release of the Oracle 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.

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`.