Oracle® Tuxedo

Using Security in ATMI Application

Release 22c (22.1.0.0.0)
F74257-04
November 2024

Oracle Tuxedo Using Security in ATMI Application, Release 22c (22.1.0.0.0)

F74257-04

Primary Authors: Preeti Gandhe, Tulika Das

Contributors: Maggie Li

Contents

1 Overview

2 Administering Security

3 Programming Security

6 Integrating Audit with Oracle Platform Security Services (OPSS)

Index

List of Figures

List of Tables

1 Overview

The following sections describe the various security capabilities available with the Oracle Tuxedo system for ATMI applications:

> **Note:**
>
> 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 an ATMI application. For information about implementing security in a CORBA application, see *Using Security in CORBA Applications*.

1.1 What Security Means

Security refers to techniques for ensuring that data stored in a computer or passed between computers is not compromised. Most security measures involve *passwords* and *data encryption*, where a password 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 is unintelligible without a deciphering mechanism.

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 Oracle Tuxedo product provides several security capabilities for ATMI applications, most of which can be customized for your particular needs.

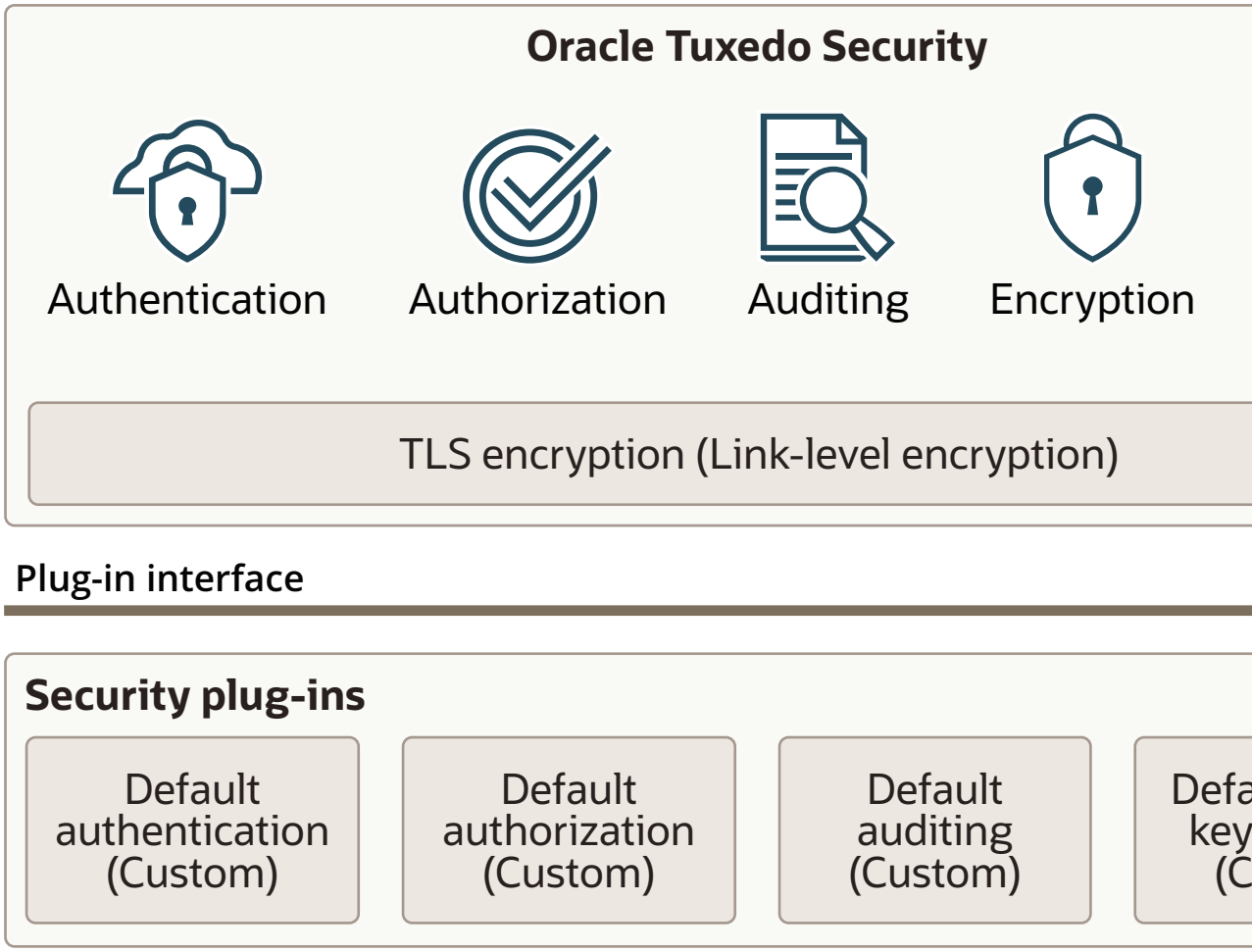
See Also:

- Security Plug-ins
- ATMI Security Capabilities
- Security Administration Tasks
- What Programming Security Means

1.2 Security Plug-ins

As shown in the following figure, all but one of the security capabilities available with the ATMI environment of the Oracle Tuxedo product are implemented through a *plug-in interface*, which allows Oracle Tuxedo customers to independently define and dynamically add their own *security plug-ins*. A security plug-in is a code module that implements a particular security capability.

Figure 1-1 Oracle Tuxedo ATMI Plug-in Security Architecture



The specifications for the security plug-in interface are not generally available, but are available to third-party security vendors. Third-party security vendors can enter into a special agreement with Oracle Systems to develop security plug-ins for Oracle Tuxedo. Oracle Tuxedo customers who want to customize a security capability must contact one of these vendors. For example, an Oracle Tuxedo customer who wants a custom implementation of public key security must contact a third-party security vendor who can provide the appropriate plug-ins. For more information about security plug-ins, including installation and configuration procedures, see your Oracle account executive.

See Also:

- ATMI Security Capabilities

1.3 ATMI Security Capabilities

The Oracle Tuxedo system can enforce security in a number of ways, which includes using the security features of the host operating system to control access to files, directories, and system

resources. In the following table describes the security capabilities available with the ATMI environment of the Oracle Tuxedo product.

Table 1-1 ATMI Security Capabilities

Security Capability	Description	Plug-in Interface	Default Implementation
Operating system security	Controls access to files, directories, and system resources.	N/A	N/A
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	The default authorization plug-in provides security at three levels: no authentication, application password, and user-level authentication. This plug-in works the same way the Oracle Tuxedo implementation of authentication has worked since it was first made available with the Oracle Tuxedo system.
Authorization	Controls access to resources based on identity or other information.	Implemented as a single interface	The default authorization plug-in provides security at two levels: optional access control lists and mandatory access control lists. This plug-in works the same way the Oracle Tuxedo implementation of authorization has worked since it was first made available with the Oracle Tuxedo system.
Auditing	Safely collects, stores, and distributes information about operating requests and their outcomes.	Implemented as a single interface	Default auditing security is implemented by the Oracle Tuxedo EventBroker and user log (ULOG) features.
Link-level encryption	Uses symmetric key encryption to establish data privacy for messages moving over the network links that connect the machines in an ATMI application.	N/A	RC4 symmetric key encryption.
TLS Encryption	Uses the industry-standard TLS protocol to establish data privacy for messages moving over the network links that connect the machines in an ATMI application. (TLS is the successor standard to the SSL protocol.)	N/A	Oracle NZ Security Layer
Public key security	Uses public key (or asymmetric key) encryption to establish end-to-end digital signing and data privacy between ATMI application clients and servers. Complies with the PKCS-7 standard.	Implemented as six interfaces	Default public key security supports the following algorithms: • RSA public key algorithm • RSA and DSA digital signature algorithms • DES-CBC, two-key triple-DES, and RC2 symmetric key algorithms • MD5 and SHA-1 message digest algorithms

See Also:

- Operating system security
- Authentication
- Authorization
- Auditing
- Link-level encryption
- TLS Encryption
- Public key security

1.4 Operating System (OS) Security

On host operating systems with underlying security features, such as file permissions, the operating-system level of security is the first line of defense. An application administrator can use file permissions to grant or deny access privileges to specific users or groups of users.

Most ATMI applications are managed by an application administrator who configures the application, starts it, and monitors the running application dynamically, making changes as necessary. Because the ATMI application is started and run by the administrator, server programs are run with the administrator's permissions and are therefore considered secure or "trusted." This working method is supported by the login mechanism and the read and write permissions on the files, directories, and system resources provided by the underlying operating system.

Client programs are run directly by users with the users' own permissions. In addition, users running native clients (that is, clients running on the same machine on which the server program is running) have access to the `UBBCONFIG` configuration file and interprocess communication (IPC) mechanisms such as the *bulletin board* (a reserved piece of shared memory in which parameters governing the ATMI application and statistics about the application are stored).

For ATMI applications running on platforms that support greater security, a more secure approach is to limit access to the files and IPC mechanisms to the application administrator and to have "trusted" client programs run with the permissions of the administrator (using the `setuid` command on a UNIX host machine or the equivalent command on another platform). For the most secure operating system security, allow only Workstation clients to access the application; client programs should not be allowed to run on the same machines on which application server and administrative programs run.

See Also:

- Security Administration Tasks
- Administering Operating System (OS) Security
- "About the Configuration File" and "Creating the Configuration File" in *Setting Up an Oracle Tuxedo Application*
- UBBCONFIG(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

1.5 Authentication

Authentication allows communicating processes to mutually prove identification. The authentication plug-in interface in the ATMI environment of the Oracle Tuxedo product can accommodate various security-provider authentication plug-ins using various authentication technologies, including *shared-secret password*, *one-time password*, *challenge-response*, and *Kerberos*. The interface closely follows the generic security service (GSS) application programming interface (API) where applicable; the GSSAPI is a published standard of the Internet Engineering Task Force. The authentication plug-in interface is designed to make integration of third-party vendor security products with the Oracle Tuxedo system as easy as possible, assuming the security products have been written to the GSSAPI.

- Authentication Plug-in Architecture
- Understanding Delegated Trust Authentication
- Establishing a Session
- Getting Authorization and Auditing Tokens
- Replacing Client Tokens with Server Tokens
- Implementing Custom Authentication

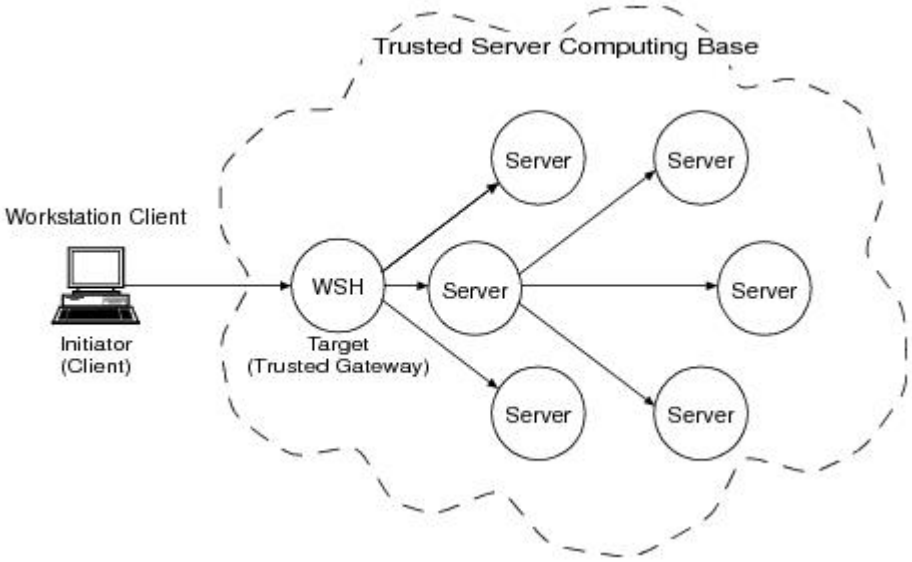
1.5.1 Authentication Plug-in Architecture

The underlying plug-in interface for authentication security is implemented as a single plug-in. The plug-in may be the default authentication plug-in or a custom authentication plug-in.

1.5.2 Understanding Delegated Trust Authentication

Direct end-to-end mutual authentication in a distributed enterprise middleware environment such as the Oracle Tuxedo system can be prohibitively expensive, especially when accomplished with security mechanisms optimized for long-duration connections. It is not efficient for clients to establish direct network connections with each server process, nor is it practical to exchange and verify multiple authentication messages as part of processing each service request. Instead, the ATMI applications use a *delegated trust* authentication model, as shown in the following figure:

Figure 1-2 ATMI Delegated Trust Authentication Model



A Workstation client authenticates to a *trusted system gateway process*, the workstation handler (WSH), at initialization time. A native client authenticates within itself, as explained later in this discussion. After a successful authentication, the authentication software assigns a security *token* to the client. A token is an opaque data structure suitable for transfer between processes. The WSH safely stores the token for the authenticated Workstation client, or the authenticated native client safely stores the token for itself.

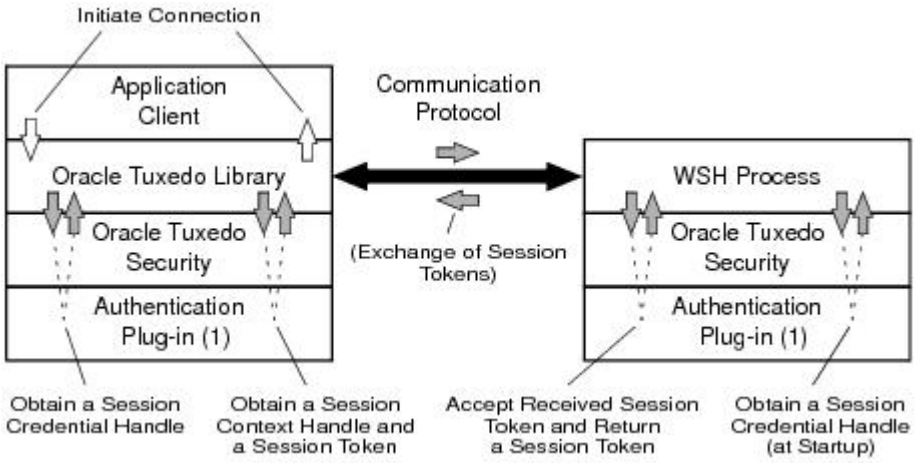
As a client request flows through a trusted gateway, the gateway attaches the client's security token to the request. The security token travels with the client's request message, and is delivered to the destination server process(es) for authorization checking and auditing purposes.

In this model, the gateway trusts that the authentication software will verify the identity of the client and generate an appropriate token. Servers, in turn, trust that the gateway process will attach the correct security token. Servers also trust that any other servers involved in the processing of a client request will safely deliver the token.

1.5.3 Establishing a Session

The following figure illustrates the control flow inside the ATMI environment of the Oracle Tuxedo system while a session is being established between a Workstation client and the WSH. The Workstation client and WSH are attempting to establish a long-term mutually authenticated connection by exchanging messages.

Figure 1-3 Control Flow in the ATMI Environment



The *initiator process* (may be thought of as a middleware client process) creates a *session context* by repeatedly calling the Oracle Tuxedo “initiate security context” function until a return code indicates success or failure. A session context associates identity information with an authenticated user.

When a Workstation client calls tpinit(3c) for C or TPINITIALIZE(3cbl) for COBOL to join an ATMI application, the Oracle Tuxedo system begins its response by first calling the internal “acquire credentials” function to obtain a session credential handle, and then calling the internal “initiate security context” function to obtain a session context. Each invocation of the “initiate security context” function takes an input *session token* (when one is available) and returns an output *session token*. A session token carries a protocol for verifying a user’s identity. The initiator process passes the output session token to the session’s *target process* (WSH), where it is exchanged for another input token. The exchange of tokens continues until both processes have completed mutual authentication.

A security-provider authentication plug-in defines the content of the session context and session token for its security implementation, so ATMI authentication must treat the session context and session token as opaque objects. The number of tokens passed back and forth is not defined, and may vary based on the architecture of the authentication system.

For a native client initiating a session, the initiator process and the target process are the same; the process may be thought of as a middleware client process. The middleware client process calls the security provider’s authentication plug-in to authenticate the native client.

1.5.4 Getting Authorization and Auditing Tokens

After a successful authentication, the trusted gateway calls two Oracle Tuxedo internal functions that retrieve an *authorization token* and an *auditing token* for the client, which the gateway stores for safekeeping. Together, these tokens represent the user identity of a security context. The term *security token* refers collectively to the authorization and auditing tokens.

When default authentication is used, the authorization token carries two pieces of information:

- *Principal name*—the name of an authenticated user.
- *Application key*—a 32-bit value that uniquely identifies the client initiating the request message. See Application Key for more detail.

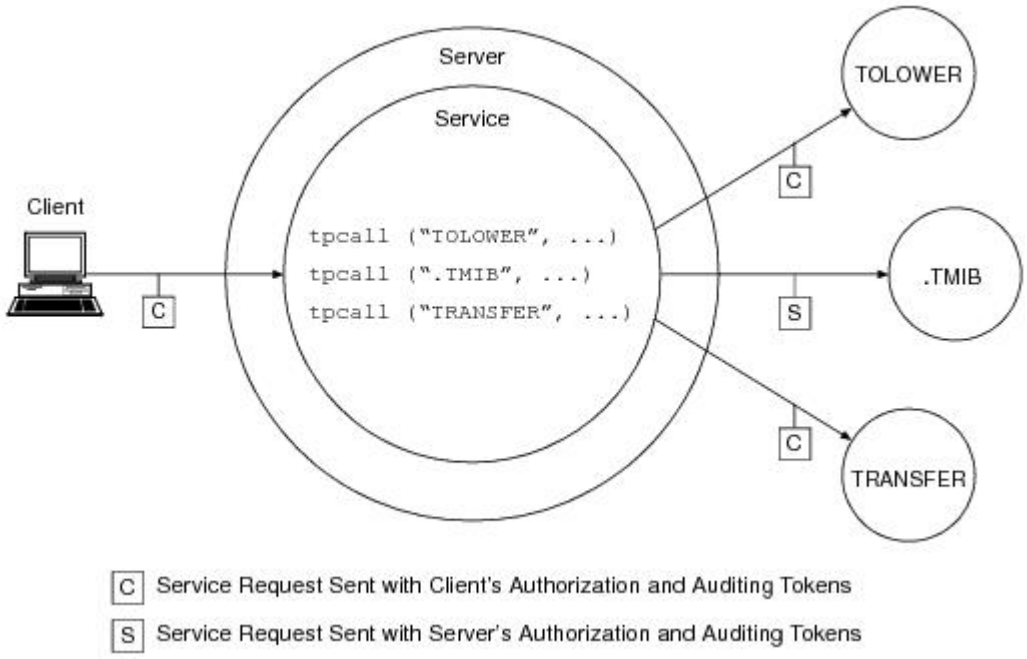
In addition, when default authentication is used, the auditing token carries the same two pieces of information: *principal name* and *application key*.

Like the session token, the authentication and auditing tokens are opaque; their contents are determined by the security provider. The authorization token can be used for performing authorization (permission) checks. The auditing token can be used for recording audit information. In some ATMI applications, it is useful to keep separate user identities for authorization and auditing.

1.5.5 Replacing Client Tokens with Server Tokens

As shown in the following figure, there are situations where a client service request forwarded by a server takes on the identity of the server. The server replaces the client tokens attached to the request with its own tokens and then forwards the service request to the destination service.

Figure 1-4 Server Permission Upgrade Example



> **Note:**
>
> See Specifying Principal Names for an understanding of how servers acquire their own authorization and auditing tokens and why they need them.

The feature demonstrated in the preceding figure is known as *server permission upgrade*, which operates in the following manner: whenever a server calls a *dot* service (a system-supplied service having a beginning period in its name—such as `.TMIB`), the service request takes on the identity of the server and thus acquires the access permissions of the server. A server's access permissions are those of the application (system) administrator. Thus, certain requests that would be denied if the client called the dot service directly would be allowed if the client sent the requests to a server, and the server forwarded the requests to the dot service. For more information about dot services, see the `TMIB` service description on the MIB(5)

reference page in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

1.5.6 Implementing Custom Authentication

You can provide authentication for your ATMI application by using the default plug-in or a custom plug-in. You choose a plug-in by configuring the Oracle Tuxedo *registry*, a tool that controls all security plug-ins.

If you want to use the default authentication plug-in, you do not need to configure the registry. If you want to use a custom authentication plug-in, however, you must configure the registry for your plug-in before you can install it. For more detail about the registry, see Setting the Oracle Tuxedo Registry.

See Also:

- Default Authentication and Authorization
- Security Administration Tasks
- Administering Authentication
- Programming an ATMI Application with Security
- Writing Security Code So Client Programs Can Join the ATMI Application

1.6 Authorization

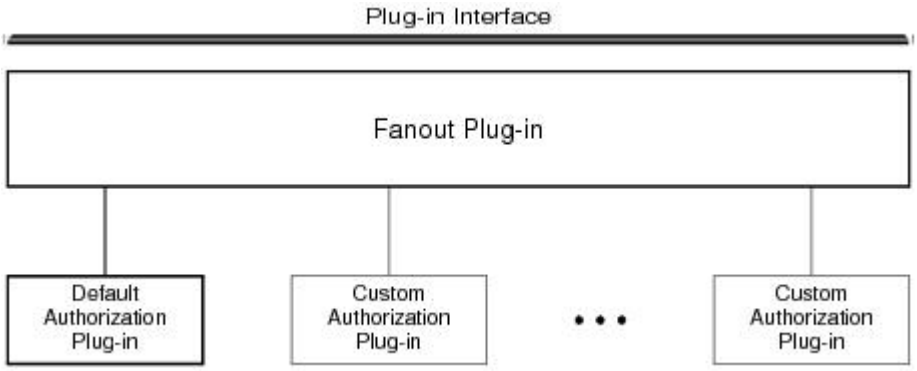
Authorization allows administrators to control access to ATMI applications. Specifically, an administrator can use authorization to allow or disallow *principals* (authenticated users) to use resources or facilities in an ATMI application.

- Authorization Plug-in Architecture
- How the Authorization Plug-in Works
- Implementing Custom Authorization

1.6.1 Authorization Plug-in Architecture

A fanout is an umbrella plug-in to which individual plug-in implementations are connected. As shown in the following figure, the authorization plug-in interface is implemented as a fanout.

Figure 1-5 Authorization Plug-in Architecture



The default authorization implementation consists of a fanout plug-in and a default authorization plug-in. A custom implementation consists of the fanout plug-in, the default authorization plug-in, and one or more custom authorization plug-ins.

In a fanout plug-in model, a caller sends a request to the fanout plug-in. The fanout plug-in passes the request to each of the subordinate plug-ins, and receives a response from each. Finally, the fanout plug-in forms a composite response from the individual responses, and sends the composite response to the caller.

The purpose of an authorization request is to determine whether a client operation should be allowed or whether the results of an operation should be kept *unchanged*. Each authorization plug-in returns one of three responses: *permit*, *deny*, or *abstain*. The *abstain* response gives writers of authorization plug-ins a graceful way to handle situations that are not accommodated by the original plug-in, such as names of operations that are added to the system after the plug-in is installed.

The authorization fanout plug-in forms a composite response as described in the following table. For default authorization, the composite response is determined solely by the default authorization plug-in.

Table 1-2 Authorization Composite Responses

If Plug-ins Return . . .	The Composite Response Is . . .
All *permit* or a combination of *permit* and *abstain*	`permit`
At least one *deny*	`deny`
All abstain	*deny* If the `SECURITY` parameter in the ATMI application's `UBBCONFIG` file is set to `MANDATORY_ACL` *permit* If the `SECURITY` parameter is *not* set in the ATMI application's `UBBCONFIG` file or is set to any value other than `MANDATORY_ACL`

As an example of custom authorization, consider a banking application in which a user is identified as a member of the `Customer` group, and the following conditions are in effect:

- The default authorization plug-in allows any user in the `Customer` group to withdraw money from a particular account.
- A custom authorization plug-in allows any user in the `Customer` group to withdraw money from a particular account but only on Monday through Friday between 9:00 A.M. and 5:00 P.M.

- A second custom authorization plug-in allows any user in the `Customer` group to withdraw money from a particular account but only if the amount being withdrawn is less than $10,000.

So, if a user in the `Customer` group attempts to withdraw $500.00 on Monday at 10 A.M., the operation is allowed. If the same user attempts the same withdrawal on Saturday morning, the operation is *not* allowed.

Many other custom authorization scenarios are possible. Feel free to improvise; define the conditions that best serve the needs of your business.

1.6.2 How the Authorization Plug-in Works

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

An Oracle Tuxedo system process or server (such as /Q server TMQUEUE(5) or EventBroker server TMUSREVT(5)) calls the authorization plug-in when it receives a client request. In response, the authorization plug-in performs a pre-operation check and returns whether the operation should be allowed.

- If allowed, the system carries out the client request.
- If not allowed, the system does not carry out the client request.

If the client operation is allowed, the Oracle Tuxedo system process or server may call the authorization plug-in after the client operation completes. In response, the authorization plug-in performs a post-operation check and returns whether the results of the operation are acceptable.

- If acceptable, the system accepts the operation results.
- If not unacceptable, the system either modifies the operation results or rolls back (reverses) the operation.

These calls are system-level calls, not application-level calls. An ATMI application cannot call the authorization plug-in.

The authorization process is somewhat different for (1) users of the default authorization plug-in provided by the Oracle Tuxedo system and (2) users of one or more custom authorization plug-ins. The default plug-in does not support post-operation checks. If the default authorization plug-in receives a post-operation check request, it returns immediately and does nothing.

The custom plug-ins support both pre-operation and post-operation checks.

- Default Authorization
- Custom Authorization

1.6.2.1 Default Authorization

When default authorization is called by an ATMI process to perform a pre-operation check in response to a client request, the authorization plug-in performs the following tasks.

1. Gets information from the client's authorization token by calling the authentication plug-in. Because the authorization token is created by the authentication plug-in, the authorization plug-in has no record of the token's content. This information is necessary for the authorization process.

2. Performs a pre-operation check.
   The authorization plug-in determines whether that operation should be allowed by examining the client's authorization token, the access control list (ACL), and the configured security level (optional or mandatory ACL) of the ATMI application.
3. Issues a decision about whether the operation will be performed.
   The authorization *fanout* plug-in receives a decision (*permit* or *deny*) from the default authorization plug-in and operates on its behalf.
   - If the decision is to permit the client operation, the fanout plug-in returns *permit* to the calling process. The system carries out the client request.
   - If the decision is to deny the operation, the fanout plug-in returns *deny* to the calling process. The system does not carry out the client request.

1.6.2.2 Custom Authorization

Users of one or more custom authorization plug-ins may take advantage of additional functionality offered by the ATMI environment of the Oracle Tuxedo product. Specifically, the custom plug-ins may perform an additional check after an operation occurs.

When custom authorization is called by an ATMI process to perform a pre-operation check in response to a client request, the authorization plug-in performs the following tasks.

1. Gets information from the client's authorization token by calling the authentication plug-in.
2. Performs a pre-operation check.
   The authorization plug-in determines whether the operation should be allowed by examining the operation, the client's authorization token, and associated data. "Associated data" may include user data and the security level of the ATMI application.

   If necessary, in order to satisfy authorization requirements, the authorization plug-in may modify the user data before the operation is performed.
3. Issues a decision about whether the operation will be performed.
   The authorization *fanout* plug-in makes the ultimate decision by checking the individual responses (*permit*, *deny*, *abstain*) of its subordinate plug-ins.
   - If the fanout plug-in allows the client operation, it returns *permit* to the calling process. The system carries out the client request.
   - If the fanout plug-in does not allow the operation, it returns *deny* to the calling process. The system does not carry out the client request.

If the client operation is allowed, custom authorization may be called by the ATMI process to perform a post-operation check after the client operation completes. If so, the authorization plug-in performs the following tasks.

1. Gets information from the client's authorization token by calling the authentication plug-in.
2. Performs a post-operation check.
   The authorization plug-in determines whether the operation results are acceptable by examining the operation, the client's authorization token, and associated data. "Associated data" may include user data and the security level of the ATMI application.
3. Issues a decision about whether the operation results are acceptable.
   The authorization *fanout* plug-in makes the ultimate decision by checking the individual responses (*permit*, *deny*, *abstain*) of its subordinate plug-ins.
   - If the fanout plug-in decides that the operation results are acceptable, it returns *permit* to the calling process. The system accepts the operation results.

- If the fanout plug-in does not allow the operation, it returns *deny* to the calling process. The system either modifies the operation results or rolls back (reverses) the operation.

A post-operation check is useful for label-based security models. For example, suppose that a user is authorized to access CONFIDENTIAL documents but performs an operation that retrieves a TOP SECRET document. (Often, a document's classification label is not easily determined until *after* the document has been retrieved.) In this case, the post-operation check is an efficient means to either deny the operation or modify the output data by expunging any restricted information.

1.6.3 Implementing Custom Authorization

You can provide authorization for your ATMI application by using the default plug-in or adding one or more custom plug-ins. You choose a plug-in by configuring the Oracle Tuxedo *registry*, a tool that controls all security plug-ins.

If you want to use the default authorization plug-in, you do not need to configure the registry. If you want to add one or more custom authorization plug-ins, however, you must configure the registry for your additional plug-ins before you can install them. For more detail about the registry, see Setting the Oracle Tuxedo Registry.

See Also:

- Default Authentication and Authorization
- Security Administration Tasks
- Administering Authorization
- Programming an ATMI Application with Security

1.7 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 security levels of an ATMI application. They may also be used to determine which operations were attempted, which ones failed, and which ones successfully completed.

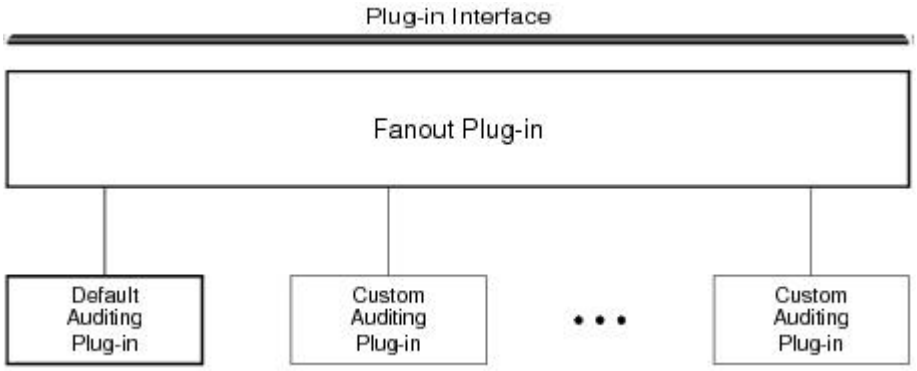
How auditing is done (that is, how information is collected, processed, protected, and distributed) depends on the auditing plug-in.

- Auditing Plug-in Architecture
- How the Auditing Plug-in Works
- Implementing Custom Auditing

1.7.1 Auditing Plug-in Architecture

A fanout is an umbrella plug-in to which individual plug-in implementations are connected. As shown in the following figure, the auditing plug-in interface is implemented as a fanout.

Figure 1-6 Auditing Plug-in Architecture



The default auditing implementation consists of a fanout plug-in and a default auditing plug-in. A custom implementation consists of the fanout plug-in, the default auditing plug-in, and one or more custom auditing plug-ins.

In a fanout plug-in model, a caller sends a request to the fanout plug-in. The fanout plug-in passes the request to each of the subordinate plug-ins, and receives a response from each. Finally, the fanout plug-in forms a composite response from the individual responses, and sends the composite response to the caller.

The purpose of an auditing request is to record an event. Each auditing plug-in returns one of two responses: *success* (the audit succeeded—logged the event) or *failure* (the audit failed—did not log the event). The auditing fanout plug-in forms a composite response in the following manner: if all responses are *success*, the composite response is *success*; otherwise, the composite response is *failure*.

For default auditing, the composite response is determined solely by the default auditing plug-in. For custom auditing, the composite response is determined by the fanout plug-in after collecting the responses of the subordinate plug-ins. For more insight into how fanouts work, see Authorization Plug-in Architecture.

1.7.2 How the Auditing Plug-in Works

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

An ATMI system process or server (such as /Q server TMQUEUE(5) or EventBroker server TMUSREVT(5)) calls the auditing plug-in when it receives a client request. Because it is called before an operation begins, the auditing plug-in can audit operation attempts and store data if that data will be needed later for a post-operation audit. In response, the auditing plug-in performs a pre-operation audit and returns whether the audit succeeded.

The ATMI system process or server may call the auditing plug-in after the client operation is performed. In response, the auditing plug-in performs a post-operation audit and returns whether the audit succeeded.

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

These calls are system-level calls, not application-level calls. An ATMI application cannot call the auditing plug-in.

The auditing process is somewhat different for (1) users of the default auditing plug-in provided by the Oracle Tuxedo system and (2) users of one or more custom auditing plug-ins. The default plug-in does not support pre-operation audits. If the default auditing plug-in receives a pre-operation audit request, it returns immediately and does nothing.

The custom plug-ins support both pre-operation and post-operation audits.

- Default Auditing
- Custom Auditing

1.7.2.1 Default Auditing

The default auditing implementation consists of the Oracle Tuxedo EventBroker component and userlog (`ULOG`). These utilities report only security violations; they do not report which operations were attempted, which ones failed, and which ones successfully completed.

When default auditing is called by an ATMI process to perform a post-operation audit when a security violation is suspected, the auditing plug-in performs the following tasks.

1. Gets information from the client's auditing token by calling the authentication plug-in. Because the auditing token is created by the authentication plug-in, the auditing plug-in has no record of the token's content. This information is necessary for the auditing process.
2. Performs a post-operation audit.
   The auditing plug-in examines the client's auditing token and the security violation delivered in the post-operation audit request.
3. Issues a decision about whether the post-operation audit succeeded.
   The auditing *fanout* plug-in receives a decision (*success* or *failure*) from the default auditing plug-in and operates on its behalf.
   - If the decision is *success*, the post-operation audit succeeded. The auditing fanout plug-in returns *success* to the calling process and logs the security violation.
   - If the decision is *failure*, the post-operation audit failed. The auditing fanout returns *failure* to the calling process.

1.7.2.2 Custom Auditing

Users of one or more custom auditing plug-ins may take advantage of additional functionality offered by the ATMI environment of the Oracle Tuxedo product. Specifically, the custom plug-ins may perform an additional audit before an operation occurs.

When custom auditing is called by an ATMI process to perform a pre-operation audit in response to a client request, the auditing plug-in performs the following tasks.

1. Gets information from the client's auditing token by calling the authentication plug-in.
2. Performs a pre-operation audit.
   The auditing plug-in examines the client's auditing token and may store user data if that data will be needed later for a post-operation audit.
3. Issues a decision about whether the pre-operation audit succeeded.
   The auditing *fanout* plug-in makes the ultimate decision by checking the individual responses (*success* or *failure*) from its subordinate plug-ins.

- If the composite decision is *success*, the pre-operation audit succeeded. The auditing fanout plug-in returns *success* to the calling process and logs the client's attempt to perform the operation.
- If the composite decision is *failure*, the pre-operation audit failed. The auditing fanout returns *failure* to the calling process.

Custom auditing may be called by the ATMI process to perform a post-operation audit after the client operation is performed. If so, the auditing plug-in performs the following tasks.

1. Gets information from the client's auditing token by calling the authentication plug-in.
2. Performs a post-operation audit.
   The auditing plug-in examines the client's auditing token, the completion status delivered in the post-operation audit request, and any data stored during the pre-operation audit.
3. Issues a decision about whether the post-operation audit succeeded.
   The auditing *fanout* plug-in decides if the post-operation audit succeeded or failed by checking the individual responses (*success* or *failure*) from its subordinate plug-ins.
   - If the composite decision is *success*, the post-operation audit succeeded. The auditing fanout plug-in returns *success* to the calling process and logs the completion status of the operation.
   - If the composite decision is *failure*, the post-operation audit failed. The auditing fanout returns *failure* to the calling process.

An operation is considered successful if it passes both pre- and post-operation audits, and the operation itself is successful. Some companies collect and store both pre- and post-operation auditing data, even though such data can occupy a lot of disk space.

1.7.3 Implementing Custom Auditing

You can provide auditing for your ATMI application by using the default plug-in or adding one or more custom plug-ins. You choose a plug-in by configuring the Oracle Tuxedo *registry*, a tool that controls all security plug-ins.

If you want to use the default auditing plug-in, you do not need to configure the registry. If you want to add one or more custom auditing plug-ins, however, you must configure the registry for your additional plug-ins before you can install them.

Now Oracle Tuxedo supports Oracle Platform Security Services (OPSS) plug-in.

1.8 Link-Level Encryption

Link-level encryption (LLE) establishes data privacy for messages moving over the network links that connect the machines in an ATMI application. 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 on the following types of ATMI application links:

- Workstation client to workstation handler (WSH)
- Bridge-to-Bridge

- Administrative utility (such as `tmboot` or `tmshutdown`) to `tlisten`
- Domain gateway to domain gateway

There are three levels of LLE security:

- 0-bit (no encryption)
- 56-bit (International)
- 128-bit (United States and Canada)

The International LLE version allows 0-bit and 56-bit encryption. The United States and Canada LLE version allows 0, 56, and 128-bit encryption.

1.8.1 How LLE Works

LLE control parameters and underlying communication protocols are different for various link types, but the setup is basically the same in all cases:

- An *initiator* process begins the communication session.
- A *target* process receives the initial connection.
- Both processes are aware of the link-level encryption feature, and have two configuration parameters.

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*) in the discussion that follows. 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.

1.8.2 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.

1.8.2.1 Determining Min-Max Values

A Tuxedo process will process the `MINENCRYTPBITS` and `MAXENCRYPTBITS` using the following steps.

- If the configured *min-max* values accommodate the default *min-max* values, then the local software assigns those values as the *min-max* values for the process.
- If one of the *min-max* values is not configured, then the default value will be used for the missing value. For instance (0, max-value-configured) or (min-value-configured, 128) will be used.
- 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 default *min-max* values as the maximum value, that is, (0, 128) for the LLE.

1.8.2.2 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.

The following table describes 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 1-3 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

1.8.3 Backward Compatibility of LLE

The ATMI environment of the Oracle Tuxedo product offers some backward compatibility for LLE.

- Interoperating with Release 6.5 Oracle Tuxedo Software
- Interoperating with Pre-Release 6.5 Oracle Tuxedo Software

1.8.3.1 Interoperating with Release 6.5 Oracle Tuxedo Software

In the following table describes which key size, if any, is agreed upon by two ATMI applications when one of them is running under release 6.5 and the other under release 7.1 or later. The header row holds the *min-max* values for the process running under release 7.1 or later; the far left column holds the *min-max* values for the process running under release 6.5.

Table 1-4 Negotiation Results When Interoperating with Release 6.5 Oracle Tuxedo Software

	(0,0)	(0,56)	(0,128)	(56,56)	(56,128)	(128,128)
		0	0	ERROR	ERROR	ERROR
		40	40	ERROR	ERROR	ERROR
		40	128	ERROR	128	128

Table 1-4 (Cont.) Negotiation Results When Interoperating with Release 6.5 Oracle Tuxedo Software

	(0,0)	(0,56)	(0,128)	(56,56)	(56,128)	(128,128)
	ERROR	40	40	ERROR	ERROR	ERROR
	ERROR	40	128	ERROR	128	128
	ERROR	ERROR	128	ERROR	128	128

If your current Oracle Tuxedo installation is configured for (0, 56), (0, 128), (56,56), or (56, 128), and you want to interoperate with a release 6.5 ATMI application that is configured for a maximum LLE level of 40 bits, then any negotiation results in an automatic upgrade to 56.

The negotiation result in this case is the same as the negotiation result for two sites running release 6.5 and configured for a maximum LLE level of 40 bits. In both scenarios, the negotiation results in an automatic upgrade to 56.

1.8.3.2 Interoperating with Pre-Release 6.5 Oracle Tuxedo Software

In the following table describes which key size, if any, is agreed upon by two ATMI applications when one of them is running under pre-release 6.5 and the other under release 7.1 or later. The header row holds the *min-max* values for the process running under release 7.1 or later; the far left column holds the *min-max* values for the process running under pre-release 6.5.

Table 1-5 Negotiation Results When Interoperating with Pre-Release 6.5 Oracle Tuxedo Software

	(0,0)	(0,56)	(0,128)	(56,56)	(56,128)	(128,128)
(0,0)	0	0	0	ERROR	ERROR	ERROR
(0,40)	0	56	56	56	56	ERROR
(0,128)	0	56	128	56	128	128
(40,40)	ERROR	56	56	56	56	ERROR
(40,128)	ERROR	56	128	56	128	128
(128,128)	ERROR	ERROR	128	ERROR	128	128

If your current Oracle Tuxedo installation is configured for (0, 56) or (0, 128), and you want to interoperate with a pre-release 6.5 ATMI applications that is configured for a maximum LLE level of 40 bits, then the result of any negotiation is 40.

If your current Oracle Tuxedo installation is configured for (56, 56), (56, 128), or (128, 128), then your system *cannot* interoperate with a pre-release 6.5 ATMI application that is configured for a maximum LLE level of 40 bits. Attempts to negotiate a common key size fail.

1.8.4 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 ATMI application not using LLE, and 60 seconds in an ATMI 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) .

See Also:

- Security Administration Tasks
- Administering Link-Level Encryption
- “Distributing ATMI Applications Across a Network” and “Creating the Configuration File for a Distributed ATMI Application” in *Setting Up an Oracle Tuxedo Application*

1.9 TLS Encryption

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

Note:

The actual network protocol used is TLS, which is the successor to the TLS protocol, however, this document follows common usage and refer to this protocol as TLS Encryption.

Like LLE, the TLS 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 TLS protocol with password authentication, you are prompted for the password of the Listener/Handler (IIOP, Workstation, or JOLT) defined by the `SEC_PRINCIPAL_NAME` parameter when you enter the `tmloadcf` command.

TLS is used to secure ATMI application links in the following methods:

- Client to server handler (IIOP, Workstation, or JOLT)
- Bridge-to-Bridge
- Administrative utility (such as `tmboot` or `tmshutdown`) to `tlisten`
- Domain gateway to domain gateway

Available TLS ciphers include 256-bit, 128-bit, and 56-bit ciphers, as described later in this chapter.

- How the TLS Protocol Works
- Requirements for Using the TLS Protocol
- TLS Version Negotiation and Configuration
- Encryption Key Size Negotiation
- Backward Compatibility of TLS
- WSL/WSH Connection Timeout During Initialization
- Supported Cipher Suites
- TLS Installation

1.9.1 How the TLS Protocol Works

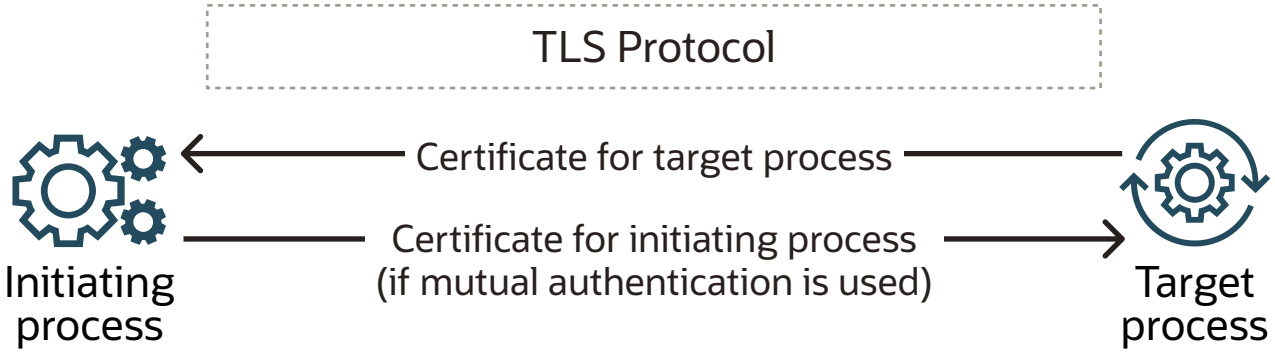
The TLS protocol works in the following manner:

1. The Target Process presents its digital certificate to the initiating application.
2. The initiating application compares the digital certificate of the Target Process against its list of trusted certificate authorities.
3. If the initiating application validates the digital certificate of the Target Process, the application and the Target Process establish an TLS connection.

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

The following figure illustrates how the TLS protocol works.

Figure 1-7 How the TLS Protocol Works in a Tuxedo Application



1.9.2 Requirements for Using the TLS Protocol

The implementation of the TLS protocol is flexible enough to fit into most public key infrastructures. Tuxedo offers two different methods to store TLS security credentials:

- The Oracle Wallet is a new feature of Tuxedo 12c. An Oracle Wallet stores the private key, certificate chain, and trusted certificates for a process within a single PKCS12 file, which can be created using either Oracle tools or tools from other security vendors.
- The plugin framework used in previous release of Tuxedo can also be used to store security credentials. The default implementation of the plug-in frame work in 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 Tuxedo application. You must have an LDAP-enabled directory service and a certificate authority in place before using the TLS protocol in a Tuxedo application.

1.9.3 TLS Version Negotiation and Configuration

Tuxedo 12.2.2 supports TLS 1.2, 1.1, and 1.0, while some Tuxedo earlier releases support just TLS 1.0. When a secure network connection is established between the TLS server and client, the TLS version to be used is negotiated. When acting as an TLS server, Tuxedo components always accept TLS1.2/1.1/1.0 initiating request. When acting as an TLS client, Tuxedo 12.2.2 components conform to following rules:

- WSC has self-adaption capability, which means it can connect to a Tuxedo 12.2.2 listener using TLS 1.2, and can connect to the old release listener using TLS 1.0 automatically.
- GWTDOMAIN, COBRA client, and GWWS outbound HTTPS use TLS 1.2 by default. You need to change their TLS version to TLS 1.0 when connecting to a Tuxedo 12.1.3 GA or earlier release.
- When GWTDOMAIN is acting as both TLS client and TLS server, only the TLS version specified for TLS client side takes effect.
- If a Tuxedo 12.2.2 master machine connects to an earlier release slave machine in an MP model, you must start tlisten on the master machine before running the `tmboot` command.

Table 1-6 Default TLS Version and Related Parameter

When an TLS client is...	The default TLS version used is...	You can change the TLS Version using...
GWTDOMAIN	TLS 1.2	The TLS version parameter in DMCONFIG. For more information, see File Formats, Data Descriptions, MIBs, and System Processes Reference.
WSC	Self-adaptive	The environment variable `WSNADDR`
CORBA client (Tobj_Bootstrap)	TLS 1.2	The Tobj_Bootstrap constructor naddress parameter or the environment variable TOBJADDR. For more information, seeFile Formats, Data Descriptions, MIBs, and System Processes Reference .
GWWS outbound	TLS 1.2	The new attribute <TLSversion> for End Point of outbound in SALT Deployment File. For more information, see Configuring a SALT Application.

1.9.4 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
- Finding a Common Key Size

1.9.4.1 Determining Min-Max Values

A Tuxedo process will process the `MINENCRYTPBITS` and `MAXENCRYPTBITS` using the following steps:

- If the configured *min-max* values accommodate the default *min-max* values, then the local software assigns those values as the *min-max* values for the process.
- If one of the *min-max* values is not configured, then the default value will be used for the missing value. For instance (0, max-value-configured) or (min-value-configured, 128) will be used.

- 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 128 as the maximum value.
- The minimum encryption key size is 112. If `min-max` value is configured with 40 or 56, then 112 will be used by default.
- The configuration information about encryption strength is processed independent of type of link level security.
- For /WS client, the default `MAXENCRYPTBITS` is 256; it will be adjusted according to the actual link level security configured.

1.9.4.2 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.

The following table describes 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 1-7 Interprocess Negotiation Results (112,112) to (112,256)

	(112,112)	**(112,128)**	**(112,256)**
(112,112)	112	112	112
(112,128)	112	128	128
(112,256)	112	128	256
(128,128)	ERROR	128	128
(128,256)	ERROR	128	256
(256,256)	ERROR	ERROR	256

Table 1-8 Interprocess Negotiation Results (128,128) to (256,256)

	(128,128)	**(128,256)**	**(256,256)**
(112,112)	ERROR	ERROR	ERROR
(112,128)	128	128	ERROR
(112,256)	128	256	256
(128,128)	128	128	ERROR
(128,256)	128	256	256
(256,256)	ERROR	256	256

1.9.5 Backward Compatibility of TLS

In order to use TLS between two Tuxedo processes, both processes must be running Tuxedo 10.0 or later (except when using the CORBA TLS capabilities described in "Using Security in CORBA Applications." It is possible to specify both non-TLS and TLS ports for WSL and JSL processes and to specify TLS or LLE connectivity for individual entries in the *DM_TDOMAIN section of a DMCONFIG file. In this way, it is possible to gradually migrate a workstation or domain application to use TLS as individual workstation clients and Tuxedo domains are upgraded to Tuxedo 10.

See Also:

- It is not possible to use TLS between BRIDGE and tlisten processes in an MP mode application until all machines in the Tuxedo domain are upgraded to Tuxedo 10.0 or later.
- Zero bit TLS ciphers (which do not actually encrypt application data) were allowed prior to Tuxedo 12.1.1, but are disallowed by the Oracle NZ Security Layer used in Tuxedo 12.1.1 and later.

1.9.6 WSL/WSH Connection Timeout During Initialization

The length of time a Workstation client can take for initialization is limited. By default, this interval is 60. The 60-second interval includes the time needed to negotiate an encrypted link. This time limit can be changed when WSL 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).

1.9.7 Supported Cipher Suites

A cipher suite is a TLS 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 ATMI security environment supports the cipher suites described in the following table.

Table 1-9 SSL/TLS Cipher Suites Supported by the ATMI Security Environment

Cipher Suite	Key Exchange Type	Symmetric Key Strength
TLS_RSA_WITH_AES_256_CBC_SHA	RSA	256
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	128
SSL_RSA_WITH_RC4_128_SHA	RSA	128
SSL_RSA_WITH_RC4_128_MD5	RSA	128
SSL_RSA_WITH_3DES_EDE_CBC_SHA SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	RSA	112
SSL_RSA_WITH_DES_CBC_SHA SSL_DH_anon_WITH_DES_CBS_SHA	RSA	56
SSL_RSA_EXPORT_WITH_RC4_40_MD5 SSL_RSA_EXPORT_WITH_DES40_DBC_SHA SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	RSA	40

1.9.8 TLS Installation

TLS is delivered as a standard feature of the Tuxedo system. If an application will not be using the Oracle Wallet to store security credentials and will be using LDAP to obtain certificates, then the administrator should have the name of their LDAP server, the LDAP port number, and the LDAP filter file location available at installation time (The default LDAP filter file location of `$TUXDIR/udataobj/security/bea_ldap_filter.dat` should be fine for most applications.)

This information can be changed after installation using the `epifregedt`command.

See Also:

- Security Administration Tasks
- Administering TLS Encryption
- "Distributing ATMI Applications Across a Network" and "Creating the Configuration File for a Distributed ATMI Application" in *Setting Up an Oracle Tuxedo Application*
- Using Security in CORBA Applications

1.10 Public Key Security

Public key security provides two capabilities that make end-to-end digital signing and data encryption possible:

- Message-based digital signature
- Message-based encryption

Message-based digital signature allows the recipient (or recipients) of a message to identify and authenticate both the sender and the sent message. Digital signature provides solid proof of the originator and content of a message; a sender cannot falsely repudiate responsibility for a message to which that sender's digital signature is attached. Thus, for example, Bob cannot issue a request for a withdrawal from his bank account and later claim that someone else issued that request.

In addition, message-based encryption protects the confidentiality of messages by ensuring that only designated recipients can decrypt and read them.

- PKCS-7 Compliant
- Supported Algorithms for Public Key Security

1.10.1 PKCS-7 Compliant

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 public key software in the ATMI environment of the Oracle Tuxedo software complies with the PKCS-7 standard.

PKCS-7 is a *hybrid cryptosystem* architecture. A *symmetric key algorithm* with a random *session key* is used to encrypt a message, and a *public key algorithm* is used to encrypt the

random session key. A random number generator creates a new session key for each communication, which makes it difficult for a would-be attacker to reuse previous communications.

1.10.2 Supported Algorithms for Public Key Security

All the algorithms on which public key security is based are well known and commercially available. To select the algorithms that will best serve your ATMI application, consider the following factors: speed, degree of security, and licensing restrictions (for example, the United States government restricts the algorithms that it allows to be exported to other countries).

- Public Key Algorithms
- Digital Signature Algorithms
- Symmetric Key Algorithms
- Message Digest Algorithms

1.10.2.1 Public Key Algorithms

The public key security in the ATMI environment of the Oracle Tuxedo product supports any public key algorithms supported by the underlying plug-ins, including RSA, ElGamal, and Rabin. (RSA stands for Rivest, Shamir, and Adelman, the inventors of the RSA algorithm.) All these algorithms can be used for digital signatures and encryption.

Public key (or *asymmetric key*) algorithms such as RSA 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.

1.10.2.2 Digital Signature Algorithms

The public key security in the ATMI environment of the Oracle Tuxedo product supports any digital signature algorithms supported by the underlying plug-ins, including RSA, ElGamal, Rabin, and Digital Signature Algorithm (DSA). With the exception of DSA, all these algorithms can be used for digital signatures and encryption. DSA can be used for digital signatures but not for encryption.

Digital signature algorithms are simply public key algorithms used to provide digital signatures. DSA is also a public key algorithm (implemented through public-private key pairs), but it can only be used to provide digital signatures, not encryption.

1.10.2.3 Symmetric Key Algorithms

Public key security supports the following three 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) and is exportable outside the United States.
- 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) and is *not* exportable outside the United States.

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 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, RC2 can be used with keys of virtually unlimited length, although the ATMI public key security restricts the key length to 128 bits.

Oracle Tuxedo customers cannot expand or modify this list of 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.

1.10.2.4 Message Digest Algorithms

Public key security supports any message digest algorithms supported by the underlying plug-ins, including MD5, SHA-1 (Secure Hash Algorithm 1), and many others. 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.

See Also:

- Message-based Digital Signature
- Message-based Encryption
- Public Key Implementation
- Security Administration Tasks
- Administering Public Key Security
- Programming an ATMI Application with Security
- Writing Security Code to Protect Data Integrity and Privacy

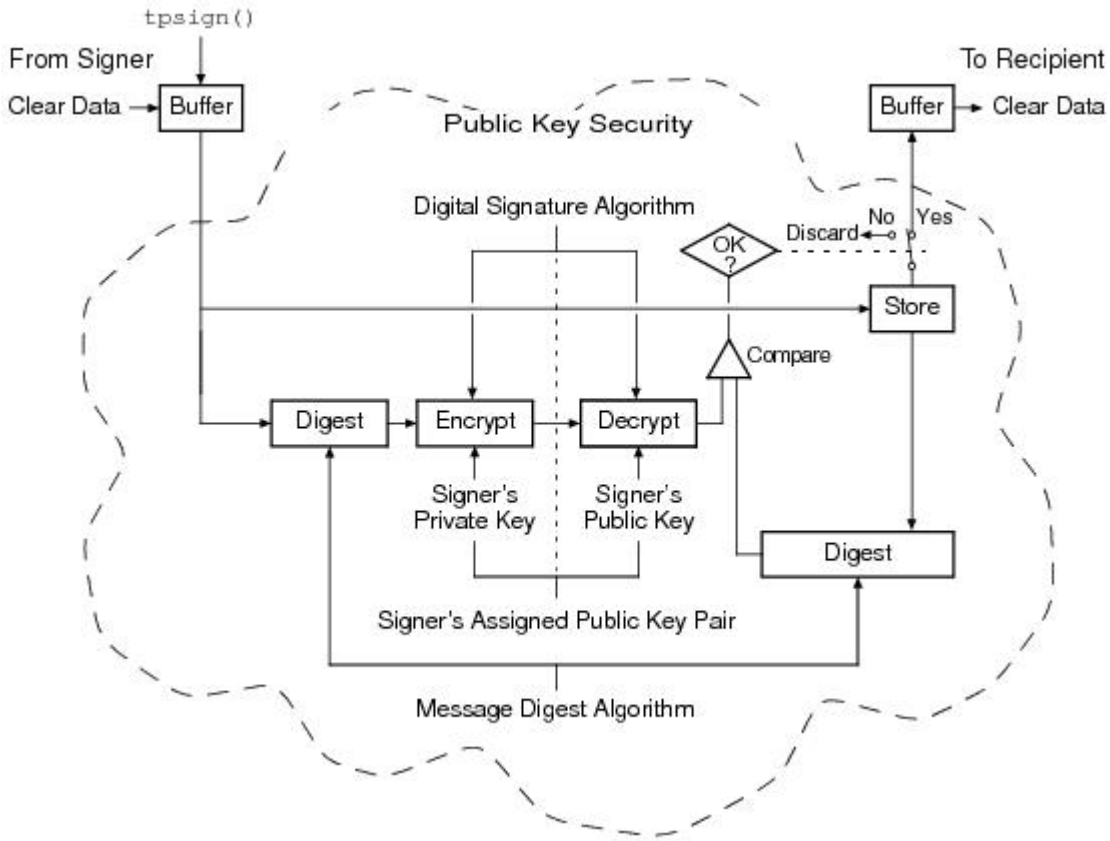
1.11 Message-based Digital Signature

Message-based digital signatures enhance ATMI security by allowing a message originator to prove its identity, and by binding that proof to a specific message buffer. Mutually authenticated and tamper-proof communication is considered essential for ATMI applications that transport data over the Internet, either between companies or between a company and the general public. It also is critical for ATMI applications deployed over insecure internal networks.

The scope of protection for a message-based digital signature is end-to-end: a message buffer is protected from the time it leaves the originating process until the time it is received at the destination process. It is protected at all intermediate transit points, including temporary message queues, disk-based queues, and system processes, and during transmission over inter-server network links.

The following figure illustrates how end-to-end message-based digital signature works.

Figure 1-8 ATMI PKCS-7 End-to-End Digital Signing



Message-based digital signature involves generating a digital signature by computing a message digest on the message, and then encrypting the message digest with the sender's private key. The recipient verifies the signature by decrypting the encrypted message digest with the signer's public key, and then comparing the recovered message digest to an independently computed message digest. The signer's public key either is contained in a *digital certificate* included in the signer information, or is referenced by an issuer-distinguished name and issuer-specific serial number that uniquely identify the certificate for the public key.

- Digital Certificates
- Certification Authority
- Certificate Repositories
- Public-Key Infrastructure

1.11.1 Digital Certificates

Digital certificates are electronic files used to uniquely identify individuals and resources over networks such as the Internet. A digital certificate securely binds the identity of an individual or resource, as verified by a trusted third party known as a *Certification Authority*, to a particular public key. Because no two public keys are ever identical, a public key can be used to identify its owner.

Digital certificates allow verification of the claim that a specific public key does in fact belong to a specific subscriber. A recipient of a certificate can use the public key listed in the certificate to verify that the 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 subscriber named in the certificate, and that the digital signature was created by that particular subscriber.

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

- The name of the subscriber (holder, owner) and other identification information required to uniquely identify the subscriber, such as the URL of the Web server using the certificate, or an individual's e-mail address.
- The subscriber's public key.
- The name of the Certification Authority that issued the certificate.
- A serial number.
- The validity period (or lifetime) of the certificate (defined by a start date and an end date).

The most widely accepted format for certificates is defined by the ITU-T X.509 international standard. Thus, certificates can be read or written by any ATMI application complying with X.509. The public key security in the ATMI environment of the Oracle Tuxedo product recognizes certificates that comply with X.509 version 3, or X.509v3.

1.11.2 Certification Authority

Certificates are issued by a Certification Authority, or CA. Any trusted third-party organization or company that is willing to vouch for the identities of those to whom it issues certificates and public keys can be a CA. When it creates a certificate, the CA signs the certificate with its private key, to obtain a digital signature. The CA then returns the certificate with the signature to the subscriber; these two parts—the certificate and the CA's signature—together form a valid certificate.

The subscriber and others can verify the issuing CA's digital signature by using the CA's public key. The CA makes its public key readily available by publicizing that key or by providing a certificate from a higher-level CA attesting to the validity of the lower-level CA's public key. The second solution gives rise to hierarchies of CAs.

The recipient of an encrypted message can develop trust in the CA's private key *recursively*, if the recipient has a certificate containing the CA's public key signed by a superior CA whom the recipient already trusts. In this sense, a 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 CAs. Through a chain of certificates, trust in a large number of users' signatures can be established.

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

> **Note:**
>
> The Oracle Tuxedo public key plug-in interface enables the customers to choose a CA of their choice.

1.11.3 Certificate Repositories

To facilitate the use of a public key in verification, a digital certificate may be published in a repository or made accessible in another manner. Repositories are databases of certificates and other information available for retrieval and use in verifying digital signatures. Retrieval can be accomplished automatically by having the verification program request certificates from the repository as required.

1.11.4 Public-Key Infrastructure

The 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 certificates; in other contexts, it embraces digital signature and encryption services provided to end-user applications as well.

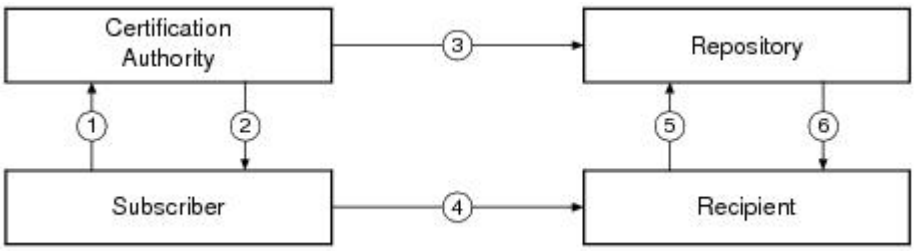
There is no single standard public key infrastructure today, though efforts are underway to define one. It is not yet clear whether a standard will be established or multiple independent PKIs evolves with varying degrees of interoperability. In this sense, the state of PKI technology today can be viewed as similar to local and wide-area 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 certificate for a public key
- Certificate revocation: for canceling a previously issued certificate
- Key selection: for obtaining a party’s public key
- Trust evaluation: for determining whether a certificate is valid and which operations it authorizes

The following figure illustrates the PKI process flow.

Figure 1-9 PKI Process Flow



1. Subscriber applies to Certification Authority (CA) for digital certificate.
2. CA verifies identity of subscriber and issues digital certificate.
3. CA publishes certificate to repository.
4. Subscriber digitally signs electronic message with private key to ensure sender authenticity, message integrity, and non-repudiation, and then sends message to recipient.
5. Recipient receives message, verifies digital signature with subscriber's public key, and goes to repository to check status and validity of subscriber's certificate.
6. Repository returns results of status check on subscriber's certificate to recipient.

> **Note:**
>
> Oracle Tuxedo enables you to utilize a PKI security solution based on PKI software from their vendor of choice through Oracle Tuxedo's public key plug-in interface.

> **See Also:**
>
> - Public Key Implementation
> - Security Administration Tasks
> - Administering Public Key Security
> - Programming an ATMI Application with Security
> - Writing Security Code to Protect Data Integrity and Privacy

1.12 Message-based Encryption

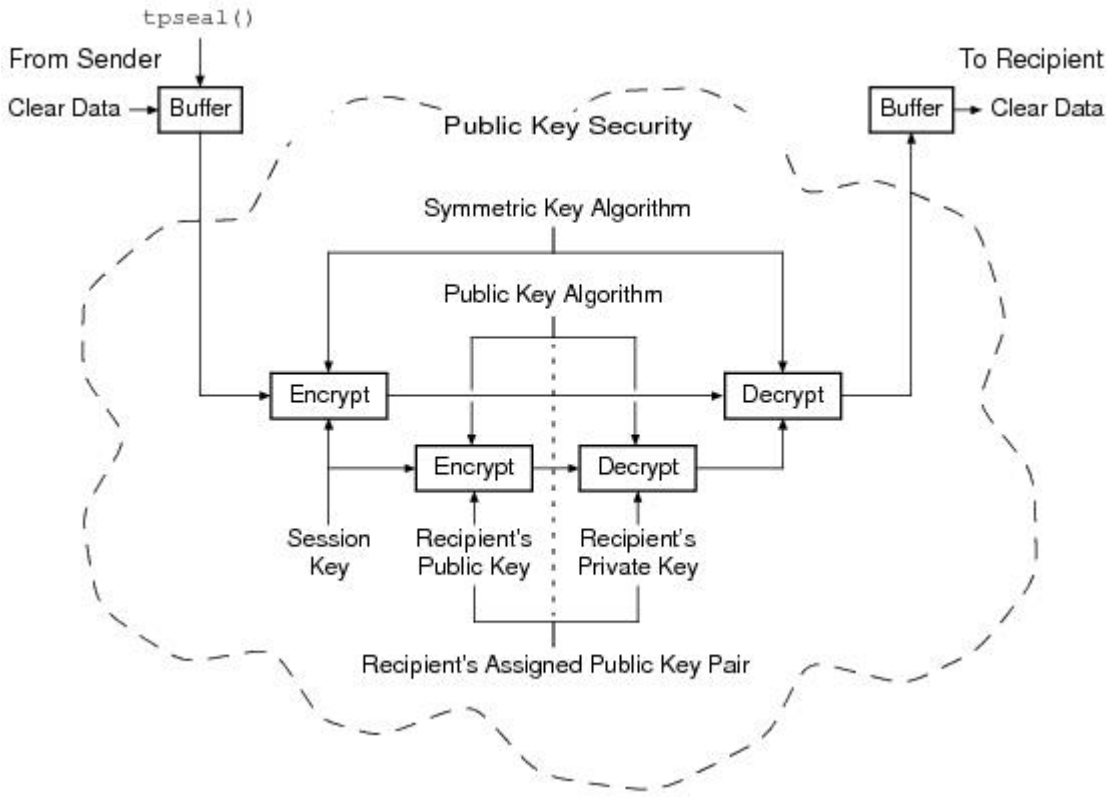
Message-based encryption keeps data private, which is essential for ATMI applications that transport data over the Internet, whether between companies or between a company and its customers. Data privacy is also critical for ATMI applications deployed over insecure internal networks.

Message-based encryption also helps ensure message integrity, because it is more difficult for an attacker to modify a message when the content is obscured.

The scope of protection provided by message-based encryption is end-to-end; a message buffer is protected from the time it leaves the originating process until the time it is received at the destination process. It is protected at all intermediate transit points, including temporary message queues, disk-based queues, and system processes, and during transmission over interserver network links.

The following figure illustrates how end-to-end message-based encryption works.

Figure 1-10 ATMI PKCS-7 End-to-End Encryption



The message is encrypted by a symmetric key algorithm and a session key. Then, the session key is encrypted by the recipient's public key. Next, the recipient decrypts the encrypted session key with the recipient's private key. Finally, the recipient decrypts the encrypted message with the session key to obtain the message content.

> **Note:**
>
> The following figure does not depict two other steps in this process: (1) the data is compressed immediately before it is encrypted; and (2) the data is uncompressed immediately after it is decrypted.

Because the unit of encryption is an ATMI message buffer, message-based encryption is compatible with all existing ATMI programming interfaces and communication paradigms. The encryption process is always the same, whether it is being performed on messages shipped between two processes in a single machine, or on messages sent between two machines through a network.

See Also:

- Public Key Implementation
- Security Administration Tasks
- Administering Public Key Security
- Programming an ATMI Application with Security
- Writing Security Code to Protect Data Integrity and Privacy

1.13 Public Key Implementation

The underlying plug-in interface for public key security consists of six component interfaces, each of which requires one or more plug-ins. By instantiating these interfaces with your preferred plug-ins, you can bring custom message-based digital signature and message-based encryption to your ATMI application.

The six component interfaces are:

- Public key initialization
- Key management
- Certificate lookup
- Certificate parsing
- Certificate validation
- Proof material mapping

1.13.1 Public Key Initialization

The public key initialization interface 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. This interface is implemented as a *fanout*.

1.13.2 Key Management

The key management interface 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.

1.13.3 Certificate Lookup

The certificate lookup interface allows public key software to retrieve X.509v3 certificates for a given *principal*. Principals are authenticated users. The certificate database may be stored using any appropriate tool, such as Lightweight Directory Access Protocol (LDAP), Microsoft Active Directory, Netware Directory Service (NDS), or local files.

1.13.4 Certificate Parsing

The certificate parsing interface allows public key software to associate a simple principal name with an X.509v3 certificate. The parser analyzes a certificate to generate a principal name to be associated with the certificate.

1.13.5 Certificate Validation

The certificate validation interface allows public key software to validate an X.509v3 certificate in accordance with specific business logic. This interface is implemented as a *fanout*, which allows Oracle Tuxedo customers to use their own business rules to determine the validity of a certificate.

1.13.6 Proof Material Mapping

The proof material mapping interface allows public key software to access the proof materials needed to open keys, provide authorization tokens, and provide auditing tokens.

1.13.7 Implementing Custom Public Key Security

You can provide public key security for your ATMI application by using custom plug-ins. You choose a plug-in by configuring the Oracle Tuxedo *registry*, a tool that controls all security plug-ins.

If you want to use custom public key plug-ins, you must configure the registry for your public key plug-ins before you can install them. For more detail about the registry, see Setting the Oracle Tuxedo Registry.

1.13.8 Default Public Key Implementation

The default public key implementation supports the following algorithms:

- Public key algorithms: RSA
- Digital signature algorithms: RSA and DSA
- Symmetric key algorithms:
  - DES-CBC
  - Two-key triple-DES
  - RC2
- Message digest algorithms:
  - MD5
  - SHA-1

See Also:

- Public Key Implementation
- Security Administration Tasks
- Administering Public Key Security
- Programming an ATMI Application with Security
- Writing Security Code to Protect Data Integrity and Privacy

1.14 Default Authentication and Authorization

The default authentication and authorization plug-ins provided by the ATMI environment of the Oracle Tuxedo product work in the same manner that implementations of authentication and authorization have worked since they were first made available with the Oracle Tuxedo system.

An application administrator can use the default authentication and authorization plug-ins to configure an ATMI application with one of five levels of security. The five levels include:

- No authentication
- Application password security
- User-level authentication
- Optional access control list (ACL) security
- Mandatory ACL security

At the lowest level, no authentication is provided. At the highest level, an access control checking feature determines which users can execute a service, post an event, or enqueue (or dequeue) a message on an application queue. The security levels are briefly described in the following table:

Table 1-10 Security Levels for Default Authentication and Authorization

Security Level	Description
No authentication	Clients do not have to be verified before joining the ATMI application. When joining an ATMI application at this security level, a user has access to all application resources.
Application password	The application administrator defines a single password for the entire ATMI application, and clients must provide the password to join the application. When successfully joining an ATMI application at this security level, a user has access to all application resources.
User-level authentication	In addition to the application password, each client must provide a valid username and user-specific data, such as a password, to join the ATMI application. When successfully joining an ATMI application at this security level, a user has access to all application resources.

Table 1-10 (Cont.) Security Levels for Default Authentication and Authorization

Security Level	Description
Optional ACL security	Clients must provide the application password, a username, and user-specific data such as a password. For a user who successfully joins an ATMI application at this security level, access to application resources is restricted in the following way. The ACL database contains a list of application resources and, for each resource, a list of users with permission to use it. A user who is not included in the list for a particular resource is not allowed to access that resource, regardless of whether optional ACL or mandatory ACL security is being used. If there is no entry in the ACL database for a resource and the security level for the ATMI application is set to optional ACL security, all users are permitted to access the resource.
Mandatory ACL security	Clients must provide the application password, a username, and user-specific data such as a password. For a user who successfully joins an ATMI application at this security level, access to application resources is restricted in the following way. The ACL database contains a list of application resources and, for each resource, a list of users with permission to use it. A user who is not included in the list for a particular resource is not allowed to access that resource, regardless of whether optional ACL or mandatory ACL security is being used. If there is no entry in the ACL database for a resource and the security level for the ATMI application is set to mandatory ACL security, users are not permitted to access the resource.

> **Note:**
>
> The term *client* is synonymous with *client process*, meaning a specific instance of a client program in execution. An ATMI client program can exist in active memory in any number of individual instances.

An application administrator can designate a security level by setting the `SECURITY` parameter in the `UBBCONFIG` configuration file to the appropriate value.

For This Security Level	Set SECURITY Parameter to . . .
No authentication	NONE
Application password security	APP_PW
User-level authentication	USER_AUTH
Optional ACL security	ACL
Mandatory ACL security	MANDATORY_ACL

The default is `NONE`. If `SECURITY` is set to `USER_AUTH`, `ACL`, or `MANDATORY_ACL`, then the application administrator must configure a system-supplied authentication server named `AUTHSVR`. `AUTHSVR` performs per-user authentication.

An application developer can replace `AUTHSVR` with an authentication server that has logic specific to the ATMI application. For example, a company may want to develop a custom authentication server so that it can use the popular Kerberos mechanism for authentication.

- Client Naming
- User, Group, and ACL Files
- Optional and Mandatory ACLs

1.14.1 Client Naming

Upon joining an ATMI application, a client process has two names: a combined user-client name and a unique client identifier known as an *application key*.

- The user-client name consists of a *username* and a *client name* and is used for security, administration, and communications.
- The application key is a 32-bit value that is called on behalf of the client and used by the access control checking feature.

Two client names are reserved for special semantics: `tpsysadm` and `tpsysop`. `tpsysadm` is treated as the application administrator, and `tpsysop` is treated as the application operator.

- User-Client Names
- Application Key

1.14.1.1 User-Client Names

When an authenticated client joins an ATMI application, it passes a username and client name to tpinit(3c) in a `TPINIT` buffer if the application is written in C, or to TPINITIALIZE(3cbl) in a `TPINFDEF-REC` record if the application is written in COBOL. The username and client name, as well as other security-related fields in the `TPINIT` buffer/ `TPINFDEF-REC` record, are described in the following table:

Table 1-11 Security-Related Fields in TPINIT Buffer/ TPINFDEF-REC Record

TPINIT	TPINFDEF-REC	Description
`usrname`	`USRNAME`	A user name consisting of a string of up to 30 characters. Required for security level `USER_AUTH`, `ACL`, or `MANDATORY_ACL`. The username represents the caller.
`cltname`	`CLTNAME`	A client name consisting of a string of up to 30 characters. Required for security level `USER_AUTH`, `ACL`, or `MANDATORY_ACL`. The client name represents the client program.
`passwd`	`PASSWD`	Application password. Required for security level `APP_PW`, `USER_AUTH`, `ACL`, or `MANDATORY_ACL`. `tpinit()` or `TPINITIALIZE()` validates this password by comparing it to the configured application password stored in the `TUXCONFIG` file.*
`datalen`	`DATALEN`	Length of the user-specific data** that follows.

Table 1-11 (Cont.) Security-Related Fields in TPINIT Buffer/ TPINFDEF-REC Record

TPINIT	TPINFDEF-REC	Description
data	N/A	User-specific data.** Required for security level USER_AUTH, ACL, or MANDATORY_ACL. tpinit() or TPINITIALIZE() forwards the user-specific data to the authentication server for validation. The authentication server is AUTHSVR.
*The binary equivalent of the UBBCONFIG file. **Usually a user password.		

For an authenticated security level (*USER_AUTH*, *ACL*, or *MANDATORY_ACL*), the username, client name, and user-specific data are transferred to *AUTHSVR* without interpretation by the Oracle Tuxedo system. The only manipulation of this information is its encryption when transmitted over the network from a Workstation client.

1.14.1.2 Application Key

Every time a client joins an ATMI application, it is assigned a 32-bit application key by the Oracle Tuxedo system. The client cannot reset the key other than by terminating its association and joining the ATMI application as a different user.

The assigned application key is the client's security credential. The client provides its application key with every service invocation as part of the *TPSVCINFO* structure in the *appkey* field. (See tpservice(3c) in the *Oracle Tuxedo ATMI C Function Reference* for more information about *TPSVCINFO*.)

The following table illustrates how the application key is set for various security levels and clients. All application key assignments are hardcoded except the last item in the table.

Table 1-12 Application Key Assignments

At This Security Level	Messages of This Type	Are Assigned the Following Application Key
Any security level	Messages from native ATMI clients that must be run by the administrator (like tmadmin(1))	0x80000000 (Application key of the administrator)
NONE or APP_PW	Messages from native ATMI clients that call tpinit()/ TPINITIALIZE() with a client name of tpsysadm and are run by the administrator	0x80000000 (Application key of the administrator)
	Messages from native ATMI clients that call tpinit()/ TPINITIALIZE() with a client name of tpsysop and are run by the administrator	0xC0000000 (Application key of the operator)
	Messages from any ATMI client other than tpsysadm or tpsysop	-1
USER_AUTH, ACL, or MANDATORY_ACL	Messages from native ATMI clients that call tpinit()/ TPINITIALIZE() with a client name of tpsysadm and are run by the administrator and *bypass authentication*	0x80000000 (Application key of the administrator)

Table 1-12 (Cont.) Application Key Assignments

At This Security Level	Messages of This Type	Are Assigned the Following Application Key
	Messages from *authenticated* ATMI clients that call `tpinit()`/ `TPINITIALIZE()` with a client name of `tpsysadm`	0x80000000 (Application key of the administrator)
	Messages from *authenticated* ATMI clients that call `tpinit()`/ `TPINITIALIZE()` with a client name of `tpsysop`	0xC0000000 (Application key of the operator)
	Messages from authenticated ATMI clients that call `tpinit()`/ `TPINITIALIZE()` with a client name other than `tpsysadm` or `tpsysop`	Application key `=user identifier` (UID) in the lower 17 bits and `group identifier` (GID) in the next higher 14 bits; remaining upper bit is 0. `AUTHSVR` returns this application key value

In addition, any message that originates from tpsvrinit(3c) or tpsvrdone(3c) in a C program (TPSVRINIT(3cbl) or TPSVRDONE(3cbl) in COBOL) is assigned the application key of the administrator: 0x80000000. The application key of the client is assigned to messages that pass through a server but originate at a client; an exception to this rule is described in Replacing Client Tokens with Server Tokens.

A user identifier (UID) is an integer, between 0 and 128K, that is used by the application to refer to a particular user. A group identifier (GID) is an integer, between 0 and 16K, that is used by the application to refer to an application group.

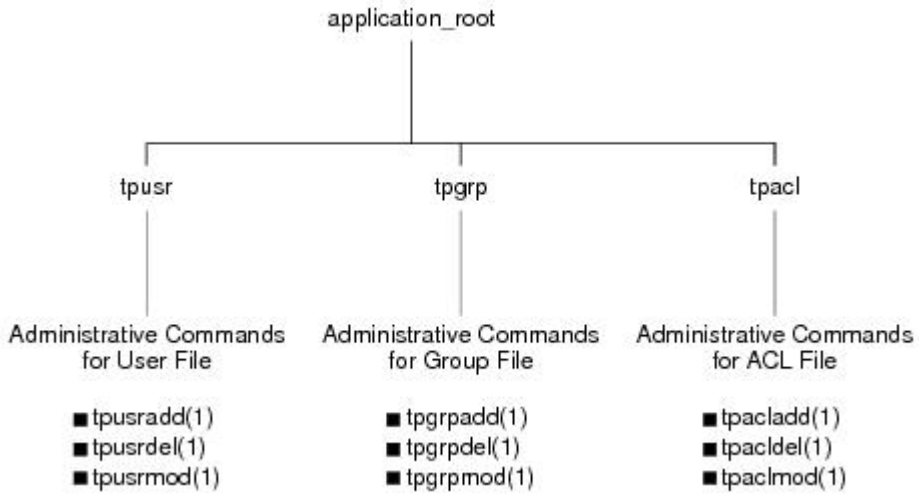
1.14.2 User, Group, and ACL Files

To use access control, an application administrator must maintain lists of (1) users, (2) groups, and (3) and mappings of groups to application entities (such as services, events, and application queues). The third type of list, a mapping of groups to application entities, is known as the access control list (ACL).

When a client tries to access an application resource, such as a service, the system checks the client's application key and thus identifies the group to which the user belongs. Next, the system checks the ACL for the target resource and determines whether the client's group has access permission. The application administrator, application operator, and processes or service requests running with the privileges of the application administrator or operator are *not* subject to ACL permission checking.

The user, group, and ACL files are located in the *application_root* directory, where *application_root* is the first pathname defined for the `APPDIR` variable. The following figure illustrates these files and specifies the administrative commands available for controlling each list.

Figure 1-11 Default User, Group, and ACL Files



> **Note:**
>
> For an ATMI application running on the Compaq VMS operating system, the names of the user, group, and ACL files have `.dat` extensions: `tpusr.dat`, `tpgrp.dat`, and `tpacl.dat`.

The files are colon-delimited, flat text files that can be read and written only by the application administrator—the owner of the `TUXCONFIG` file referenced by the `TUXCONFIG` variable. The format of the files is irrelevant, since the files are fully administered with a set of dedicated commands. Only the application administrator is allowed to use these commands.

An application administrator can use the tpaclcvt(1) command to convert security data files to the format needed by the ACL checking feature. For example, on a UNIX host machine, an administrator can use `tpaclcvt` to convert the `/etc/password` file and store the converted version in the `tpusr` file. The same administrator can use `tpaclcvt` to convert the `/etc/group` file and store the converted version in the `tpgrp` file.

The `AUTHSVR` server uses the user information stored in the `tpusr` file to authenticate users who want to join the ATMI application.

When extensible security administration is enabled with the default `XAUTHSVR` implemented, user, group, and ACL definition are placed in the LDAP repository rather than in a plain text. These information must follow the LDAP schemas. For information about LDAP schemas, refer to How to Enable The Extended Security in Administering Security.

The `XAUTHSVR` server uses the user, group, and permission information in the LDAP repository to authenticate users who want to join the ATMI application or access Tuxedo resources.

1.14.3 Optional and Mandatory ACLs

The `ACL` and `MANDATORY_ACL` security levels constitute the default authorization implementation for the ATMI environment in the Oracle Tuxedo product.

When the security level is `ACL`, if there is no entry in the `tpacl` file or LDAP `Orcljaznpermission` class associated with the target application entity, the client is permitted to access the entity. This security level enables an administrator to configure access for only those resources that need more security. That is, there is no need to add entries to the `tpacl` file for services, events, or application queues that are open to everyone.

When the security level is `MANDATORY_ACL`, if there is no entry in the `tpacl` file or LDAP `Orcljaznpermission` class associated with the target application entity, the client is *not* permitted to access the entity. For this reason, this level is called *mandatory*. There must be an entry in the `tpacl` file or LDAP `Orcljaznpermission` class for each and every application entity that the client needs to access.

For both the `ACL` and `MANDATORY_ACL` security levels, if an entry for an application entity exists in the `tpacl` file or LDAP `Orcljaznpermission` class and the client attempts to access that entity, the user associated with that client *must* be a member of a group that is allowed to access that entity; otherwise, permission is denied.

For some ATMI applications, it may be necessary to use both system-level and application-level authorization. An entry in the `tpacl` file can be used to control which users can access a service, and application logic can control data-dependent access, for example, which users can handle transactions for more than a million dollars.

Note that there is no ACL permission checking for administrative services, events, and application queues with names that begin with a dot (.). For example, any client can subscribe to administrative events such as `.SysMachineBroadcast`, `.SysNetworkConfig`, and `.SysServerCleaning`. In addition, there is no ACL permission checking for the application administrator, application operator, or processes or service requests running with the privileges of the application administrator or operator.

See Also:

- What Administering Security Means
- Security Administration Tasks
- Administering Authentication
- Administering Authorization
- What Programming Security Means
- Programming an ATMI Application with Security
- Writing Security Code So Client Programs Can Join the ATMI Application
- “About the Configuration File” and “Creating the Configuration File” in *Setting Up an Oracle Tuxedo Application*
- UBBCONFIG(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*
- AUTHSVR(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

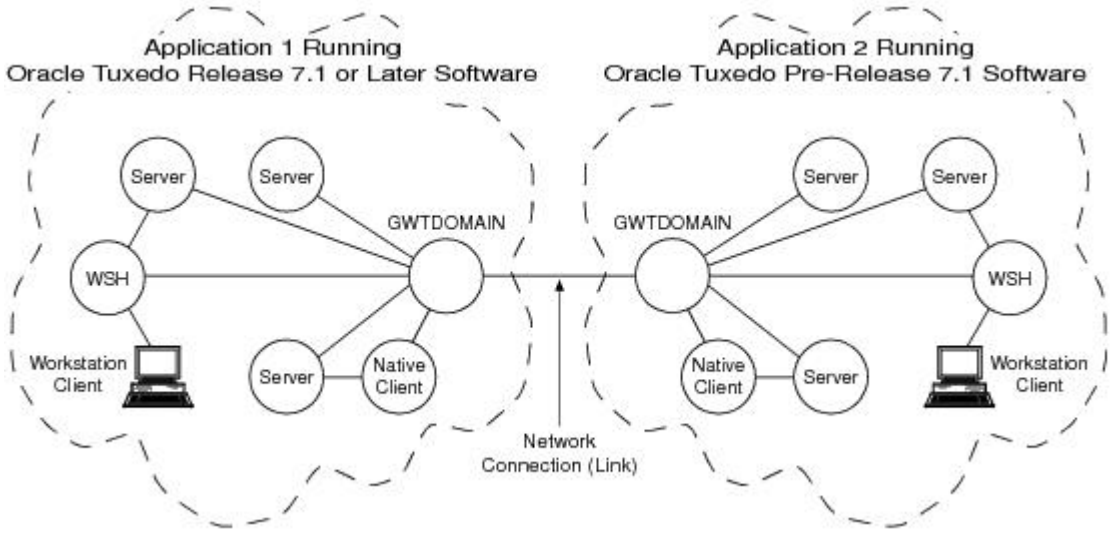
1.15 Security Interoperability

Application developers and administrators must be aware of certain security issues when configuring ATMI applications to interoperate with Oracle Tuxedo pre-release 7.1 (6.5 or earlier) software.

Interoperability, as defined in this discussion, is the ability of the current release of Oracle Tuxedo software to communicate over a network with a previous release of Oracle Tuxedo software. Specifically, inter-domain interoperability and intra-domain interoperability have the following meanings:

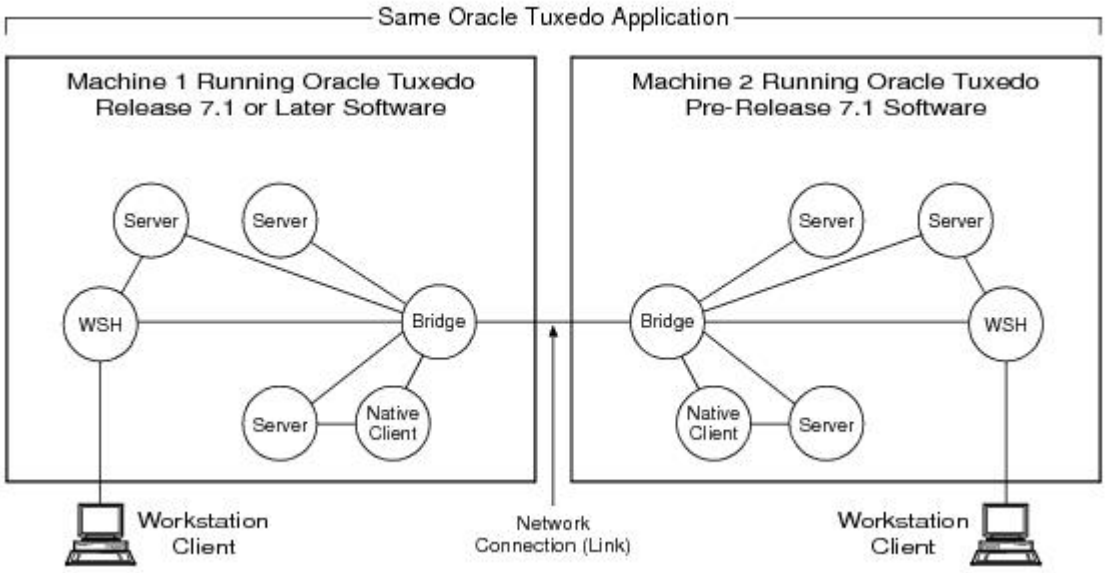
- Inter-domain interoperability
  Involves one ATMI application running Oracle Tuxedo release 7.1 or later software, and another ATMI application running Oracle Tuxedo pre-release 7.1 software.

Figure 1-12 Inter-Domain Interoperability



- Intra-domain interoperability
  Involves one machine in a multiple-machine ATMI application running Oracle Tuxedo release 7.1 or later software, and another machine in the same application running Oracle Tuxedo pre-release 7.1 software.

Figure 1-13 Intra-Domain Interoperability



- Interoperating with Pre-Release 7.1 Software
- Interoperability for Link-Level Encryption
- Interoperability for TLS Encryption
- Interoperability for Public Key Security

1.15.1 Interoperating with Pre-Release 7.1 Software

Interoperating with Oracle Tuxedo pre-release 7.1 software is allowed or disallowed at the *authentication* security level. Authentication, as implemented by Oracle Tuxedo release 7.1 or later software, allows communicating processes to mutually prove their identities.

By default, interoperability with a machine running Oracle Tuxedo pre-release 7.1 software is not allowed. To change the default, an application administrator can use the `CLOPT -t` option to allow workstation handlers (WSHs), domain gateways (`GWTDOMAINs`), and servers in the release 7.1 or later ATMI application to interoperate with Oracle Tuxedo pre-release 7.1 software. Mandating Interoperability Policy provides instructions for using the `CLOPT -t` option as well as the security ramifications for authentication and authorization when using `CLOPT -t`.

1.15.2 Interoperability for Link-Level Encryption

Whenever a network link is established between machines running Oracle Tuxedo software, link-level encryption may be used to encrypt data before sending it over the network link, and decrypt it as it comes off the link. Of course, link-level encryption is possible only if LLE is installed on both the sending and receiving machines.

LLE interoperability with Oracle Tuxedo pre-release 7.1 software is described in Backward Compatibility of LLE.

1.15.3 Interoperability for TLS Encryption

TLS encryption can be used over network links between machines running Oracle Tuxedo software only if both machines are running Tuxedo 10.0 or later. LLE encryption can be used over network links to machines running earlier releases of Tuxedo.

> **Note:**
>
> The only exception to the TLS encryption interoperability rules is that the CORBA related TLS capabilities described in "*Using Security in CORBA Applications*" can be used when interoperating with Tuxedo 8.0 and above, and when interoperating with the former WLE product.

1.15.4 Interoperability for Public Key Security

The following interoperability rules for public key security shown in the following table apply to a machine running release 7.1 or later Oracle Tuxedo software that is configured to interoperate with a machine running Oracle Tuxedo pre-release 7.1 software. To clarify the rules, each rule has an accompanying example scenario involving a Workstation client running Oracle Tuxedo pre-release 7.1 software.

Table 1-13 Interoperability Rules for Public Key Security

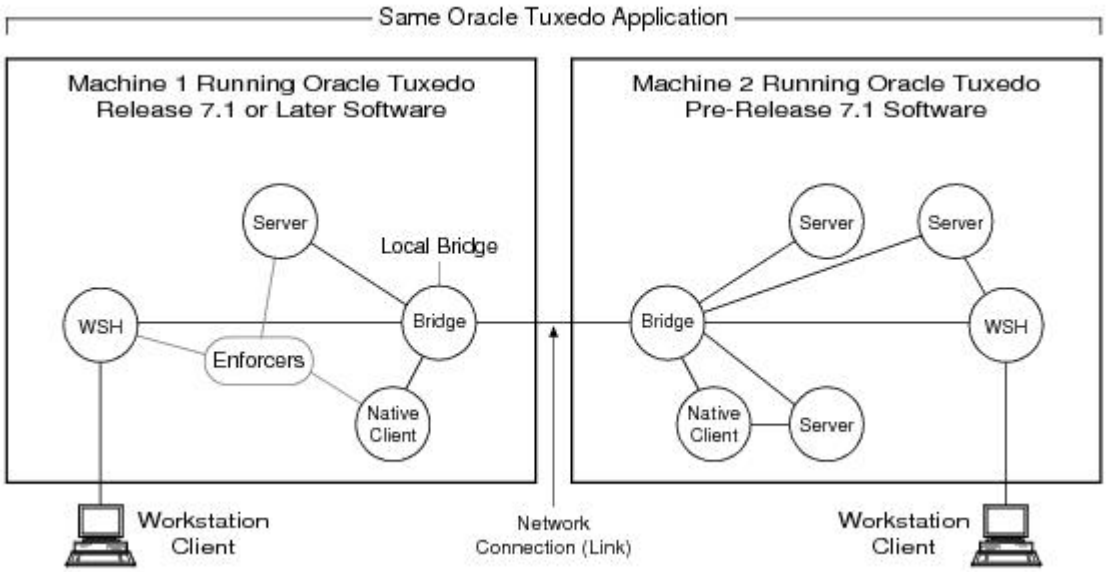
Interoperability Rule	Example	Comments
Encrypted outgoing message buffers destined for a machine running Oracle Tuxedo pre-release 7.1 software are not transmitted to the machine.	Encrypted outgoing message buffers destined for a pre-release 7.1 Workstation client are not transmitted to the Workstation client.	"Encrypted" refers to public key message-based encryption, not link-level encryption.
Incoming message buffers from a machine running an Oracle Tuxedo pre-release 7.1 software are not accepted if routed to a process requiring encryption.	Incoming message buffers from a pre-release 7.1 Workstation client do not have encryption envelopes attached, and are not accepted if routed to a process requiring encryption.	See Setting Encryption Policy for a description of the `ENCRYPTION_REQUIRED` configuration parameter.
For outgoing message buffers destined for the machine running Oracle Tuxedo pre-release 7.1 software, any digital signatures are verified and then removed before the message buffers are transmitted to the older machine.	Digital signatures are verified and then removed from outgoing message buffers destined for a pre-release 7.1 Workstation client.	It is assumed that the outgoing message buffer is digitally signed but not encrypted. If the outgoing message buffer is digitally signed and encrypted, the message is not decrypted, the digital signatures are not verified, and the message is not transmitted to the older machine.
Incoming message buffers from a machine running Oracle Tuxedo pre-release 7.1 software are not accepted if routed to a process requiring digital signatures.	Incoming message buffers from a pre-release 7.1 Workstation client do not have digital signatures attached, and are not accepted if routed to a process requiring digital signatures.	See Setting Digital Signature Policy for a description of the `SIGNATURE_REQUIRED` configuration parameter.

For inter-domain interoperability, release 7.1 or later domain gateway (`GWTDOMAIN`) processes enforce the interoperability rules for public key security.

For intra-domain interoperability, release 7.1 or later native clients, workstation handlers (WSHs), or server processes communicating with the local bridge process enforce the interoperability rules for public key security, as shown in

A bridge process operates only as a conduit; it does not decrypt message buffer content or verify digital signatures.

Figure 1-14 Enforcing Intra-Domain Interoperability Rules for Public Key Security



> **Note:**
>
> Typically, a release 7.1 or later WSH does not verify digital signatures. But when routing a digitally signed message buffer to a process running Oracle Tuxedo pre-release 7.1 software, the WSH verifies any digital signatures before removing them.

> **See Also:**
>
> - Security Compatibility
> - Mandating Interoperability Policy
> - Setting Digital Signature Policy
> - Setting Encryption Policy

1.16 Security Compatibility

For an ATMI application running Oracle Tuxedo release 7.1 or later software, it is possible to have any combination of default or custom authentication, authorization, auditing, and public key security. In addition, any combination of these four security capabilities is compatible with link-level encryption.

- Mixing Default/Custom Authentication and Authorization

- Mixing Default/Custom Authentication and Auditing
- Compatibility Issues for Public Key Security

1.16.1 Mixing Default/Custom Authentication and Authorization

It is possible to have default authentication and custom authorization, or custom authentication and default authorization, as long as the application developer is aware of the following restriction: the *authorization security token* must carry at a minimum (1) an authenticated username, or *principal name*, and (2) an application key value as defined in Application Key.

Authorization decisions are based partly on user identity, which is stored in an *authorization token*. Because authorization tokens are generated by the authentication security plug-in, providers of authentication and authorization plug-ins need to ensure that these plug-ins work together. (See Authentication and Authorization for more detail.)

1.16.2 Mixing Default/Custom Authentication and Auditing

It is possible to have default authentication and custom auditing, or custom authentication and default auditing, as long as the application developer is aware of the following restriction: the auditing security token must carry at a minimum (1) an authenticated username, or principal name, and (2) an application key value as defined in Application Key.

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

1.16.3 Compatibility Issues for Public Key Security

Public key security is compatible with all features and processes supported by Oracle Tuxedo release 7.1 or later software except the compression feature. Encrypted message buffers cannot be compressed using the compression feature. But, because the public key software compresses the message content just before it encrypts the message buffer, any size savings are still achieved.

This topic describes the compatibility/interaction of public key security with the following ATMI features and processes:

- Data-dependent routing
- Threads
- EventBroker
- /Q
- Transactions
- Domain gateways (`GWTDOMAINs`)
- Other vendors' gateways
- Compatibility/Interaction with Data-dependent Routing
- Compatibility/Interaction with Threads
- Compatibility/Interaction with the EventBroker
- Compatibility/Interaction with /Q
- Compatibility/Interaction with Transactions

- Compatibility/Interaction with Domain Gateways
- Compatibility/Interaction with Other Vendors' Gateways

1.16.3.1 Compatibility/Interaction with Data-dependent Routing

Central to the data-dependent routing feature is the ability of a process to examine the content of incoming message buffers. If an incoming message buffer is encrypted, a process configured for data-dependent routing must have opened a recipient's private key so that the public key software can use that key to decrypt the message buffer. For data-dependent routing, the public key software does *not* verify digital signatures.

If a decryption key is *not* available, the routing operation fails. The system generates an ERROR userlog(3c) message to report the failure.

If a decryption key is available, the process makes a routing decision based on a decrypted *copy* of the encrypted message buffer. The chain of events is as follows:

1. The public key software makes a copy of the encrypted message buffer and uses the decryption key to decrypt the buffer.
2. The process reads the resulting *plaintext* (unencrypted text) message content to make the routing decision.
3. The public key software overwrites the plaintext message content with zero values to preserve privacy.

The system then transmits the original encrypted message buffer in accordance with the routing decision.

1.16.3.2 Compatibility/Interaction with Threads

Public-private keys are represented and manipulated via *handles*. A handle has data associated with it that is used by the public key application programming interface (API) to locate or access the item named by the handle. A process opens a *key handle* for digital signature generation, message encryption, or message decryption.

A key handle is a process resource; it is not bound to any specific thread or context. Any communication necessary to open a key is performed within the thread's currently active context. Thereafter, the key is available to any context in the process, whether or not the context is associated with the same ATMI application.

A key's internal data structures are *thread safe*. As such, a key may be accessed concurrently by multiple threads.

1.16.3.3 Compatibility/Interaction with the EventBroker

In general, a TMUSREVT(5) system server handles encrypted message buffers without decrypting them, that is, both digital signatures and encryption envelopes remain intact as messages flow through the Oracle Tuxedo EventBroker component. However, the following cases require that the EventBroker component decrypt posted message buffers:

- To evaluate subscription filter expressions based on message content.
  If the EventBroker does not have access to a suitable decryption key, the subscription's filter expression is assumed to be false, and the subscription is not considered a *match*.
- To perform subscription notification actions that require access to message content: userlog(3c) processing or system command execution.

If the EventBroker does not have access to a suitable decryption key, the subscription's notification action fails, and the system generates an ERROR userlog(3c) message to report the failure.

- To perform subscription notification actions that, based on system configurations, need to access message content for data-dependent routing.
  If the EventBroker does not have access to a suitable decryption key, the subscription's notification action fails, and the system generates an ERROR userlog(3c) message to report the failure.

  For a transactional subscription, the system also marks the transaction as *rollback-only*.
- To comply with an administrative system policy requiring encryption (as explained in Setting Encryption Policy).
  If the EventBroker does not have access to a suitable decryption key, the tppost(3c) operation fails, and the system generates an ERROR userlog(3c) message to report the failure.
- To verify that a posted encrypted message has a valid digital signature attached, if required to do so by an administrative system policy requiring digital signatures (as explained in Setting Digital Signature Policy).
  If the EventBroker does not have access to a suitable decryption key, the tppost(3c) operation fails, and the system generates an ERROR userlog(3c) message to report the failure.

1.16.3.4 Compatibility/Interaction with /Q

In general, a TMQUEUE(5) or TMQFORWARD(5) system server handles encrypted message buffers without decrypting them, that is, both signatures and encryption envelopes remain intact as messages flow through the Oracle Tuxedo /Q component. However, the following cases require that the /Q component decrypt enqueued message buffers:

- To perform `TMQFORWARD` operations that, based on system configurations, need to access message content for data-dependent routing.
  If `TMQFORWARD` does not have access to a suitable decryption key, the forward operation fails. The system returns the message to the queue and generates an ERROR userlog(3c) message to report the failure

  After a number of periodic retry attempts, `TMQFORWARD` might place the unreadable message on an error queue.

  After a number of periodic retry attempts, `TMQFORWARD` might place the unreadable message on an error queue.
- To comply with an administrative system policy requiring encryption (as explained in Setting Encryption Policy).
  If the /Q component does not have access to a suitable decryption key, the tpdequeue(3c) operation fails, and the system generates an ERROR userlog(3c) message to report the failure.
- To verify that an enqueued encrypted message has a valid signature attached, if required to do so by an administrative system policy requiring digital signatures (as explained in Setting Digital Signature Policy).
  If the /Q component does not have access to a suitable decryption key, the tpdequeue(3c) operation fails, and the system generates an ERROR userlog(3c) message to report the failure.

A non-transactional tpdequeue(3c) operation has the side effect of destroying an encrypted queued message if the invoking process does not hold a valid decryption key.

If a message with an invalid signature is placed in a queue (or if the message is corrupted or tampered with while on the queue), any attempt to dequeue it fails. A non-transactional tpdequeue(3c) operation has the side effect of destroying such a message. A transactional tpdequeue(3c) operation causes transaction rollback, and all future transactional attempts to dequeue the message will continue to fail.

1.16.3.5 Compatibility/Interaction with Transactions

Public key security operations—opening and closing keys, requesting a digital signature, or requesting encryption—are not transactional, and are not undone by transaction rollback. However, transactions might rollback due to failure conditions associated with the following public key operations:

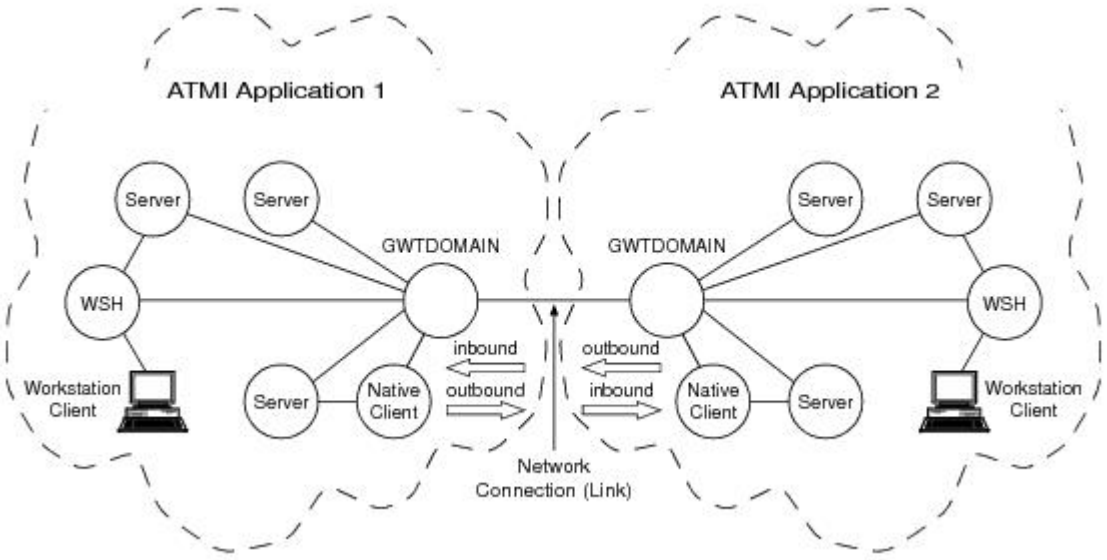
- If a transactional request or reply message cannot be decrypted, its associated transaction is rolled back.
- If a transactional request or reply message is discarded because of an invalid or missing digital signature, its associated transaction is rolled back.
- If a transactional request or reply message is rejected because it violates an administrative system policy requiring encryption or digital signatures, its associated transaction is rolled back.

1.16.3.6 Compatibility/Interaction with Domain Gateways

Domain gateway (`GWTDOMAIN`) processes connecting two ATMI applications running Oracle Tuxedo release 7.1 or later software preserve digital signatures and encryption envelopes. In addition, the domain gateway processes verify digital signatures and enforce administrative system policies regarding digital signatures and encryption.

The following figure illustrates understanding how domain gateway processes interact with local and remote ATMI applications.

Figure 1-15 Communication Between ATMI Applications



The following table describes how release 7.1 or later domain gateway processes handle digitally signed and encrypted message buffers.

Table 1-14 Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes

Message Type	Condition	Resulting Operation
Inbound message—originating from a remote process and received over a network connection	Has encryption envelope and may or may not have digital signature	The domain gateway process accepts the message and forwards it in encrypted form. If the data-dependent routing feature applies and the domain gateway process does not have a suitable decryption key, the gateway process rejects the message. (See Compatibility/Interaction with Data-dependent Routing for clarification.)
Inbound message	Does not have encryption envelope or digital signature	If the domain gateway process is running within a domain, machine, or group requiring encryption, the gateway process rejects the message. If a service advertised by the domain gateway requires encryption, the gateway process rejects the message. (See Setting Encryption Policy for clarification). If the domain gateway does not require encryption, the gateway process accepts and forwards the message.
Inbound message	Has digital signature but is not encrypted	The domain gateway process verifies the digital signature and forwards the message with digital signature attached.
Inbound message	Does not have digital signature and is not encrypted	If the domain gateway process is running within a domain, machine, or group requiring digital signatures, the gateway process rejects the message. If a service advertised by the domain gateway requires digital signatures, the gateway process rejects the message. (See Setting Digital Signature Policy for clarification.) If the domain gateway does not require digital signatures, the gateway process accepts and forwards the message.
Outbound message—originating from a local process and transmitted over a network connection	Has encryption envelope and may or may not have digital signature	The domain gateway process accepts the message and forwards it in encrypted form over the network. If the data-dependent routing feature applies and the domain gateway process does not have a suitable decryption key, the gateway process rejects the message. (See Compatibility/Interaction with Data-dependent Routing for clarification.) If the encrypted message is destined for a process running Oracle Tuxedo pre-release 7.1 (6.5 or earlier) software, the domain gateway process rejects the message. (See Interoperating with Pre-Release 7.1 Software and Interoperability for Public Key Security for clarification.)

Table 1-14 (Cont.) Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes

Message Type	Condition	Resulting Operation
Outbound message	Does not have encryption envelope or digital signature	If the domain gateway process is running within a domain, machine, or group requiring encryption, the gateway process rejects the message. If a service advertised by the domain gateway requires encryption, the gateway process rejects the message. (See Setting Encryption Policy for clarification.) If the domain gateway does not require encryption, the gateway process accepts the message and forwards it over the network.
Outbound message	Has digital signature but is not encrypted	The domain gateway process verifies the digital signature and forwards the message with digital signature attached over the network. If the message is destined for a process running Oracle Tuxedo pre-release 7.1 software and assuming interoperability with Oracle Tuxedo pre-release 7.1 software is allowed, the domain gateway process verifies and then removes the digital signature before forwarding the message over the network. (See Interoperating with Pre-Release 7.1 Software and Interoperability for Public Key Security for clarification.)
Outbound message	Outbound message	If the domain gateway process is running within a domain, machine, or group requiring digital signatures, the gateway process rejects the message. If a service advertised by the domain gateway requires digital signatures, the gateway process rejects the message. (See Setting Digital Signature Policy for clarification.) If the domain gateway does not require digital signatures, the gateway process accepts the message and forwards it over the network.

1.16.3.7 Compatibility/Interaction with Other Vendors' Gateways

A domain gateway (`GWTDOMAIN`) process connecting a release 7.1 or later ATMI application to another vendor's gateway process operates on *outbound* message buffers as follows:

1. Decrypts encrypted messages.
2. Verifies digital signatures (if any) and then removes digital signatures.
3. Transmits messages in plaintext format over the network to the vendor's gateway process.

In addition, the domain gateway process enforces the administrative system policies regarding encryption and digital signatures for the ATMI application. As an example, if encryption and/or digital signatures are required at the domain level for the ATMI application, the local domain gateway process rejects any message coming from the other vendor's gateway process.

See Also:

- Security Interoperability
- Mandating Interoperability Policy
- Setting Digital Signature Policy
- Setting Encryption Policy

1.17 Denial-of-Service (DoS) Defense

With more distributed multi-domain Tuxedo applications extending their reach to public networks and less secure environments, the Tuxedo domain gateway is required to better defend against potential threats. These environments may contain insecure networks and untrusted participants, who can initiate or propagate malicious attacks such as Denial-of-Service (DoS) attacks.

The Tuxedo TDomain gateway (GWTDOMAIN) uses the following features to defend against DoS attacks.

Limited/Restricted Connection Numbers

Message Sanity Check

Message Authentication Code (MAC) Usage

- Limited/Restricted Connection Numbers
- Setting Up Connection Limitations/Restrictions
- Message Sanity Check
- Message Authentication Code (MAC) Usage
- Setting up Message Authentication Code (MAC) Usage

1.17.1 Limited/Restricted Connection Numbers

GWTDOMAIN is a daemon server that waits on a well-known TCP port to accept incoming connection requests. This opens the vulnerability to connection flood attack, a type of DoS attack where the attacker continuously tries to establish many connections with GWTDOMAIN at the same time using particular tools (for example, a port scanning program). This causes the domain gateway to waste computing power (time, memory, and so on) to accept the connection requests and allocate resources for each connection.

By limiting the number of connections, GWTDOMAIN can avoid this problem. For more GWTDOMAIN information, see GWTDOMAIN(5).

1.17.2 Setting Up Connection Limitations/Restrictions

The Limited/Restricted Connection Numbers feature requires modification of the `*SERVERS` section in the UBBCONFIG file.

- UBBCONFIG File
- Messages

1.17.2.1 UBBCONFIG File

The CLOPT used to specify the parameter for GWTDOMAIN is "`-x`" using the following syntax: `-x` limit `[:{[` duration `][:` period `]}]`. A colon (:) is used to separate each option.

> **Note:**
>
> The colon (:) can only be used between two options. For example, configurations like ":duration" or "limit::" are invalid.

The default value(s) for the `duration` and `period` options are used if they are not specified.

Please be aware that the timing is not exact for performance reason. There may be a one-second difference.

If the number of current active connections plus the number of closed connections in a specified previous period is greater than the limit, GWTDOMAIN is suspended for a duration specified in seconds.

> **Note:**
>
> The number of current active connections includes both active incoming connections and active outgoing connections. The number of closed connections in a previous *period* includes *both* closed incoming connections and closed outgoing connections. However, when GWTDOMAIN is suspended, none of the closed connections are counted.

`limit`, `duration`, and `period` are defined as follows:

limit
The maximum number of connections. The minimum `limit` value is 0, and the maximum value is 2,147,483,647.

When the limit is reached (or exceeded) and there is an incoming request, GWTDOMAIN is suspended for the given duration. At the same time, the current incoming request which triggers the suspending is not accepted. Polling is resumed after duration has elapsed.

Setting the `limit` to 0 prohibits the domain gateway from accepting any incoming connection requests. In other words, this is an "OUTGOING_ONLY" connection policy.

duration
The duration in seconds to suspend polling for incoming connection when `limit` is reached. The default value is (SCANUNIT * SANITYSCAN) seconds. The minimum `duration` value is 5, and the maximum value is 65,535.

period
The time interval (in seconds) proceeding GWTDOMAIN check point to count the closed connections in the past. When not specified, the default value is the same as `duration`. The minimum `period` value is 0, and the maximum value is 65,535.

If `period` is specified as 0, the number of closed connections in a prior period will *always* be 0, `limit` only counts active connections.

- Examples

1.17.2.1.1 Examples

The following example depicts an example where the GWTDOMAIN `limit` is set to 512 concurrent socket connections. When the 512 `limit` is reached and there is an incoming request, GWTDOMAIN will stop polling and accepting new incoming connection requests for a duration of 300 seconds (or, 5 minutes). Since `period` is specified as 0, only the active connections are counted.

Listing 1-1 UBBCONFIG File Example 1

```
# UBBCONFIG
 ...
 *SERVERS
 GWTDOMAIN SRVGRP=GWGRP1 SRVID=2 CLOPT= "-A -- -x
 512:300:0"
```

The following example depicts an example where the GWTDOMAIN limit is set to 200 concurrent socket connections. When the 200 `limit` is reached, (for example:

- there are100 outgoing connections
- 50 incoming connections,
- in the passed 60 seconds 50 connections were closed (including outgoing connections and incoming connection
- a current incoming connection is requested

and since the `duration` value is not specified, GWTDOMAIN will stop polling and accepting new incoming connection requests for the `duration` default value SCANUNIT * SANITYSCAN seconds.

> **Note:**
>
> The current incoming connection that triggered the suspension is also not accepted, and is closed at the end of the suspended duration.

Listing 1-2 UBBCONFIG File Example 2

```
# UBBCONFIG
 ...
 *SERVERS
 GWTDOMAIN SRVGRP=GWGRP1 SRVID=2 CLOPT= "-A -- -x
 200::60"
```

1.17.2.2 Messages

The following conditions will post messages to USERLOG:

- A new connection request arrives that reaches the preset number of connections limit:

```
<LIBGW_CAT 5359> "WARN: The number of connections for
               <ldom-name> exceeds limit <%d>, start to suspend for
               <%d> seconds"
```

- GWTDOMAIN resumes checking for new incoming connection request:

```
<LIBGW_CAT 5360> "INFO: Resume accepting connection
               request"
```

> **Note:**
>
> These two messages can be controlled using the "throttle message" mechanism to avoid the potential of flooding the USERLOG.

- If limit is specified as 0, when GWTDOMAIN starts up:

```
<LIBGW_CAT 5361> "INFO: The connection limit for
               <ldom-name> is set to 0. No incoming connection request will
               be accepted."
```

1.17.3 Message Sanity Check

The sanity check of message is strengthened with this feature, to protect GWTDOMAIN from crash when under attack. This feature is deployed automatically after installed, no configuration work needed.

1.17.4 Message Authentication Code (MAC) Usage

By associating the message authentication code (MAC) with messages, a Tuxedo domain gateway can validate and authenticate them. With MAC, the domain gateway can defend against various types of DoS attacks (for example, message tampering, message forging, and message replay attack).

This feature can only take effect when LLE and/or domain SECURITY is configured. MAC works after connection is established. When a MAC message from a remote domain gateway fails validation and authentication, the corresponding connection is dropped. All pending messages are also dropped, and all on-going service requests fail.

GWTDOMAIN determines whether MAC is turned on for the session during the session negotiation phase. MAC can only be enabled when either LLE and/or SECURITY is supported and activated for the session.

> **Note:**
>
> SSL does not support MAC usage.

It is not necessary to turn on the SECURITY feature to enable MAC; however, *it is recommended* since SECURITY can be used to defend against the “man-in-the-middle” attack.

- Performance Impact

1.17.4.1 Performance Impact

When MAC is turned on, it may cause degradation on the throughput and response time for requests across domains.

1.17.5 Setting up Message Authentication Code (MAC) Usage

There are two options that you configure the MAC feature. You can use DMCONFIG file configuration, or MIB configuration.

- DMCONFIG File Configuration
- MIB Configuration

1.17.5.1 DMCONFIG File Configuration

This feature can be configured in DM_TDOMAIN section of DMCONFIG file with two new keywords, MAC and MACLEVEL. MAC is used to toggle the MAC feature for a session; MACLEVEL is used to specify the MAC level.

> **Note:**
>
> A large number MACLEVEL means the stronger algorithm from cryptographic point of view, but will introduce more performance degradation.

Table 1-15 DMCONFIG File Keywords

Keyword	Option	Definition
MAC	OFF	Turn off feature. This is the default value.
	ON	Turn on feature. The established session MAC support depends on the negotiation result between the two domain gateways.
	MANDATORY	Turn on feature. The session cannot be setup if: • the remote domain does not support or disable the MAC feature, or • neither LLE nor domain SECURITY is available.
MACLEVEL	0	Only protects the message header with MAC. This is the default value
	1	Protects the entire message with MAC using MD5-based algorithm
	2	Protects the entire message with MAC using SHA1-based algorithm.
	3	Protects the entire message with MAC, using SHA256-based algorithm.

The following example depicts an example DMCONFIG configuration.

Listing 1-3 DMCONFIG File Configuration

```
# DMCONFIG
...
*DM_TDOMAIN
"RDOM" NWADDR="//RHOST:RPORT"
       MAC="ON"
       MACLEVEL=1
```

1.17.5.2 MIB Configuration

Dynamic setting of MAC via MIB does not have any impact on existing domain sessions. It only takes effect for new connections.

Two new attributes are added to support MIB interface in the `T_DM_TDOMAIN` class definition attribute table: `TA_DMMAC` and `TA_DMMACLEVEL`.

Table 1-16 DM_MIB(5): T_DM_TDOMAIN Class Definition Attribute Table

Attribute	Type	Permissions	Values	Default
TA_DMMAC	string	rw-------	string "{OFF|ON| MANDATORY}"	"OFF"
TA_DMMACLEVEL	string	rw-------	string "{0|1|2|3}"	"0"

```
TA_DMMAC="{OFF|ON|MANDATORY}"
```

Relevant to remote domain access points only. Specifies whether to activate MAC feature when connecting to the remote domain. Supported values are "OFF", "ON", "MANDATORY".

"OFF"
Specifies the connection to a domain gateway does not use the MAC feature.

"ON"
Specifies the connection to a domain gateway that uses the MAC feature.

"MANDATORY"
Specifies the connection to a domain gateway must use the MAC feature, otherwise a successful connection cannot be established.

```
TA_DMMACLEVEL="{0|1|2|3}"
```

Relevant to remote domain access points only. Specifies the manner when protecting the whole message with MAC. "0" specifies that only the message header is protected by MAC. "1", "2", and "3" specify that the entire message is protected by MAC via an algorithm based on MD5, SHA1 and SHA256.

The following listings depicts examples of how to *retrieve* and *update* MAC attributes using ud32 respectively.

Listing Sample Retrieve MAC Attribute Script

```
SRVCNM .TMIB
         TA_OPERATION GET
```

```
        TA_CLASS T_DM_TDOMAIN
        TA_DMACCESSPOINT RDOM
        TA_DMNWADDR //host:port
```

Listing Sample Update MAC Attribute Script

```
SRVCNM .TMIB
        TA_OPERATION SET
        TA_CLASS T_DM_TDOMAIN
        TA_DMACCESSPOINT RDOM
        TA_DMNWADDR //host:port
        TA_DMLACCESSPOINT LDOM
        TA_DMMAC MANDATORY
        TA_DMMACLEVEL 2
```

- MAC Negotiation
- Messages
- ERROR Messages

1.17.5.2.1 MAC Negotiation

Suppose there are two domains: DOM1 and DOM2. When DOM1 (initiator) establishes a session with DOM2 (acceptor), the MAC negotiation result is (1) MAC = ON; and (2) MACLEVEL = 2.

The first column from each table contains the configuration parameter for DOM2 in the DM_TDOMAIN section of the DOM1 DMCONFIG file. The header row holds the configuration parameter for DOM1 in the DM_TDOMAIN section of the DOM2 DMCONFIG file

An "ERROR" result in Table 4 means that the connection cannot be established. When MAC negotiation result is ON, the MACLEVEL for the entire message is determined as shown in Table 5.

When MAC is turned on, the MACLEVEL in use is set to the higher number, or max (m1,m2) for safety purpose. It must be supported by both endpoints (that is, not greater than min (Max1,Max2)). In short, the negotiated MACLEVEL must satisfy following relationship: *max(m 1 , m 2)<=negotiated MACLEVEL<=min(Max 1 , Max 2*), otherwise the connection is closed with one ERROR message logged in USERLOG.

1.17.5.2.2 Messages

The following messages are posted to the USERLOG:

INFO Messages

The following INFO messages are printed after agreement about MAC is made to denote MAC feature for one session

- MAC is not supported for the session:

  ```
  <LIBGWT 1686> "INFO: MAC is not supported for session(<ldom-name>, <rdom-
  name>"
  ```

> **Note:**
>
> This message is printed only in the domain with MAC set to "ON".

- MAC is turned on for the session:

```
<LIBGWT 1687> "INFO: MAC is turned on for session(<ldom-name>, <rdom-
name>) and effective MACLEVEL is <%d>"
```

1.17.5.2.3 ERROR Messages

The following error messages appear during session negotiation and MAC validation phase. The connection is dropped when these messages are printed:

- MAC is mandatory, but MAC is not supported for the session when negotiation:

```
<LIBGWT 1681> "ERROR: MAC is MANDATORY but remote domain <rdom-name> does
not support this feature"
```

- MAC is mandatory but neither LLE nor SECURITY is supported when negotiation:

```
<LIBGWT 1682> "ERROR: MAC is MANDATORY but neither LLE nor SECURITY is
supported for connection of (<ldom-name>,<rdom-name>)"
```

- MAC is mandatory in the remote domain but MAC is not supported in local domain:

```
<LIBGWT 1683> "ERROR: MAC is MANDATORY in remote domain <rdom-name> but
not supported in local domain <ldom-name>"
```

- MAC negotiation fails to make an agreement on MACLEVEL:

```
<LIBGWT 1684> "ERROR: MAC failed to make an agreement on MACLEVEL
(<ldom_name> is <%d>..<%d>,<rdom-name> is <%d>..<%d>)"
```

> **Note:**
>
> The four corresponding parameters for "%d" placeholder in this message are m1, Max1, m2, and Max2.

- MAC fails validation and authentication:

```
<LIBGWT 1685> "ERROR: Message from <rdom-name> has invalid MAC"
```

1.18 Password Pair Protection

Password pair protection is deployed automatically after installation; configuration is not required. It improves the GWTDOMAIN security mechanism and removes the previous security restriction that did not allow dual password pairs with the same remote password.

Password pair protection is functional only when supported by both local and remote domains. If it is not supported by both local and remote domains, it does not affect existing behavior.

2

Administering Security

The following sections explain how to set security policies for an Oracle Tuxedo ATMI application:

2.1 What Administering Security Means

Administering security for an ATMI application involves setting and enforcing security policies for the components of the application, including its clients, server machines, and gateway links.

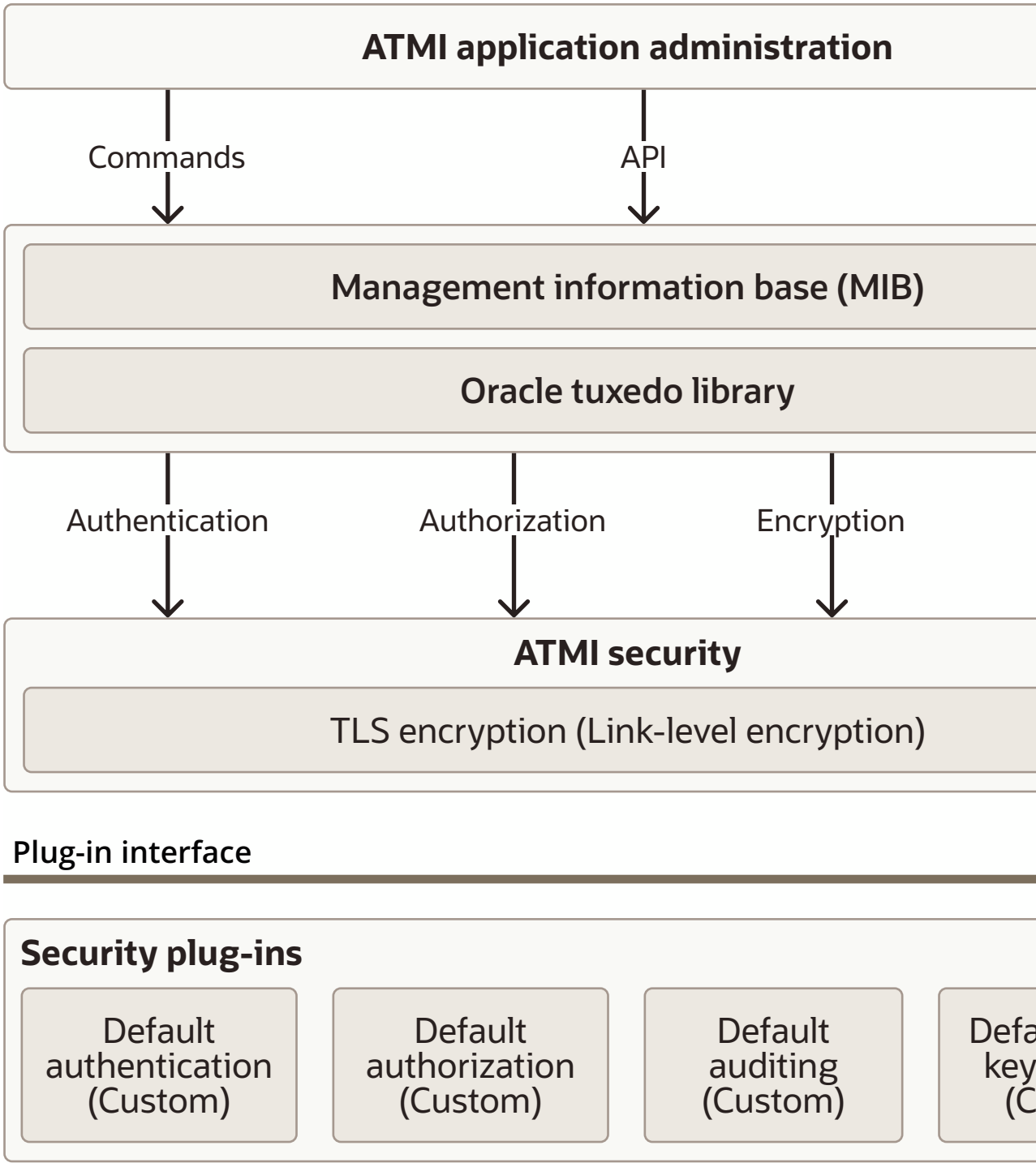
The application administrator sets the security policies for the ATMI application, and the Oracle Tuxedo system upon which the ATMI application is built enforces those policies.

The Oracle Tuxedo system offers the following ATMI security capabilities:

- Authentication
- Authorization
- Auditing
- Link-level encryption
- TLS Encryption
- Public key security

All but one of the security capabilities can be configured by the application administrator. The exception is auditing, which cannot be configured, as shown in the following figure.

Figure 2-1 Administering ATMI Security



See Also:

- Security Administration Tasks
- What Security Means
- What Programming Security Means

2.2 Security Administration Tasks

Security administration consists of the following tasks:

- Setting the Oracle Tuxedo Registry
- Configuring an ATMI Application for Security
- Setting Up the Administration Environment
- Administering Operating System (OS) Security
- Administering Authentication
- Administering Authorization
- Administering Link-Level Encryption
- Administering SSL Encryption
- Administering Public Key Security
- Administering Default Authentication and Authorization

2.3 Setting the Oracle Tuxedo Registry

The application administrator needs to know about the Oracle Tuxedo registry if the ATMI application is to be configured with one or more custom security capabilities. On the other hand, if the ATMI application is to be configured only with default security, the Oracle Tuxedo registry does not need to be changed.

The Oracle Tuxedo registry is a disk-based repository for storing information related to plug-in modules. Initially, this registry holds registration information about the default security plug-ins.

- Purpose of the Oracle Tuxedo Registry
- Registering Plug-ins

2.3.1 Purpose of the Oracle Tuxedo Registry

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

The first step in loading a plug-in is to register the plug-in with the host operating system. Registering a plug-in adds an entry for the plug-in to the Oracle Tuxedo registry, which is a set of binary files that stores information about active plug-ins. There is one registry per Oracle Tuxedo installation.

- On a UNIX host machine, the Oracle Tuxedo registry is in the `$TUXDIR/udataobj` directory.
- On a Windows 2003 host machine, the Oracle Tuxedo registry is in the `%TUXDIR%\udataobj` directory.

Every Workstation client and server machine in an ATMI application must use the same set of plug-in modules.

2.3.2 Registering Plug-ins

The administrator of an ATMI application in which custom plug-ins will be used is responsible for registering those plug-ins and performing other registry related tasks. An administer can register plug-ins in the Oracle Tuxedo registry *only* from the local machine. That is, an administrator cannot register plug-ins while logged on to the host machine from a remote location.

Three commands are available for administering plug-ins:

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

Instructions for using these commands are available in *Developing Security Services for ATMI and CORBA Environments.* (This document contains the specifications for the security plug-in interface, and describes the *plug-in framework* feature that makes the dynamic loading and linking of security plug-in modules possible.) Also, when installing custom plug-ins, the supplying third-party security vendor should provide instructions for using these commands to set up the Oracle Tuxedo registry to access the custom plug-ins.

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

> **See Also:**
>
> - Configuring an ATMI Application for Security

2.4 Configuring an ATMI Application for Security

An application administrator configures security for the ATMI application on the `MASTER` machine when the application is inactive. The underlying Oracle Tuxedo system propagates the configuration information to the other machines in the ATMI application when the application is booted.

As the administrator, you can configure security for your ATMI application by:

- Editing the configuration file (`UBBCONFIG`)
- Changing the`TM_MIB`

The set of security parameters involved depends upon the security capability (authentication, authorization, link-level encryption, or public key) and whether you are using the default or custom security software.

- Editing the Configuration File

- Changing the TM_MIB

2.4.1 Editing the Configuration File

You can edit the `UBBCONFIG` configuration file to set security policies for an ATMI application. The `UBBCONFIG` configuration file may have any filename, as long as the content of the file conforms to the format described on the UBBCONFIG(5) reference page in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

For more details about `UBBCONFIG` and its binary equivalent, `TUXCONFIG`, see About the Configuration File and Creating the Configuration File in *Setting Up an Oracle Tuxedo Application*.

2.4.2 Changing the TM_MIB

The `TM_MIB` defines a set of classes through which the fundamental aspects of an ATMI application may be configured and managed. Separate classes are designated for machines, servers, networks, and so on. You should use the reference page TM_MIB(5) in combination with the generic Management Information Base (MIB) reference page MIB(5) to format administrative requests and interpret administrative replies. The MIB reference pages are defined in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

Other component MIBs, including the `ACL_MIB`, `DM_MIB`, and `WS_MIB`, also play a role in managing security for an ATMI application. The reference page ACL_MIB(5) defines the `ACL_MIB`, the reference page DM_MIB(5) defines the `DM_MIB`, and the reference page WS_MIB(5) defines the `WS_MIB`.

For more information about Oracle Tuxedo MIBs, start with MIB(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*. Also, see *Introducing Oracle Tuxedo ATMI*.

> **See Also:**
>
> - Setting Up the Administration Environment

2.5 Setting Up the Administration Environment

The application administrator defines certain environment variables for an ATMI application as part of configuring the application. The values defined for the variables are absolute pathnames that reference Oracle Tuxedo executables and data libraries.

Being able to find such files is essential to the job of administering an ATMI application. For example, all commands needed to manage application security are located in `$TUXDIR/bin` on a UNIX host machine, and in `%TUXDIR%\bin` on a Windows 2003 host machine.

For details on setting up the administration environment, see *Administering an Oracle Tuxedo Application at Run Time*.

Note:

- Administering Operating System (OS) Security
- Administering Authentication
- Administering Authorization
- Administering Link-Level Encryption
- Administering SSL Encryption
- Administering Public Key Security
- Administering Default Authentication and Authorization

2.5.1 Administering Operating System (OS) Security

In addition to the security features in the ATMI environment of the Oracle Tuxedo product, the application administrator needs to take full advantage of the security features of the host operating system to control access to files, directories, and system resources.

Most ATMI applications are managed by an application administrator who configures and boots the application, monitors the running application, and makes changes to it dynamically, as necessary. Because the ATMI application is started and run by the administrator, server programs are run with the administrator's permissions and are therefore considered secure or "trusted." This working method is supported by the login mechanism and the read and write permissions on the files, directories, and system resources provided by the underlying operating system.

Clients, on the other hand, are not started by the administrator. Instead, they are run directly by users with their own permissions. As a result, clients are not trusted.

In addition, users running native clients (that is, clients running on the same machine on which the server is running) have access to the configuration file and interprocess communication (IPC) mechanisms such as the *bulletin board* (in shared memory). Users running native clients always have such access, even when additional ATMI security is configured.

2.5.1.1 Recommended Practices for OS Security

As the administrator, you can improve operating system security by observing the following general rules:

- Limit access to files and IPC resources to the application administrator.
- Have "trusted" client programs run only with the permissions of the administrator (using a `setuid` utility).
- For maximum security on your operating system, allow only Workstation clients to access the application; client programs should not be allowed to run on the same machines on which application servers and administrative programs run.
- Combine all of these practices with ATMI security so that the application can identify any client making a request.

> **See Also:**
>
> - Operating System (OS) Security
> - Security Administration Tasks

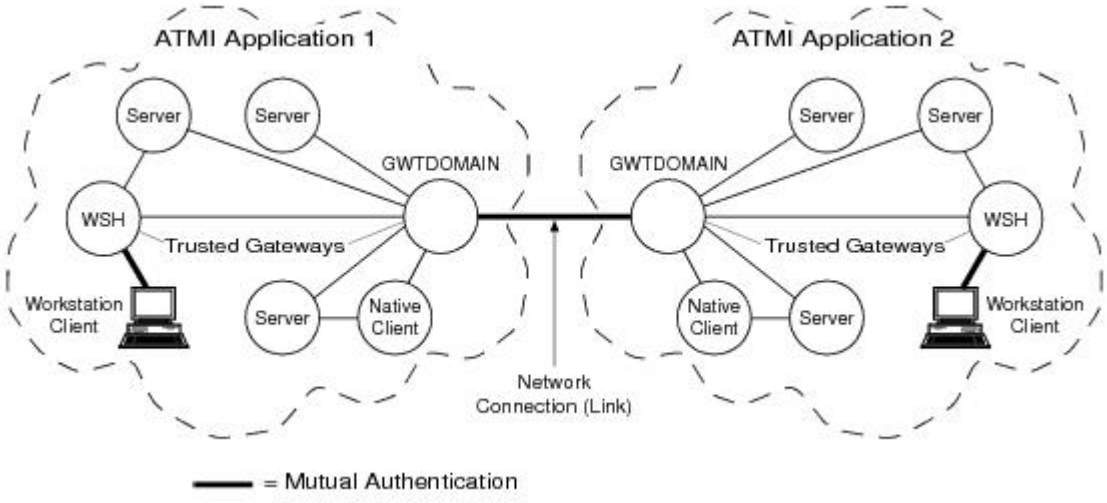
2.6 Administering Authentication

Authentication allows communicating processes to prove their identities. It is the foundation for most other security capabilities.

Except for the configuration instructions identified in this topic, the procedures for administering authentication depend upon the underlying authentication system of the application. For procedures to administer a custom authentication system, see the documentation for that system. For procedures to administer the default authentication system, see Administering Default Authentication and Authorization.

The following figure demonstrates the use of the *delegated trust authentication model* by applications running Oracle Tuxedo release 7.1 or later software. Workstation handlers (WSHs) and domain gateways (`GWTDOMAINs`) are known as *trusted system gateway processes* in the delegated trust authentication model, which is described in Understanding Delegated Trust Authentication.

Figure 2-2 Mutual Authentication in the Delegated Trust Authentication Model



> **Note:**
>
> Mutual authentication is not used for a native client, which authenticates with itself.

The following topics provide the instructions needed to set up the configuration shown in the preceding figure. All of the topics involve authentication and the authentication plug-in.

- Specifying principal names
- Mandating interoperability policy
- Establishing a link between domains
- Setting ACL policy
- Setting credential policy

See Also:

- Authentication
- Default Authentication and Authorization
- Administering Default Authentication and Authorization
- Security Administration Tasks
- Security Interoperability
- Security Compatibility
- Oracle Tuxedo Domains (Multiple-Domain) Servers in *Introducing Oracle Tuxedo ATMI*

2.7 Specifying Principal Names

As the administrator, you use the following configuration parameters to specify principal names for the workstation handler (WSH), domain gateway (GWTDOMAIN), and server processes running in your ATMI application built with release 7.1 or later of the Oracle Tuxedo software.

Parameter Name	Description	Setting
SEC_PRINCIPAL_NAME in UBBCONFIG (TA_SEC_PRINCIPAL_NAME in TM_MIB)	During application booting, each WSH, domain gateway, and server process in the ATMI application calls the authentication plug-in to acquire security credentials for the security principal name specified inSEC_PRINCIPAL_NAME.*	1 - 511 characters. If not specified at any level in the configuration hierarchy, the security principal name defaults to theDOMAINID string specified in the UBBCONFIG file.
CONNECTION_PRINCIPAL_NAME for local domain access point in DMCONFIG (TA_DMCONNPRINCIPALNAME for LACCESSPOINT in DM_MIB)	During application booting, each domain gateway process in the ATMI application calls the authentication plug-in a second time to acquire security credentials for the connection principal name specified in CONNECTION_PRINCIPAL_NAME.*	1 - 511 characters. If not specified, the connection principal name defaults to theACCESSPOINTID** string for the local domain access point specified in the DMCONFIG file.
*The topics that follow explain how the system processes acquire credentials and why they require them. **. The ACCESSPOINTID parameter is also known as DOMAINID.		

SEC_PRINCIPAL_NAME is specified any of the following four levels in the configuration hierarchy:

- RESOURCES section in UBBCONFIG or T_DOMAIN class in TM_MIB

- MACHINES section in UBBCONFIG or T_MACHINE class in TM_MIB
- GROUPS section in UBBCONFIG or T_GROUP class in TM_MIB
- SERVERS section in UBBCONFIG or T_SERVER class in TM_MIB

A security principal name at a particular configuration level can be overridden at a lower level. For example, suppose you configure terri as the principal name for machine mach1, and john as the principal name for server serv1 running on mach1. The processes on mach1 behave as follows:

- All WSH, domain gateway, and server processes on mach1 except serv1 processes use terri as a principal name.
- All serv1 processes use john as a principal name.

> **Note:**
>
> Security principal information must be specified for all machines in a networked application (MP mode) configuration. If a boot failure occurs, examine the ULOG files on both sides of the connection where the failure occurred for more information about the cause of the failure.

- How System Processes Acquire Credentials
- Why System Processes Need Credentials
- Example UBBCONFIG Entries for Principal Names

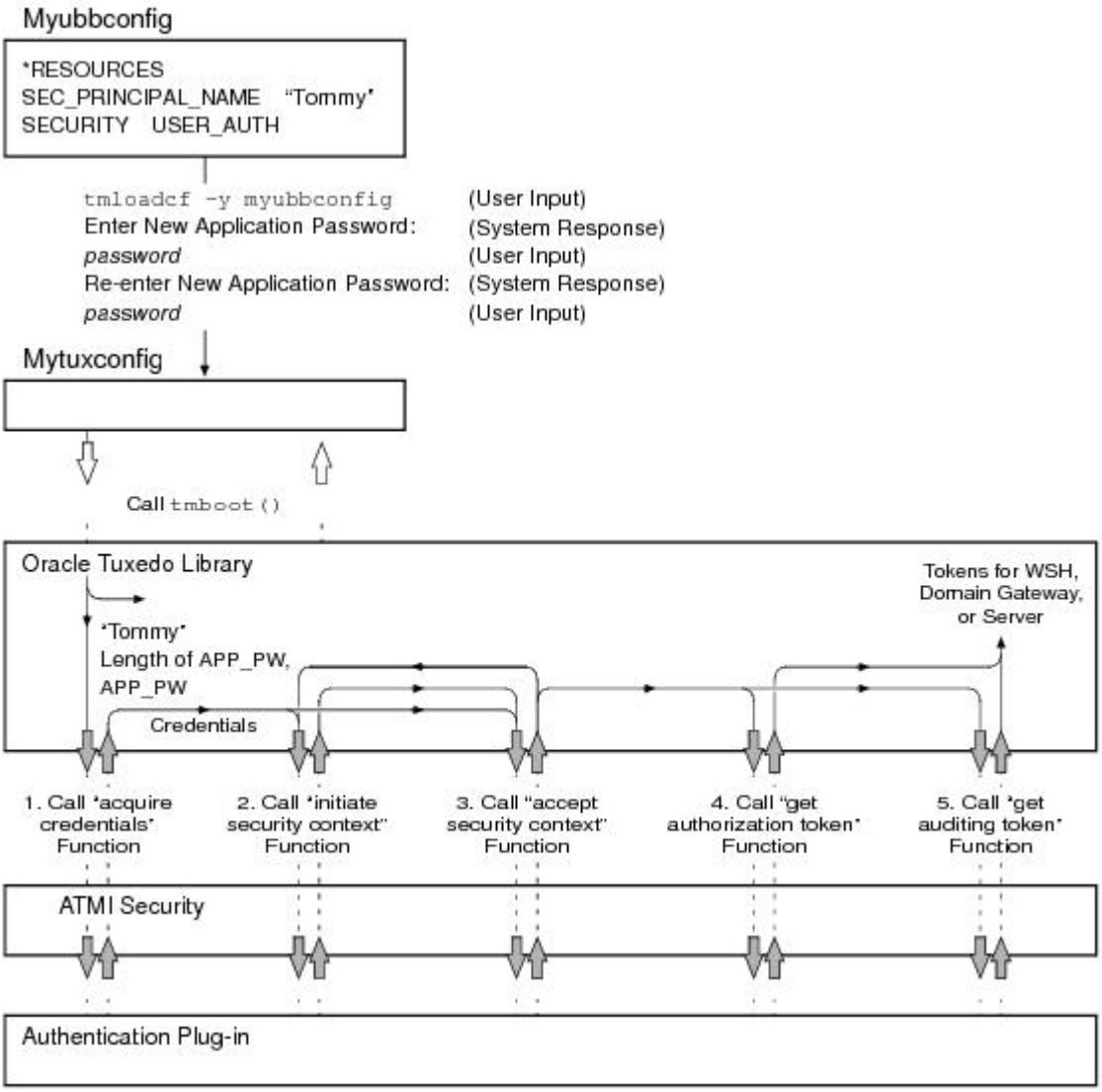
2.7.1 How System Processes Acquire Credentials

During application booting, each WSH, domain gateway, and server process in the ATMI application includes its *security principal name* as an argument when calling the authentication plug-in:

1. to acquire security credentials
2. get authorization and auditing tokens for itself

The following figure illustrates the procedure.

Figure 2-3 Acquiring Credentials and Tokens During Application Booting



Each domain gateway process in the application calls the authentication plug-in a second time to acquire credentials and tokens for its assigned *connection principal name*.

2.7.2 Why System Processes Need Credentials

A WSH needs credentials so that it can authenticate Workstation clients that want to join the application, and to get authorization and auditing tokens for the authenticated Workstation clients. A WSH needs its own authorization and auditing tokens when handling requests from pre-release 7.1 clients (clients running Oracle Tuxedo release 6.5 or earlier software) so that it can call the authentication plug-in to establish identities for the older clients. This behavior is described in Mandating Interoperability Policy .

A domain gateway needs one set of credentials so that it can authenticate remote domain gateways for the purpose of establishing links between ATMI applications, as described in Establishing a Link Between Domains. (No authorization or auditing tokens are assigned to authenticated remote domain gateways.) A domain gateway acquires these credentials for the principal name specified in the `CONNECTION_PRINCIPAL_NAME` parameter.

A domain gateway needs a second set of credentials so that it can handle requests from pre-release 7.1 clients, which involves calling the authentication plug-in to establish identities for the older clients. This behavior is described in Mandating Interoperability Policy . It also needs these credentials to establish identities when enforcing the local access control list (ACL) policy, as described in Setting ACL Policy. A domain gateway acquires these credentials for the principal name specified in the `SEC_PRINCIPAL_NAME` parameter.

A system or application server needs its own authorization and auditing tokens when handling requests from pre-release 7.1 clients so that it can call the authentication plug-in to establish identities for the older clients. This behavior is described in Mandating Interoperability Policy .

A server also needs its own tokens when performing a *server permission upgrade*, which occurs when the authorization and auditing tokens of the server are assigned to messages that pass through the server but originate at a client. The service upgrade capability is described in Replacing Client Tokens with Server Tokens.

> **Note:**
>
> An application server cannot call the authentication plug-in itself. It is the underlying system code that calls the authentication plug-in for the application server.

2.7.3 Example UBBCONFIG Entries for Principal Names

The following example pertains to specifying security principal names in the `UBBCONFIG` file using the `SEC_PRINCIPAL_NAME` parameter. For an example of specifying connection principal names in the `DMCONFIG` file using the `CONNECTION_PRINCIPAL_NAME` parameter, see Example DMCONFIG Entries for Establishing a Link.

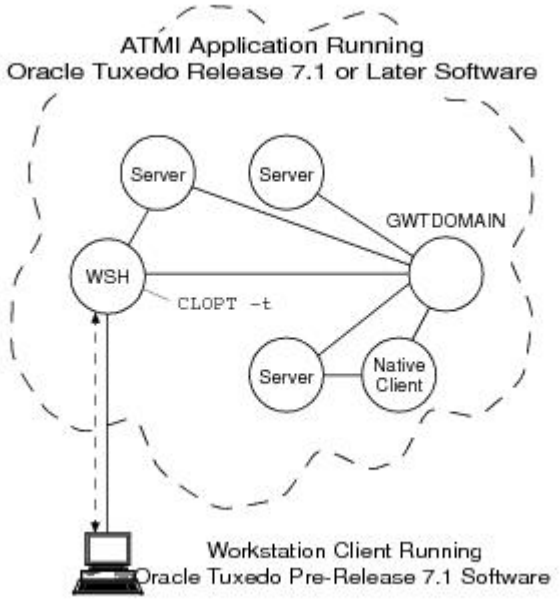
```
*RESOURCES
         SEC_PRINCIPAL_NAME    "Tommy"
               .
               .
               .
*SERVERS
"TMQUEUE"     SRVGRP="QUEGROUP"     SRVID=1
CLOPT="-t -s secsdb:TMQUEUE"
SEC_PRINCIPAL_NAME="TOUPPER"
```

> **Note:**
>
> - Mandating Interoperability Policy
> - Establishing a Link Between Domains
> - Setting ACL Policy
> - Security Administration Tasks

2.8 Mandating Interoperability Policy

As the administrator, you use the `CLOPT -t` option in the `UBBCONFIG` file to allow WSH, domain gateway (`GWTDOMAIN`), and server processes in your ATMI application to interoperate with machines running Oracle Tuxedo pre-release 7.1 (6.5 or earlier) software. In addition, you use the `WSINTOPPRE71` environment variable to allow Workstation clients to interoperate with machines running Oracle Tuxedo pre-release 7.1 software. The following four figures show what interoperability means for these processes.

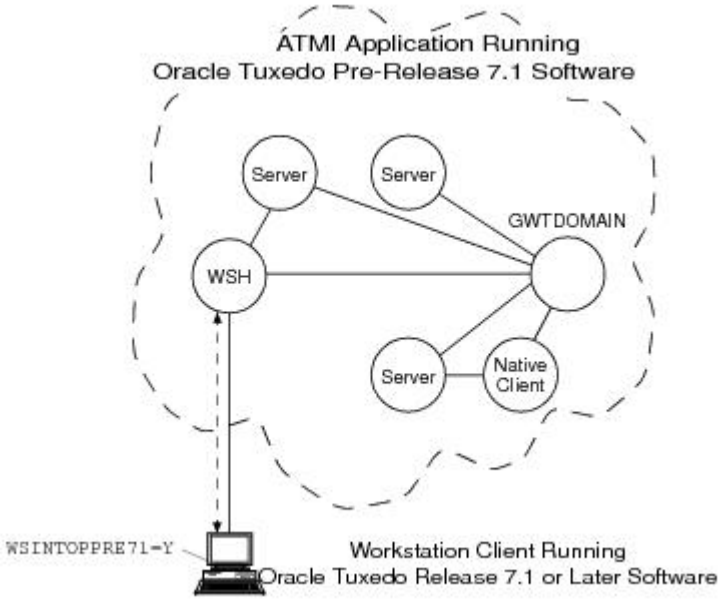
Figure 2-4 WSH Operating with Older Workstation Client



In the preceding figure, the WSH authenticates with the Workstation client using an older (pre-release 7.1) authentication protocol, calls the internal impersonate user function to get authorization and auditing tokens for the client, and attaches the tokens to the client request. If the `CLOPT -t` option is not specified for the workstation listener (WSL) that controls the WSH, no communication is possible between the newer WSH and the older Workstation client.

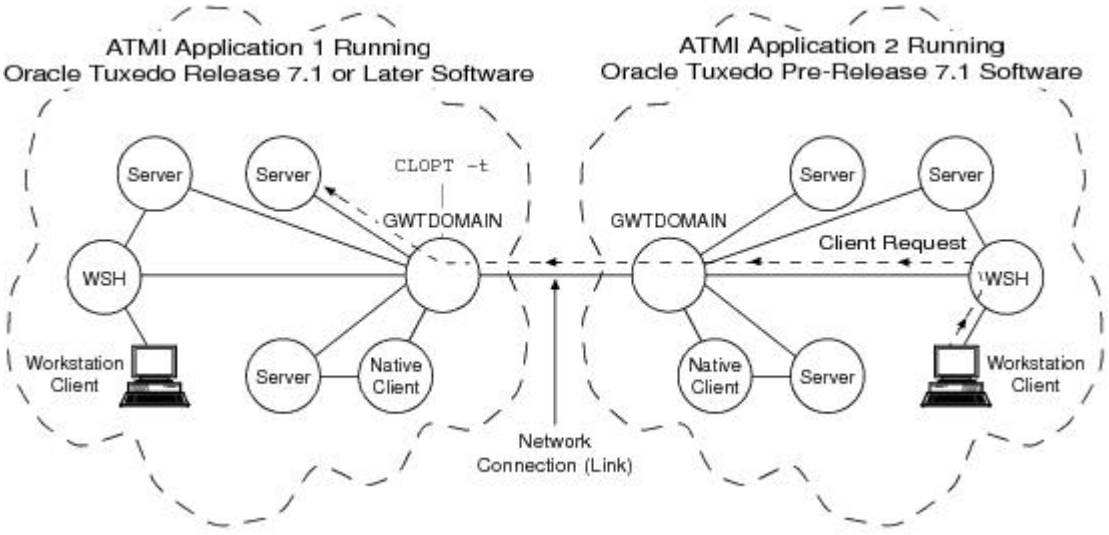
> **Note:**
>
> The impersonate user function involves calling the authentication plug-in to establish an identity for the older client. See Establishing an Identity for an Older Client for details.

Figure 2-5 Older WSH Operating with Workstation Client



In the preceding figure, the WSH authenticates with the Workstation client using an older (pre-release 7.1) authentication protocol; the client request does *not* receive authorization and auditing tokens. If the `WSINTOPPRE71` environment variable is not set at the Workstation client or is set to `N`, no communication is possible between the older WSH and the newer Workstation client.

Figure 2-6 Server Interoperating with Older ATMI Application

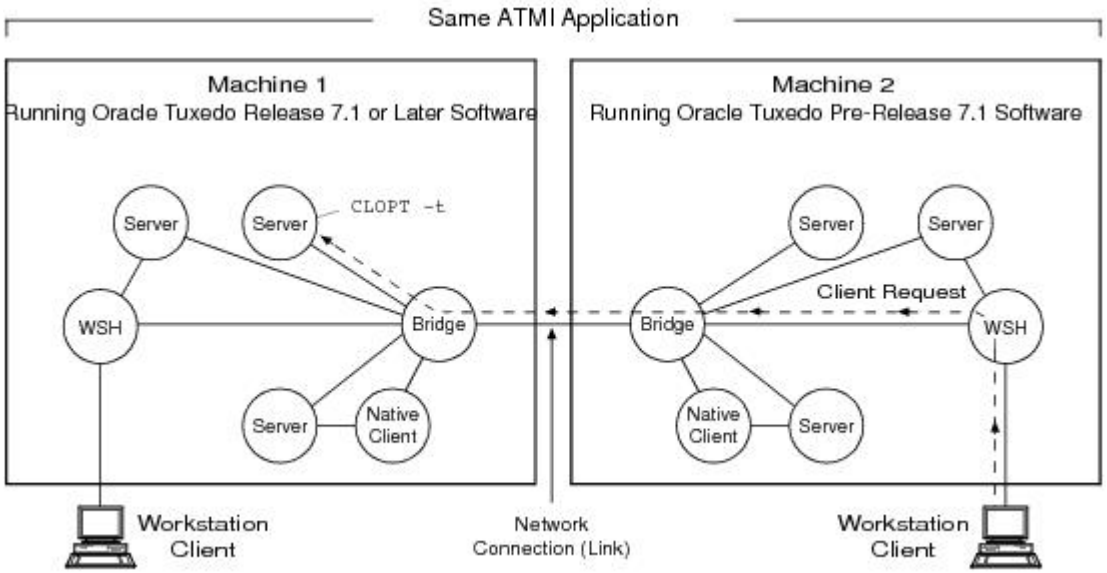


In the preceding figure, the local domain gateway (`GWTDOMAIN`) in application 1 authenticates with the remote domain gateway in application 2 using an older (pre-release 7.1)

authentication protocol. Upon receiving a request from a remote client, the local domain gateway calls the internal impersonate user function to get authorization and auditing tokens for the remote client and then attaches the tokens to the client request. For any outbound client request (client request originating in application 1 and destined for application 2), the local domain gateway strips the tokens from the request before sending the request along with the client's *application key* to the older application. (See Application Key for a description of the application key.)

If the `CLOPT -t` option is not specified for the domain gateway, no communication is possible between the newer ATMI application and the older ATMI application.

Figure 2-7 Server Interoperating with Older Oracle Tuxedo Systems



In the preceding figure, the destination server on machine 1 calls the internal impersonate user function to get authorization and auditing tokens for the remote client on machine 2, attaches the tokens to the client request, and then performs the request *assuming* the client passes any authorization checks. If the `CLOPT -t` option is not specified for the server, no communication is possible between the newer server and the older client.

> **Note:**
>
> Also, in the preceding figure, if the WSH on machine 1 receives a client request destined for a server on machine 2, the WSH strips the tokens from the request before sending the request along with the client's application key to the older system. Similarly, if the native client on machine 1 sends a request to a server on machine 2, the native client strips the tokens from the request before sending the request along with the client's application key to the older system. See Application Key for a description of the application key.

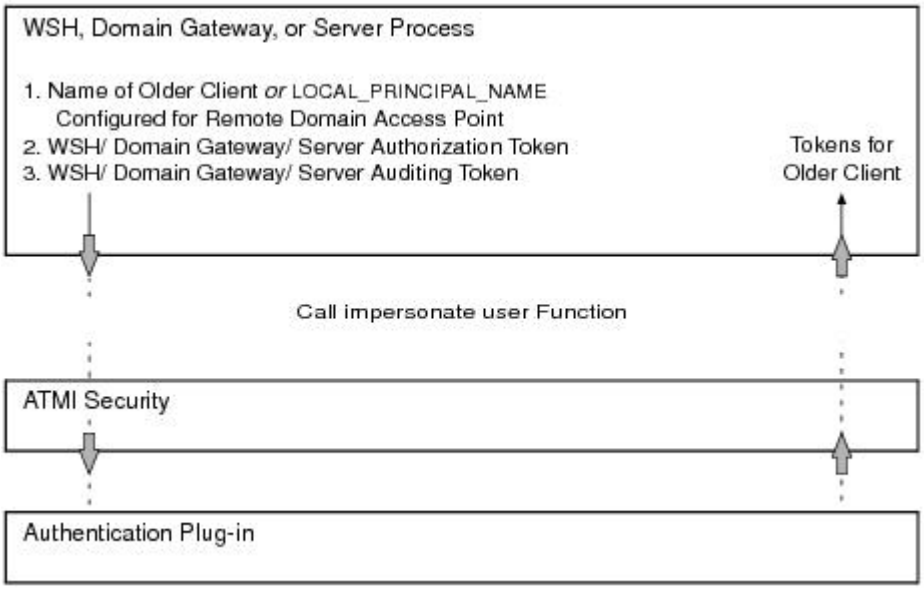
- Establishing an Identity for an Older Client
- Summarizing How the CLOPT -t Option Works

- Example UBBCONFIG Entries for Interoperability

2.8.1 Establishing an Identity for an Older Client

For a WSH, domain gateway (`GWTDOMAIN`), or server process to establish an identity for an older client, the process calls the internal impersonate user function to obtain authorization and auditing tokens for the older client. The following figure demonstrates the procedure.

Figure 2-8 Obtaining Authorization and Auditing Tokens for an Older Client



- How the WSH Establishes an Identity for an Older Client
- How the Domain Gateway Establishes an Identity for an Older Client
- How the Server Establishes an Identity for an Older Client

2.8.1.1 How the WSH Establishes an Identity for an Older Client

When the `CLOPT -t` option is specified, the WSH establishes an identity for an older client using the `usrname` field of the `TPINIT` buffer for C, or the `USRNAME` field of the `TPINFDEF-REC` record for COBOL. (The WSH receives a `TPINIT` buffer/ `TPINFDEF-REC` record from a client when the client attempts to join the application, as described in Joining the ATMI Application.) The WSH includes the user name as the principal name when calling the impersonate user function.

For default authentication plug-ins, the impersonate user function finds the user name and its associated application key (user identifier, group identifier combination) in the local `tpusr` file, and then includes the user name and application key in both the authorization and auditing tokens created for the older client. The `tpusr` file is briefly described in Setting Up the User and Group Files.

2.8.1.2 How the Domain Gateway Establishes an Identity for an Older Client

When the `CLOPT -t` option is specified, the domain gateway establishes an identity for an older client using the `LOCAL_PRINCIPAL_NAME` string configured for the remote domain access point.

(The domain gateway searches the DM_REMOTE section of the local BDMCONFIG file—the binary equivalent of the DMCONFIG(5) file—to find the LOCAL_PRINCIPAL_NAME string for the remote domain access point. If not specified, the identity defaults to the ACCESSPOINTID string for the remote domain access point.) The domain gateway uses the LOCAL_PRINCIPAL_NAME string as the principal name when calling the impersonate user function.

For default authentication plug-ins, the impersonate user function finds the LOCAL_PRINCIPAL_NAME string and its associated application key in the local tpusr file, and then includes that string (identity) and application key in both the authorization and auditing tokens created for the older client.

2.8.1.3 How the Server Establishes an Identity for an Older Client

When the CLOPT -t option is specified, the server establishes an identity for an older client using the client's assigned application key. (The client request received by the server contains the client's assigned application key.) The server finds the application key and its associated name in the local tpusr file, and then includes the name as the principal name when calling the impersonate user function.

For default authentication plug-ins, the impersonate user function finds the name and its associated application key in the local tpusr file, and then includes the name and application key in both the authorization and auditing tokens created for the older client.

2.8.2 Summarizing How the CLOPT -t Option Works

The following table summarizes the functionality of WSH, domain gateway, and server processes when interoperability *is* and is *not* allowed using the CLOPT -t option.

Table 2-1 Functionality of WSH, Domain Gateway, and Server Processes When Interoperability Is and Is Not Allowed

Process	Interoperability Allowed (CLOPT -t)	Interoperability Not Allowed
Workstation Handler (WSH)	If the WSH receives a request from a pre-release 7.1 Workstation client to join the application, the WSH authenticates the client using a pre-release 7.1 authentication protocol and calls the impersonate user function to get authorization and auditing tokens for the client based on the user name given in the request. When the WSH receives a service request from the authenticated Workstation client, it attaches the tokens to the client request and forwards the request to the destination server.	If the WSH receives a request from a pre-release 7.1 Workstation client to join the application, the WSH rejects the request. No communication is possible between the newer WSH and the older Workstation client.

Table 2-1 (Cont.) Functionality of WSH, Domain Gateway, and Server Processes When Interoperability Is and Is Not Allowed

Process	Interoperability Allowed (CLOPT -t)	Interoperability Not Allowed
Domain gateway (GWTDOMAIN)	When the domain gateway sets up a connection to a pre-release 7.1 remote domain gateway, it authenticates the remote domain gateway using a pre-release 7.1 authentication protocol and then sets up the network connection. When the domain gateway receives a client request from the older domain, the domain gateway calls the impersonate user function to get authorization and auditing tokens for the client based on the `LOCAL_PRINCIPAL_NAME` (defaults to `ACCESSPOINTID`) identity configured for the remote domain access point, attaches the tokens to the client request, and then forwards the request to the destination server. The client has the same access permissions as the `LOCAL_PRINCIPAL_NAME` identity. For any outbound client request, the domain gateway strips the tokens from the request before sending the request along with the client's application key to the older domain.	The domain gateway does not set up a connection to a pre-release 7.1 remote domain gateway. No communication is possible between the newer and older domains.
System or application server	If the server receives a request from a remote client running Oracle Tuxedo pre-release 7.1 software, the server calls the impersonate user function to get authorization and auditing tokens for the client based on the client's assigned application key, and then performs the client request assuming the client passes any authorization checks.	If the server receives a request from a remote client running Oracle Tuxedo pre-release 7.1 software, the server rejects the client request. No communication is possible between the newer server and the older client.

2.8.3 Example UBBCONFIG Entries for Interoperability

In the following example, all WSHs controlled by the workstation listener (WSL) are configured for interoperability.

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

> **Note:**
>
> - Specifying Principal Names
> - Establishing a Link Between Domains
> - Setting ACL Policy
> - Security Administration Tasks
> - Security Interoperability
> - Setting Up Security in a Domains Configuration and Setting Up Connections in a Domains Configuration in *Using the Oracle Tuxedo Domains Component*

2.9 Establishing a Link Between Domains

When a domain gateway (`GWTDOMAIN`) attempts to establish a network link with another domain gateway, the following major events occur.

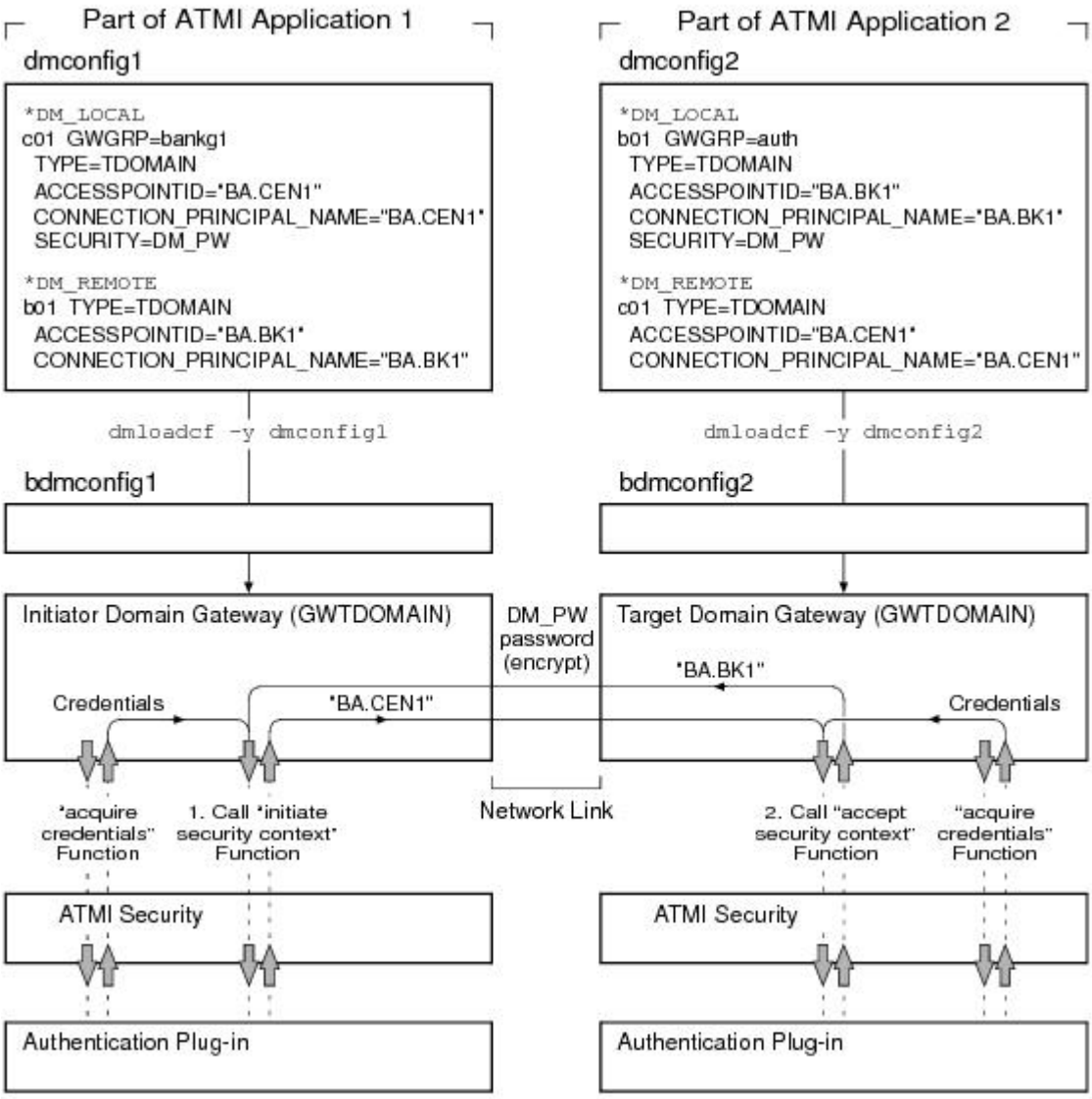
1. The *initiator* domain gateway and the *target* domain gateway exchange TLS or link-level encryption (LLE) *min-max* values to be used to set up TLS or LLE on the link between the gateways. If TLS is being used, the initiator and target domain gateways also authenticate each other through the use of TLS certificates. LLE is described in Link-Level Encryption. TLS is described in TLS Encryption.
2. The initiator and target domain gateways authenticate one another through the exchange of security tokens *assuming* that both gateways are running Oracle Tuxedo release 7.1 or later software.
   If one or both of the domain gateways are running Oracle Tuxedo pre-release 7.1 software, the gateway processes use an older (pre-release 7.1) authentication protocol when setting up the connection.

As the administrator, you use the following configuration parameter to establish a link between domain gateways running Oracle Tuxedo release 7.1 or later software.

Parameter Name	Description	Setting
`CONNECTION_PRINCIPAL_NAME in DMCONFIG (TA_DM CONNPRINCIPALNAME in DM_MIB)`	When this parameter appears in the `DM_LOCAL` section* of the `DMCONFIG` file, its value becomes the principal name of the local domain access point when setting up a connection with a remote domain access point. For default authentication plug-ins, if a value is assigned to `CONNECTION_PRINCIPAL_NAME` for the local domain access point, it must be the same as the value assigned to the `ACCESSPOINTID` parameter* for the local domain access point. If these values do not match, the local domain gateway process will not boot, and the system will generate the following userlog(3c) message: ERROR: Unable to acquire credentials.	1-511 characters. If not specified, the principal name defaults to the `ACCESSPOINTID` string for the local domain access point.

Parameter Name	Description	Setting
	When this parameter appears in the DM_REMOTE section* of the DMCONFIG file for a particular remote domain access point, its value becomes the principal name of the remote domain access point when setting up a connection with the local domain access point. For default authentication plug-ins, if a value is assigned to CONNECTION_PRINCIPAL_NAME for a remote domain access point, it must be the same as the value assigned to the ACCESSPOINTID parameter* for the remote domain access point. If these values do not match, any attempt to set up a connection between the local domain gateway and the remote domain gateway fails, and the system generates the following userlog(3c) message: ERROR: Unable to initialize administration key for domain domain_name.	1-511 characters. If not specified, the principal name defaults to the ACCESSPOINTID string for the remote domain access point.
*The DM_LOCAL section is also known as DM_LOCAL_DOMAINS; the DM_REMOTE section is also known as DM_REMOTE_DOMAINS; and the ACCESSPOINTID parameter is also known as DOMAINID.		

Figure 2-9 Establishing a Link Between Domains Using Default Authentication



> **Note:**
>
> The “Credentials” shown in the preceding figure were acquired by each domain gateway process at application booting using the `CONNECTION_PRINCIPAL_NAME` identity configured for the local domain access point.

In the preceding figure, notice that the information exchanged between the initiator and target domain gateways involves the `CONNECTION_PRINCIPAL_NAME` strings configured for the domain gateways, as specified in the `BDMCONFIG` files. Each authentication plug-in uses the password assigned to the remote domain access point (as defined in the `DM_PASSWORDS` section of the `BDMCONFIG` file) to encrypt the string before transmitting it over the network, and uses the password assigned to the local domain access point (as defined in the `DM_PASSWORDS` section of the `BDMCONFIG` file) to decrypt the received string. The encryption algorithm used is 56-bit DES, where DES is an acronym for the Data Encryption Standard.

For the encryption/decryption operation to succeed, the assigned password for the remote domain access point in the local `BDMCONFIG` file must be the same as the assigned password for the local domain access point in the remote `BDMCONFIG` file. (Similarly, if the domain security level is set to `APP_PW`, the application passwords in the respective `TUXCONFIG` files must be identical for the encryption/decryption operation to succeed.) For the authentication process to succeed, the received string must match the `CONNECTION_PRINCIPAL_NAME` string configured for the sender.

When the domain gateways pass the security checks, the link is established, and the gateways can forward service requests and receive replies over the established link.

- Example DMCONFIG Entries for Establishing a Link

2.9.1 Example DMCONFIG Entries for Establishing a Link

In the following example, the configurations shown in the local `DMCONFIG` file are used when establishing a connection through the local domain access point `c01` and the remote domain access point `b01`.

```
*DM_LOCAL
        # <local domain access point name> <gateway group
        name> <domain type>
        # <domain id> [<connection principal name>]
        [<security>]...
        c01     GWGRP=bankg1
                TYPE=TDOMAIN
                ACCESSPOINTID="BA.CENTRAL01"
                CONNECTION_PRINCIPAL_NAME="BA.CENTRAL01"
                SECURITY=DM_PW
           .
           .
           .
        *DM_REMOTE
        # <remote domain access point name> <domain type>
        <domain id>
        # [<connection principal name>]...
        b01     TYPE=TDOMAIN
                ACCESSPOINTID="BA.BANK01"
                CONNECTION_PRINCIPAL_NAME="BA.BANK01"
```

Note:

- Specifying Principal Names
- Mandating Interoperability Policy
- Setting ACL Policy
- Security Administration Tasks
- Setting Up Security in a Domains Configuration in *Using the Oracle Tuxedo Domains Component*

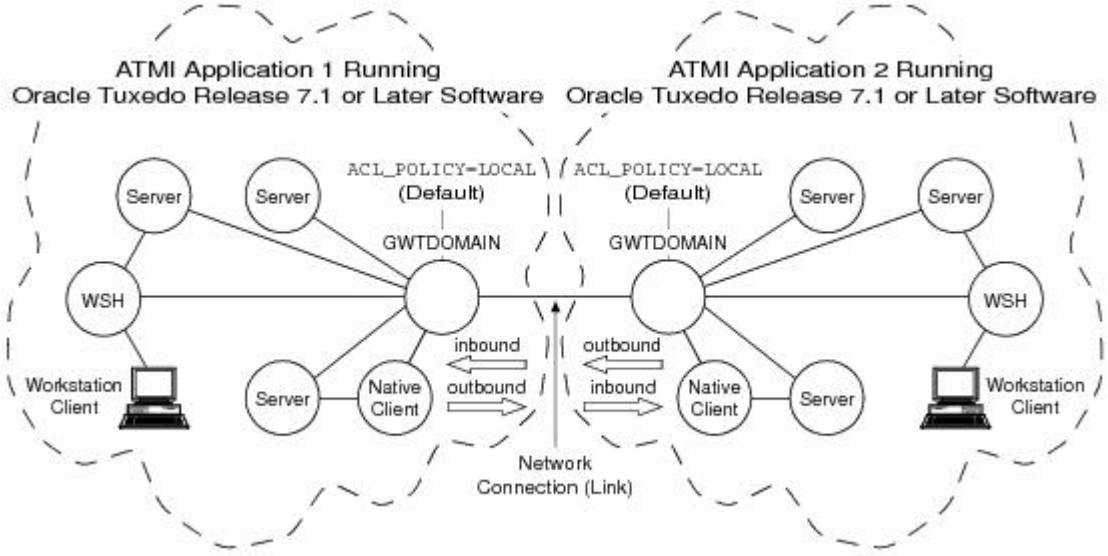
2.10 Setting ACL Policy

As the administrator, you use the following configuration parameters to set and control the access control list (ACL) policy between ATMI applications running Oracle Tuxedo release 7.1 or later software.

Parameter Name	Description	Setting
ACL_POLICY in DMCONFIG (TA_DMACLPOLICY in DM_MIB)	May appear in the DM_REMOTE section of the DMCONFIG file for each remote domain access point. Its value for a particular remote domain access point determines whether or not the local domain gateway modifies the credential (identity) of service requests received from the remote domain.	LOCAL or GLOBAL. Default is LOCAL. LOCAL means replace credential of any service request received from remote domain, and GLOBAL means pass service requests with no change.
LOCAL_PRINCIPAL_NAME in DMCONFIG (TA_DMLOCALPRINCIPALNAME in DM_MIB)	May appear in the DM_REMOTE section of the DMCONFIG file for each remote domain access point. If the ACL_POLICY parameter is set (or defaulted) to LOCAL for a particular remote domain access point, the local domain gateway replaces the credential of any service request received from the remote domain with the principal name specified in the LOCAL_PRINCIPAL_NAME parameter for this remote domain access point.	1-511 characters. If not specified, the principal name defaults to the ACCESSPOINTID string for the remote domain access point.

The following three figures illustrates how the ACL_POLICY configuration affects the operation of local domain gateway (GWTDOMAIN) processes.

Figure 2-10 Establishing a Local ACL Policy

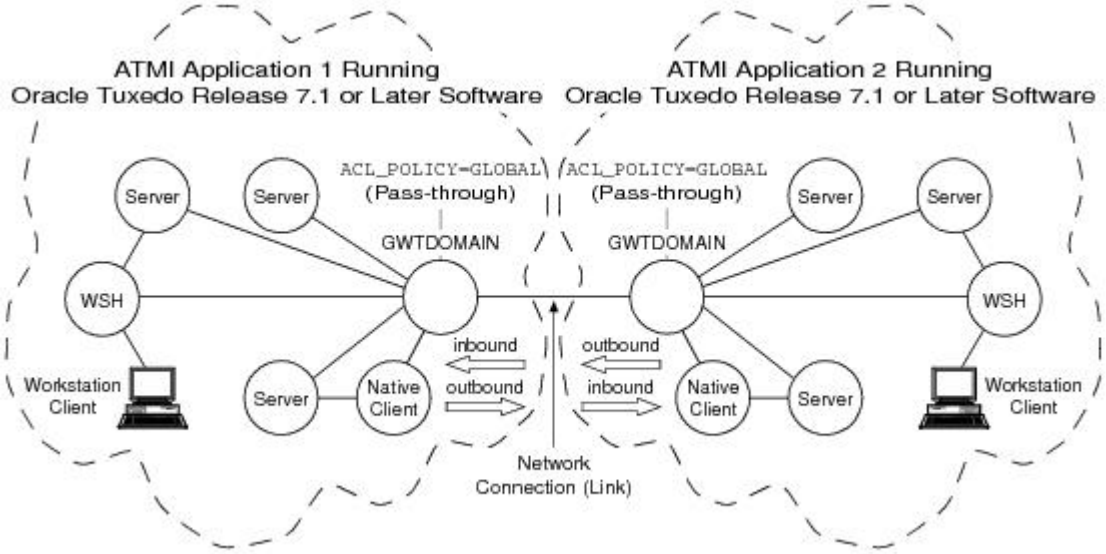


In the preceding figure, each domain gateway (`GWTDOMAIN`) modifies *inbound* client requests (requests originating from the remote application and received over the network connection) so that they take on the `LOCAL_PRINCIPAL_NAME` identity configured for the remote domain access point and thus have the same access permissions as that identity. Each domain gateway passes *outbound* client requests without change.

In this configuration, each ATMI application has an ACL database containing entries *only* for users in its own domain. One such user is the `LOCAL_PRINCIPAL_NAME` identity configured for the remote domain access point.

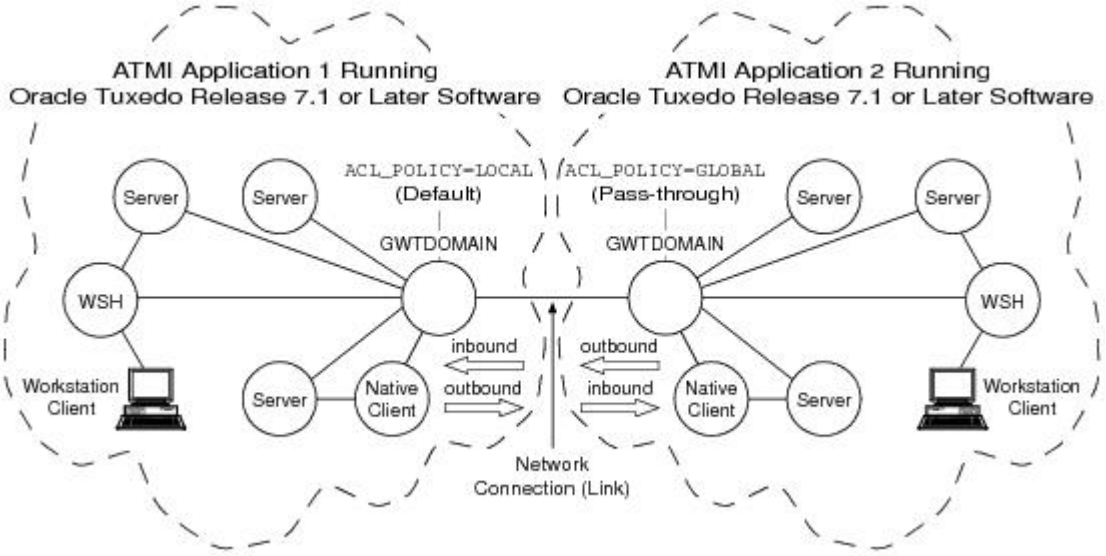
> **Note:**
>
> The preceding description also applies to ATMI applications running Oracle Tuxedo pre-release 7.1 software except that the system uses the `ACCESSPOINTID` identity configured for the remote domain access point. Essentially, the local ACL policy is hardcoded in Oracle Tuxedo release 6.5 or earlier software.

Figure 2-11 Establishing a Global ACL Policy



In the preceding figure, each domain gateway (`GWTDOMAIN`) passes inbound and outbound client requests without change. In this configuration, each ATMI application has an ACL database containing entries for users in its own domain *as well as* users in the remote domain.

Figure 2-12 Establishing a One-way Local and One-way Global ACL Policy



In the preceding figure, the domain gateway (`GWTDOMAIN`) in ATMI application 1 modifies inbound client requests so that they take on the `LOCAL_PRINCIPAL_NAME` identity configured for the remote domain access point for ATMI application 2 and thus have the same access permissions as that identity; the domain gateway passes outbound client requests without change. The domain gateway (`GWTDOMAIN`) in ATMI application 2 passes inbound and outbound client requests without change.

In this configuration, ATMI application has:

1. an ACL database containing entries *only* for users in its own domain; one such user is the LOCAL_PRINCIPAL_NAME identity configured for the remote domain access point for application 2. ATMI application
2. an ACL database containing entries for users in its own domain *as well as* users in ATMI application 1.

- Impersonating the Remote Domain Gateway
- Example DMCONFIG Entries for ACL Policy

2.10.1 Impersonating the Remote Domain Gateway

If the domain gateway receives a client request from a remote domain for which the ACL_POLICY parameter is set (or defaulted) to LOCAL in the local DMCONFIG file, the domain gateway performs the following tasks:

1. Calls the internal impersonate user function to get authorization and auditing tokens for the client *based on the* LOCAL_PRINCIPAL_NAME *identity configured for the remote domain access point.*
2. Uses these tokens to overwrite the tokens already attached to the client request.
3. Forwards the request to the destination server.

For more detail on the impersonate user function, see Establishing an Identity for an Older Client.

2.10.2 Example DMCONFIG Entries for ACL Policy

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

```
*DM_LOCAL
         # <local domain access point name> <gateway group
         name>
         # <domain type> <domain id> [<connection principal
         name>]
         # [<security>]...
         c01     GWGRP=bankg1
                 TYPE=TDOMAIN
                 ACCESSPOINTID="BA.CENTRAL01"
                 CONNECTION_PRINCIPAL_NAME="BA.CENTRAL01"
                 SECURITY=DM_PW
            .
            .
            .
         *DM_REMOTE
         # <remote domain access name> <domain type> <domain
         id>
         # [<ACL policy>] [<connection principal name>]
         # [<local principal name>]...
         b01     TYPE=TDOMAIN
```

```
ACCESSPOINTID="BA.BANK01"
ACL_POLICY=GLOBAL
CONNECTION_PRINCIPAL_NAME="BA.BANK01"
LOCAL_PRINCIPAL_NAME="BA.BANK01.BOB"
```

Note:

- Specifying Principal Names
- Mandating Interoperability Policy
- Establishing a Link Between Domains
- Security Administration Tasks

2.11 Setting Credential Policy

As the administrator, you use the following configuration parameter to set and control the credential policy between ATMI applications running Oracle Tuxedo release 8.0 or later software.

Parameter Name	Description	Setting
CREDENTIAL_POLICY in DMCONFIG (TA_DMCREDENTIALPOLICY in DM_MIB)	May appear in the DM_REMOTE section of the DMCONFIG file for each remote domain access point. Its value for a particular remote domain access point determines whether or not the local domain gateway removes the credential (identity) from a local service request destined for this remote domain access point. **Note:** The CREDENTIAL_POLICY parameter controls whether or not the local domain gateway removes the credential from a local service req	LOCAL or GLOBAL. Default is LOCAL. LOCAL means remove the credential from a local service request destined for this remote domain access point, and GLOBAL means do not remove the credential from a local service request destined for this remote domain access point.

Parameter Name	Description	Setting
	uest before sending the request to a remote domain. The ACL_POLICY parameter controls whether or not the local domain gateway replaces the credential of a service request receive	

Parameter Name	Description	Setting
	d fro m a rem ote do mai n with the prin cipa l na me spe cifie d in the LO CA L_P RIN CIP AL_ NA ME par am eter .	

2.12 Administering Authorization

Authorization enforces limitations on user access to resources or facilities within an ATMI application in accordance with application-specific rules. Only when users are authenticated to join an ATMI application does authorization go into effect.

The procedures for administering authorization depend upon the underlying authorization system of the ATMI application. For procedures to administer a custom authorization system, see the documentation for that system. For procedures to administer the default authorization system, see Administering Default Authentication and Authorization.

Note:

- Authorization
- Default Authentication and Authorization
- Administering Default Authentication and Authorization
- Security Administration Tasks
- Security Compatibility

2.13 Administering Link-Level Encryption

Link-level encryption establishes data privacy for messages moving over the network links that connect the machines in an ATMI application. There are three levels of link-level encryption (LLE) security: 0-bit (no encryption), 56-bit, and 128-bit.

LLE applies to the following types of ATMI links:

- Workstation client to workstation handler (WSH)
- Bridge-to-Bridge
- Administrative utility (such as `tmboot`) to `tlisten`
- Domain gateway to domain gateway
- Understanding LLE min and max Values
- How to Configure LLE on Workstation Client Links
- How to Configure LLE on Bridge Links
- How to Configure LLE on tlisten Links
- How to Configure LLE on Domain Gateway Links

2.13.1 Understanding LLE min and max Values

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

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

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

For more information, see How LLE Works and Encryption Key Size Negotiation.

2.13.2 How to Configure LLE on Workstation Client Links

If Workstation clients are included in an application, the administrator must configure one or more workstation listeners (WSLs) to listen for connection requests from Workstation clients. Each WSL uses one or more associated workstation handlers (WSHs) to handle the Workstation client workload. Each WSH can manage multiple Workstation clients by

multiplexing all requests and replies with a particular Workstation client over a single connection.

As the administrator, you enable Workstation client access to the ATMI application by specifying a WSL server in the SERVERS section of the application's UBBCONFIG file. You need to specify the -z and -Z command-line options for the WSL server if you want to override the defaults for the LLE *min* and *max* parameters. (See Understanding LLE min and max Values for details.) Of course, link-level encryption is possible only if LLE is installed on both the local machine and the Workstation client.

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

To configure LLE on Workstation client links, follow these steps:

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

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

3. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.

In the preceding example, when tmloadcf(1) starts the ATMI application, it passes the "-A -- -z *min* -Z *max* " command-line options to the WSL server. When establishing a network link between a Workstation client and the WSH, the Workstation client and WSL negotiate the key size until they agree on the largest key size supported by both.

See WSL(5), WS_MIB(5), and UBBCONFIG(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference* for additional information.

2.13.3 How to Configure LLE on Bridge Links

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

As the administrator, you place an entry in the NETWORK section of the UBBCONFIG file for each machine in an ATMI application on which a Bridge server resides. You need to specify the MINENCRYPTBITS and MAXENCRYPTBITS optional run-time parameters for the Bridge server if you want to override the defaults for the LLE *min* and *max* parameters. (See Understanding LLE min and max Values for details.) Of course, Bridge-to-Bridge link-level encryption is possible only if LLE is installed on the machines where the Bridge servers reside.

To configure LLE on Bridge links, follow these steps:

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

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

   LMID is the logical machine where the Bridge server resides; it has direct access to the network device specified in the BRIDGE parameter.
3. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.

In the preceding example, when tmboot(1) starts the ATMI application, the Bridge server reads the TUXCONFIG file to access various parameters, including MINENCRYPTBITS and MAXENCRYPTBITS. When establishing a network link with a remote Bridge server, the local and remote Bridge servers negotiate the key size until they agree on the largest key size supported by both.

See TM_MIB(5) and UBBCONFIG(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference* for additional information.

2.13.4 How to Configure LLE on tlisten Links

tlisten(1) is a network-independent *listener* process that provides connections between nodes of a multiple-machine application, on which administrative utilities such as tmboot(1) can run. The application administrator installs tlisten on all machines defined in the NETWORK section of the UBBCONFIG file.

To configure LLE on tlisten links, follow the steps given in the previous topic, How to Configure LLE on Bridge Links. If you so desire, you can start a separate instance of tlisten on the local machine by entering a command such as:

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

The *nlsaddr* value must be the same as that specified for the NLSADDR parameter for this machine in the NETWORK section of the UBBCONFIG file. See tlisten(1) in the *Oracle Tuxedo Command Reference*, and TM_MIB(5) and UBBCONFIG(5) in the Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference for additional information.

2.13.5 How to Configure LLE on Domain Gateway Links

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

A domain gateway belongs to a *domain gateway group*, for which a Domains configuration file is required. A domain gateway group represents a local domain access point that communicates with one or more remote domain access points. Like the application configuration files, UBBCONFIG and TUXCONFIG, a Domains configuration file is created in text

format and then converted to binary format. The text and binary files are referred to as DMCONFIG and BDMCONFIG, respectively. The DMCONFIG and BDMCONFIG files, and the environment variables associated with them, are described on reference page DMCONFIG(5) in *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

As the administrator, you must place an entry in the DM_TDOMAIN section of the DMCONFIG file for each:

- Local domain access point to accept requests for local services from remote domain access points
- Remote domain access point accessible by a defined local domain access point
- TDomain session between specific local and remote access points

You need to specify the MINENCRYPTBITS and MAXENCRYPTBITS optional run-time parameters for each domain access point and TDomain session for which you want to override the defaults for the LLE *min* and *max* parameters. (See Understanding LLE min and max Values for details.) Of course, domain-to-domain link-level encryption is possible only if LLE is installed on the machines where the domains reside.

To configure LLE on domain gateway links, follow these steps:

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

```
*DM_TDOMAIN
         # Local network addresses
         LDOM      NWADDR="local_domain_network_address"
                   NWDEVICE="local_domain_device" MINENCRYPTBITS=min
                   MAXENCRYPTBITS=max
                   .
                   .
                   .
        # Remote network addresses
        RDOM       NWADDR="remote_domain_network_address"
                   NWDEVICE="remote_domain_device" MINENCRYPTBITS=min
                   MAXENCRYPTBITS=max
                   .
                   .
                   .
        # TDomain network addresses
        RDOM       NWADDR="remote_domain_network_address"
                   NWDEVICE="remote_domain_device"
CONNECTION_POLICY=ON_START
                   LACCESSPOINT="local_domain_access_point_identifier"
FAILOVERSEQ=100
                   MINENCRYPTBITS=min MAXENCRYPTBITS=max
LDOM is replaced with a local domain access point identifier, and RDOM is
replaced with a remote domain access point identifier
```

3. Load the configuration by running dmloadcf(1). The dmloadcf command parses DMCONFIG and loads the binary BDMCONFIG file to the location referenced by the BDMCONFIG variable.

In the preceding example, when tmboot(1) starts the ATMI application, each domain gateway reads the BDMCONFIG file to access various parameters, including MINENCRYPTBITS and MAXENCRYPTBITS, and propagates those parameters to its local and remote domains. When the

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

See DMCONFIG(5) in *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference* for additional information. Also, see Setting Up Security in a Domains Configuration in *Using the Oracle Tuxedo Domains Component*.

See Also:

- Link-Level Encryption
- Security Administration Tasks
- Security Interoperability
- Security Compatibility

2.14 Administering TLS Encryption

TLS encryption establishes data privacy for messages moving between the machines in an ATMI application. The industry-standard TLS 1.0 protocol is used for TLS encryption. Customers can used 256-bit, 128-bit, and 56-bit TLS ciphers.

- Understanding TLS min and max Values
- How to Configure TLS on Workstation Client Links
- How to Configure TLS on Bridge Links
- How to Configure TLS on tlisten Links
- How to Configure TLS on Domain Gateway Links
- Development Process for the TLS Protocol
- Creating an Oracle Wallet
- Runtime Creation of an Oracle Wallet
- Use of the TUXCREATEWALLET Environment Variable
- Debugging TLS Connection Problems

2.14.1 Understanding TLS min and max Values

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

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

If you want to change the defaults, you can do so by assigning new values to *min* and *max* in the `UBBCONFIG` file for your application. For more information, see How the SSL Protocol Works and Encryption Key Size Negotiation.

2.14.2 How to Configure TLS on Workstation Client Links

To configure TLS on Workstation client links, follow these steps:

1. Ensure that you are working on the ATMI application `MASTER` machine and that the application is inactive.
2. `SEC_PRINCIPAL_NAME`, `SEC_PRINCIPAL_LOCATION`, and `SEC_PRINCIPAL_PASSVAR` parameters must be specified. This may be done in the `*RESOURCES`, `*MACHINES`, `*GROUPS`, or `*SERVERS` sections.

> **Note:**
>
> In general, it is recommended to specify these parameters at the highest level possible to avoid duplicating information in the `UBBCONFIG` and to avoid multiple password prompts if running `tmloadcf` interactively.

3. Open `UBBCONFIG` with a text editor and add the following lines to the `SERVERS` sections:

```
*SERVERS
         WSL    SRVGRP="group_name" SRVID=server_number ...
         CLOPT="-A -- -z min -Z max -n <network_address> -S <secure port>
[-a] [-R <renegotiation_interval>] ..."
```

If the secure port is set to the same port used in the network address then the WSL will accept only TLS connections; if different ports are used, the same WSL can accept both non-TLS and TLS connections.

The WSC must set the `SEC_PRINCIPAL_LOCATION`, `SEC_PRINCIPAL_NAME` and/or `SEC_PRINCIPAL_PASSWORD` environment variables as appropriate.

All workstation clients using TLS must specify the list of trusted certificate(s) used to verify the credentials presented by the WSH. When using legacy security credentials, the location is specified via the plugin framework `certificate_validation` interface and does not require setting any environment variables. When the Oracle Wallet is used for security credentials, the trusted certificates are contained in the Oracle Wallet. The `SEC_PRINCIPAL_LOCATION` and `SEC_PRINCIPAL_NAME` environment variables are used to locate the wallet as described in Runtime Creation of an Oracle Wallet. The `SEC_PRINCIPAL_PASSWORD` environment variable is used to open the wallet.

> **Note:**
>
> - It is possible for `SEC_PRINCIPAL_NAME` to be unset, in which case it will be interpreted as a 0-length string.
> - If legacy security credentials for 1-way TLS are converted to an Oracle Wallet at runtime and the `SEC_PRINCIPAL_PASSWORD` environment variable is not set at the time of creation, then a default password `TrustedCertsOnlyNoPWNeeded` is used to create the wallet. Such a wallet can be subsequently accessed without setting the `SEC_PRINCIPAL_PASSWORD` environment variable.

If the WSL -a (mutual authentication) option is being used then the WSC must also specify the location of its own certificate and private key. Regardless of whether legacy security credentials or the Oracle Wallet are being used, the SEC_PRINCIPAL_LOCATION, SEC_PRINCIPAL_NAME, and SEC_PRINCIPAL_PASSWORD environment variables must be set to access these credentials.

It is possible for SEC_PRINCIPAL_NAME to be unset, in which case it will be interpreted as a 0-length string.

4. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.

2.14.3 How to Configure TLS on Bridge Links

To configure TLS on Bridge links, follow these steps:

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

```
*RESOURCES
          OPTIONS SSL,LAN
          SSL_RENEGOTIATION (optional)    [value]
        *NETWORK
         LMID  NADDR="bridge_network_address" BRIDGE="bridge_device"
               NLSADDR="listen_network_address"
               MINENCRYPTBITS=min MAXENCRYPTBITS=max
```

SEC_PRINCIPAL_NAME, SEC_PRINCIPAL_LOCATION, and SEC_PRINCIPAL_PASSVAR must be specified in the *RESOURCES and/or*MACHINES sections.

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

3. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.

2.14.4 How to Configure TLS on tlisten Links

To configure TLS on tlisten links, follow the steps given in the previous topic, How to Configure SSL on Bridge Links. You must enter the following command:

```
tlisten -l nlsaddr [-z min -Z
         max][-s][-c <sec_principal_location>][-n
         <sec_principal_name>][-p
         <sec_principal_passvar>]
```

> **Note:**
>
> The -s option specifies an TLS connection instead of an LLE connection.

The -c, -n, and -p options specify TLS security principal information and must match the values specified for the SEC_PRINCIPAL_NAME, SEC_PRINCIPAL_LOCATION, and SEC_PRINCIPAL PASSVAR in the UBBCONFIG file.

2.14.5 How to Configure TLS on Domain Gateway Links

To configure TLS on domain gateway links, follow these steps:

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

```
*DM_TDOMAIN
         # SSL DEFAULT: NWPROTOCOL={SSL|SSL_ONE_WAY}
         SSL_RENEGOTIATION = [value]


 # Local network addresses
         LDOM   NWADDR="local_domain_network_address"
                 NWDEVICE="local_domain_device" MINENCRYPTBITS=min
                 MAXENCRYPTBITS=max


  # Remote network addresses
           RDOM     NWADDR="remote_domain_network_address"
                    NWDEVICE="remote_domain_device" MINENCRYPTBITS=min
                    MAXENCRYPTBITS=max
                    .
                    .
                    .
           # TDomain network addresses
           RDOM     NWADDR="remote_domain_network_address"
                    NWDEVICE="remote_domain_device"
CONNECTION_POLICY=ON_START
                    LACCESSPOINT="local_domain_access_point_identifier"
FAILOVERSEQ=100
                    MINENCRYPTBITS=min MAXENCRYPTBITS=max


 LDOM is replaced with a local domain access point identifier, and RDOM is
replaced with a remote domain access point identifier.
```

3. SEC_PRINCIPAL_NAME, SEC_PRINCIPAL_LOCATION , and SEC_PRINCIPAL_PASSWORD must be specified in the UBBCONFIG file.
4. Load the configuration by running dmloadcf(1) . The dmloadcf command parses DMCONFIG and loads the binary BDMCONFIG file to the location referenced by the BDMCONFIG variable.

2.14.6 Development Process for the TLS Protocol

Using the TLS protocol in a Tuxedo application is primarily an administration process. The following table describes the administration steps required to set up the infrastructure required to use the TLS protocol and configure the servers and clients in your application to use TLS.

For a detailed description of the administration steps, see Managing Public Key Security and Configuring the SSL Protocol in *Using Security in CORBA Applications*.

Once the administration steps are complete, you can use either password authentication or certificate authentication in your Tuxedo application. The steps are similar for CORBA application authentication. For more information, see Writing a CORBA Application That Implements Security in *Using Security in CORBA Applications*.

> **Note:**
>
> If you are using the Oracle CORBA C++ ORB as a server application, the ORB can also be configured to use the TLS protocol. For more information, see Configuring the SSL Protocol in *Using Security in CORBA Applications*.

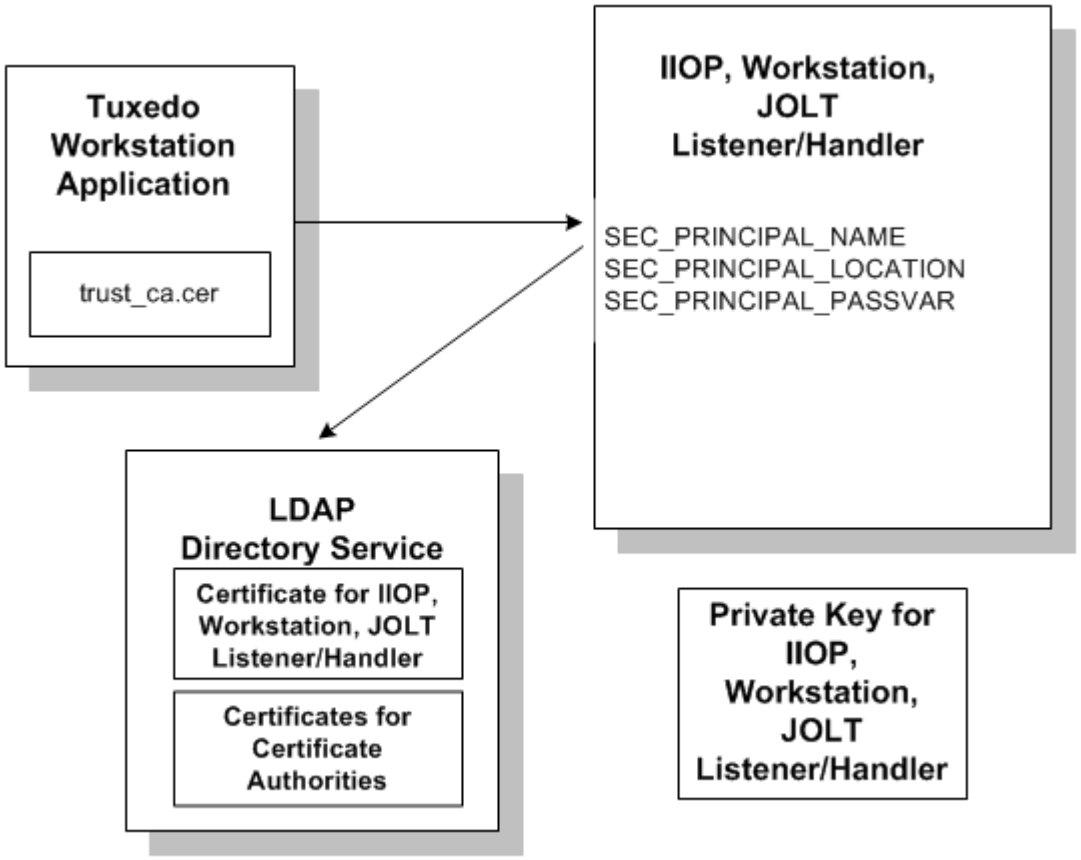
Table 2-2 Administration Steps for the TLS Protocol

Step	Description
1	Set up an LAP-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 TLS protocol.
3	Obtain a digital certificate and private key for the Oracle Tuxedo application from a certificate authority.
4	Publish the digital certificates for the Oracle Tuxedo application and the certificate authority in the LAP-enabled directory service.
5	Define the `SEC_PRINCIPAL_NAME`, `SEC_PRINCIPAL_LOCATION`, and `SEC_PRINCIPAL_PASSVAR` parameters for the Tuxedo server process in the `UBBCONFIG` file.
6	Change to "Set the UBBCONFIG parameters, DMCONFIG parameters, WSL CLOPT, JSL CLOPT, or ISL CLOPT so that TLS is turned on.
7	Define a port for TLS communication in the appropriate configuration file or server CLOPT.
8	Create a Trusted Certificate Authority file (`trust_ca.cer`) that defines the certificate authorities trusted by the Oracle Tuxedo application.
9	Change to "Use the `tmloadcf` and/or `dmloadcf` commands to load the appropriate configuration file(s).
10	Optionally, create a Peer Rules file (`peer_val.rul`) for the Oracle Tuxedo product.
11	Optionally, modify the LDAP Search filter file to reflect the directory hierarchy in place in your enterprise.

If you use the TLS 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" in *Using Security in ATMI Applications.*.

The following figure illustrates the configuration of a Tuxedo application that uses the TLS protocol.

Figure 2-13 Configuration for Using the TLS Protocol in a Tuxedo Application



2.14.7 Creating an Oracle Wallet

An Oracle Wallet can be created in any of the following ways:

- Using the **owm** graphical tool for those customers who have installed Oracle Database
- Using the **orapki** command line tool for those customers who have installed Oracle Database
- Using **openssl** or another third party tool
- Automatically at execution time by conversion of security credentials used in Tuxedo 11g or earlier releases.

- Creating an Oracle Wallet with orapki
- Creating an Oracle Wallet with openssl

2.14.7.1 Creating an Oracle Wallet with orapki

For information about how to create an Oracle Wallet using orapki, see the orapki Utility in Oracle Database Advanced Security Administrator's Guide.

Oracle Tuxedo wallets require a password, so the **Auto Login** option should not be used. `orapki` and `owm` can be used to generate wallet with a new private key and certificate, but

current versions of these tools cannot import a previously used private key and certificate into a wallet. If it is necessary to import a preexisting private key and certificate pair into a wallet, use runtime conversion, openssl, or another third party tool.

2.14.7.2 Creating an Oracle Wallet with openssl

An example of an `openssl` command that can be used to create an Oracle Wallet is as follows:

Listing 2-1 Example of Creating an Oracle Wallet with openssl

```
openssl pkcs12 \
         -export \
         -chain \
         -inkey private_key_file.pem \
         -in certificate_file.pem \
         -CAfile trusted_certificate_file.pem \
         -out ewallet.p12 \
         -passin Spass:private_key_password \
         -passout pass:wallet_password \
```

Where,

- `-export`: indicates that a PKCS 12 file is being created.
- `-chain`: specifies that an attempt is made to include the entire certificate chain of the user certificate.
- `-inkey`: specifies the private key file.
- `-in`: specifies the file that contains the user certificate and any other certificates in the certificate chain.

> **Note:**
>
> If the private key and the certificate chain are in the same file, the `-inkey` and `-in` parameters can specify the same file.

- `-CAfile`: specifies a file containing trusted certificates.
- `-out`: specifies the output file name, which must be **ewallet.p12** for an Oracle Wallet.
- `-passin`: specifies the password for the private key file.
- `-passout`: specifies the password for the newly created wallet.

> **Note:**
>
> - If there is any concern about other users executing "`ps`" while `openssl` is running, then the `-passin` and `-passout` parameters should be omitted and `openssl` will prompt for the passwords.
> - When you create an Oracle Wallet with `openssl`, the "`-passin`" parameter must have the same value as the "`-passout`" parameter, for Oracle Wallet does not distinguish wallet password from private key password.

2.14.8 Runtime Creation of an Oracle Wallet

When the SEC_PRINCIPAL_LOCATION configuration parameter or the workstation client SEC_PRINCIPAL_LOCATION environment variable does not point to an Oracle Wallet, Tuxedo looks for legacy security credentials and attempts to create an Oracle Wallet as follows:

- As in previous releases, SEC_PRINCIPAL_LOCATION points to the private key file for the process. A private key file is mandatory for processes that will be on the server side of an TLS connection or that will be on the client side of the connection when mutual authentication is used. It is optional for processes that will be on the client side of a one-way TLS connection. The value of the SEC_PRINCIPAL_PASSVAR configuration file environment variable (or the workstation client SEC_PRINCIPAL_PASSWORD environment variable) will be used to decrypt the private key.
- The certificate chain for the process is obtained via the plugin framework passing the value of SEC_PRINCIPAL_NAME as input (In the default plugin framework implementation this uses LDAP). A certificate chain is mandatory for processes that will be on the server side of an TLS connection or that will be on the client side of the connection when mutual authentication is used. It is optional for processes that will be on the client side of a one-way TLS connection.
- The trusted certificates for the process are contained in the file specified as the caCertificateFile parameter of the plugin framework certificate_validation interface. The default caCertificateFile is $TUXDIR/udataobj/security/certs/trust_ca.cer. Trusted certificates need to exist for TLS servers and TLS clients.

A PKCS12 wallet file is created using the process' private key (if any) and user certificate (if any) as well as the other certificates in the chain and the trusted certificates.

During Oracle wallet runtime creation, SEC_PRINCIPAL_LOCATION is used to specify the location of the newly created wallet; it must be defined as either server's or client's own private key.

For example, if there is a private key file "ISH_tuxqa.pem" in "/home/tuxedo/myapp", you should define SEC_PRINCIPAL_LOCATION="/home/tuxedo/myapp/ISH_tuxqa.pem". In this way, the wallet is created at /home/tuxedo/myapp/wallet.ISH_tuxqa.pem/ewallet.p12.

> **Note:**
>
> - If you want to create the wallet manually with the method mentioned in Creating an Oracle Wallet, you must follow the same rules as above to create your wallet at a proper directory; otherwise, the wallet cannot be found.
> - Exceptionally, when creating the wallet manually, besides defining the SEC_PRINCIPAL_LOCATION as a private key file, you can also define it as a directory. In this way, both SEC_PRICIPAL_LOCATION and SEC_PRINCIPAL_NAME will be used to locate the wallet.
> - For example, if you define SEC_PRINCIPAL_LOCATION="/home/tuxedo/myapp" and SEC_PRINCIPAL_NAME="ISH_tuxqa", you should copy your manually created wallet to /home/tuxedo/myapp/wallet.ISH_tuxqa/ewallet.p12; otherwise, it cannot be found.

2.14.9 Use of the TUXCREATEWALLET Environment Variable

The conversion of legacy security credentials to the Oracle Wallet format is affected by the `TUXCREATEWALLET` environment variable, which may have the following settings:

- `TUXCREATEWALLET=KEEP` or `TUXCREATEWALLET=YES` or `TUXCREATEWALLET` unset: If a wallet does not exist but old-style security credentials do exist then convert the legacy security credentials to a wallet. This is the default behavior. The directory where the wallet is created will have 700 permissions and the ewallet.p12 file will have 600 permissions. The user must have proper permissions to read any existing wallet or to create a wallet. If `ULOG_SSLINFO=y` is set then the following message will be logged:

  ```
  LIBTUX_CAT:6908: INFO: Security credentials for principal
  name have been converted to Oracle Wallet wallet_directory
  ```

  On subsequent process invocations the newly created wallet will be used so that the legacy security credentials do not need to be recreated.
- `TUXCREATEWALLET=TEMP`: If a wallet does not exist but old-style security credentials do exist create a wallet in a temporary directory and then remove the temporary file wallet once it is open. `No LIBTUX_CAT:6908` message will be logged when using this option. The TEMP option is less efficient but is needed if:
  - Old-style security credentials gotten from the plugin framework could change dynamically, or
  - The application does not want to store wallets on a local file system for security reasons or for any other reason, or
  - `SEC_PRINCIPAL_LOCATION` is located on a read-only file system.
- `TUXCREATEWALLET=NO` or `TUXCREATEWALLET=anyothervalue`: If a wallet does not exist report an error and do not look at old-style security credentials.
  The values KEEP or TEMP may be in any case but must be those 4 characters. The values `YES` or `NO` may be in the local language as is true for many other `Yes/No` environment variables in Tuxedo.

2.14.10 Debugging TLS Connection Problems

- Enabling NZ Tracing
- Connection Establishment Log Message
- Displaying the Contents of an Oracle Wallet
- Obtaining NZ Error Code Information

2.14.10.1 Enabling NZ Tracing

If the environment variable `TUXNZTRACE=8191` is set, Tuxedo will output an TLS trace for the process to a file named `trace-process_id.log`. The trace output will contain information sent across the TLS handshake process as well as encrypted application data. This trace can be very helpful in determining why a particular certificate chain is not considered valid or why there is some other error in the TLS handshake process.

2.14.10.2 Connection Establishment Log Message

If the environment variable `ULOG_SSLINFO=yes` is set, then Tuxedo will write a message to the userlog each time a TLS connection is established which includes the name of the negotiated cipher.

2.14.10.3 Displaying the Contents of an Oracle Wallet

Various tools can be used to display information about an Oracle Wallet, which is a PKCS12 file.

Openssl is available as part of the OS distribution on some operating systems and can be downloaded and compiled from source on other operating systems.

The following openssl command displays the certificates and private keys in an Oracle Wallet:

```
openssl pkcs12 -in ewallet.p12
```

`openssl` will prompt for a password to be used to open the wallet. (The option `-password pass:password` can be used to avoid the prompt but using this option could allow the password to be seen by another user on the machine who is executing the `ps` command.)

`openssl` will also prompt for a password to be used to encrypt the decrypted private key when displaying it on the terminal. The option -nodes can be used to avoid this prompt and to display the private key in unencrypted format.

Any of the certificates contained in the output of `openss pkcs12` can be copied into another file and the following command displays the fields in the certificate:

```
openssl x509 -in certificatefile -text -noout
```

Users who have Oracle Database software installed can also use the **orapki** command or the **owm** graphical command to display information about a wallet. The **orapki** command to display wallet information looks like this:

```
orapki wallet display -wallet wallet_location
```

2.14.10.4 Obtaining NZ Error Code Information

Many TLS error messages include an error code number returned by the Oracle NZ security layer. In some but not all error messages this is followed by a short text description of the NZ error number. For those error messages where no text description of the NZ error code is included, this information can be obtained by looking in the file.

```
$TUXDIR/locale/C/ORACLE.text
```

Users who have Oracle Database software installed can also use the oerr command to determine the string associated with a particular error number

Note:

- SSL Encryption
- Security Administration Tasks
- UBBCONFIG(5) Resources Section
- DM_MIB(5) `T_DM_TDOMAIN`Class
- DMCONFIG(5) `DM_TDOMAIN` section
- WS_MIB(5) `T_WSL` Class
- Using Security in CORBA Applications

2.15 Administering Public Key Security

The most effective way to make a distributed ATMI application secure is to combine link-level encryption with public key encryption. Public key encryption is the framework on which public key security is built.

Public key security allows you to incorporate message-based digital signatures and message-based encryption into your ATMI applications. Together, these capabilities provide data integrity and privacy, which are especially important when an ATMI application interacts with other ATMI applications or Workstation clients from outside the company.

- Recommended Practices for Public Key Security
- Assigning Public-Private Key Pairs
- Setting Digital Signature Policy
- Setting Encryption Policy
- Initializing Decryption Keys Through the Plug-ins
- Failure Reporting and Auditing

2.15.1 Recommended Practices for Public Key Security

- The ATMI application's operating environment largely determines the level of security achieved. For maximum safety, install hardware devices that protect private key information.
- Establish policies regarding key expiration intervals and key renewal procedures. Expiration of a Certification Authority's certificate might have a dramatic impact on system operation, and should be anticipated so updated user certificates can be issued in advance.

2.15.2 Assigning Public-Private Key Pairs

Application administrators and developers need to choose a Certification Authority to provide public-private key pairs and the digital certificates associated with them. Then they must decide how to assign the key pairs to the ATMI application. There are many options for assigning key pairs. An administrator can assign one or more of the following:

- One public-private key to an entire ATMI application

- A public-private key pair to each machine in an ATMI application
- A public-private key pair to each server in an ATMI application
- A public-private key pair to each service in an ATMI application
- A public-private key pair to each end user

Application administrators and developers are responsible for choosing a method of assigning key pairs and assigning them. Once key pairs are assigned, however, no more administrative work is required; the plug-ins for public key security distribute and manage the keys.

2.15.3 Setting Digital Signature Policy

As the administrator, you use the following configuration parameters to set the digital signature policy for your ATMI application.

Parameter Name	Description	Setting
SIGNATURE_AHEAD in UBBCONFIG (TA_SIGNATURE_AHEAD in TM_MIB)	Maximum permissible time difference between (1) the timestamp value attached to a digitally signed message buffer and (2) the time at which the message buffer is received. If the signature timestamp is too far into the future, the receiving process rejects the message buffer.	1-2147483647 seconds. Default is 3600 seconds (one hour).
SIGNATURE_BEHIND in UBBCONFIG (TA_SIGNATURE_BEHIND in TM_MIB)	Maximum permissible time difference between (1) the time at which a digitally signed message buffer is received and (2) the timestamp value attached to the message buffer. If the signature timestamp is too far into the past, the receiving process rejects the message buffer.	1-2147483647 seconds. Default is 604800 seconds (one week).
SIGNATURE_REQUIRED in UBBCONFIG (TA_SIGNATURE_REQUIRED in TM_MIB)	Determines whether a receiving process will accept *only* message buffers that are digitally signed.	Y (yes—digital signature is required) or N (no—digital signature is not required). Default is N.

- Setting a Postdated Limit for Signature Timestamps
- Setting a Predated Limit for Signature Timestamps
- Enforcing the Signature Policy for Incoming Messages
- How the EventBroker Signature Policy Is Enforced
- How the /Q Signature Policy Is Enforced
- How the Remote Client Signature Policy Is Enforced

2.15.3.1 Setting a Postdated Limit for Signature Timestamps

SIGNATURE_AHEAD is specified at the domain-wide level of the configuration hierarchy, meaning that the value you assign to it applies to all processes running in the ATMI application. Domain-wide parameters are set in the RESOURCES section in the UBBCONFIG file, and the T_DOMAIN class in the TM_MIB.

The SIGNATURE_AHEAD parameter establishes the maximum permissible time difference between (1) the timestamp attached to the incoming message buffer and (2) the current time shown on the verifying system's local clock. The minimum value is 1 second; the maximum, 2147483647 seconds. The default is 3600 seconds (one hour).

If the attached timestamp shows a time too far into the future, the signature is considered invalid. This parameter is useful for rejecting signatures that are postdated, while allowing a certain amount of leeway for unsynchronized local clocks.

- Example UBBCONFIG Entries for Postdated Limit

2.15.3.1.1 Example UBBCONFIG Entries for Postdated Limit

```
*RESOURCES
SIGNATURE_AHEAD 2400
```

2.15.3.2 Setting a Predated Limit for Signature Timestamps

SIGNATURE_BEHIND is specified at the domain-wide level of the configuration hierarchy, meaning that the value you assign to it applies to all processes running in the ATMI application. Domain-wide parameters are set in the RESOURCES section in the UBBCONFIG file, and the T_DOMAIN class in the TM_MIB.

The SIGNATURE_BEHIND parameter establishes the maximum permissible time difference between (1) the current time shown on the verifying system's local clock and (2) the timestamp attached to the incoming message buffer. The minimum value is 1 second; the maximum, 2147483647 seconds. The default is 604800 seconds (one week).

If the attached timestamp shows a time too far into the past, the signature is considered invalid. This parameter is useful for resisting replay attacks, in which a valid signed buffer is injected into the system a second time. However, in a system with asynchronous communication—for example, in a system in which disk-based queues are used—buffers signed a long time ago may still be considered valid. So, in a system with asynchronous communication, you may want to increase the SIGNATURE_BEHIND setting.

- Example UBBCONFIG Entries for Predated Limit

2.15.3.2.1 Example UBBCONFIG Entries for Predated Limit

```
*RESOURCES
        SIGNATURE_BEHIND 300000
```

2.15.3.3 Enforcing the Signature Policy for Incoming Messages

SIGNATURE_REQUIRED may be specified any of the following four levels in the configuration hierarchy:

- RESOURCES section in UBBCONFIG or T_DOMAIN class in TM_MIB
- MACHINES section in UBBCONFIG or T_MACHINE class in TM_MIB
- GROUPS section in UBBCONFIG or T_GROUP class in TM_MIB
- SERVICES section in UBBCONFIG or T_SERVICE class in TM_MIB

Setting SIGNATURE_REQUIRED to Y (yes) at a particular level means that signatures are required for all processes running at that level or below. For example, setting SIGNATURE_REQUIRED to Y for a machine named mach1 means that all processes running on mach1 will accept only incoming messages that are digitally signed.

- Set at the domain-wide level (RESOURCES section or T_DOMAIN class), this parameter covers all application services advertised within the domain, including those advertised by gateway processes. The default is N.
- Set at the machine level (MACHINES section or T_MACHINE class), this parameter covers all application services advertised on a particular machine, including those advertised by gateway processes. The default is N.
- Set at the group level (GROUPS section or T_GROUP class), this parameter covers all application services advertised by a particular group, including those advertised by gateway processes. The default is N.
- Set at the service level (SERVICES section T_SERVICE class), this parameter covers all instances of a particular service advertised within the domain, including those advertised by gateway processes. The default is N.

You may specify both SIGNATURE_REQUIRED=Y and ENCRYPTION_REQUIRED=Y *together* at the domain-wide level, machine level, group level, or service level. See Enforcing the Encryption Policy for Incoming Messages for a description of ENCRYPTION_REQUIRED.

2.15.3.3.1 Qualifier

The enforcement policy for SIGNATURE_REQUIRED applies only to application services, application events, and application enqueue requests. It does not apply to system-generated service invocations and system event postings.

2.15.3.3.2 Example

To configure SIGNATURE_REQUIRED for a machine named mach1, follow these steps:

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

```
*MACHINES
         mach1     LMID="machine_logical_name"
                   TUXCONFIG="absolute_path_name_to_tuxconfig_file"
                   TUXDIR="absolute_path_name_to_BEA_Tuxedo_directory"
                   APPDIR="absolute_path_name_to_application_directory"
                   SIGNATURE_REQUIRED=Y
```

3. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.

In the preceding example, when tmboot(1) starts the ATMI application, it passes the SIGNATURE_REQUIRED=Y parameter to the machine named mach1. At that point, all application services advertised by mach1, including those advertised by gateway processes, are allowed to accept only messages that include valid digital signatures. If a process controlled by mach1

receives a message that does *not* include a valid digital signature, the system takes the following actions:

- Generates a userlog(3c) message (severity `WARN`)
- Discards the buffer as if it were never received by the process
- A NULL (empty) buffer cannot be digitally signed, meaning that the system rejects any NULL buffer received by a process requiring digital signatures, in the manner stated in the preceding bullet list.

2.15.3.4 How the EventBroker Signature Policy Is Enforced

When digital signatures are attached to a posted message buffer, these signatures are preserved and forwarded along with the message buffer to subscribers for the relevant event.

If theTMUSREVT(5) system server is running in a domain, machine, or server group that requires digital signatures, it rejects any incoming posting without a `TPSIGN_OK` composite signature status—see Understanding the Composite Signature Status.

Possible subscription notification actions that the `TMUSREVT` server might take include invoking a service or enqueuing a message. If the target service or queue requires a valid digital signature, but one is not attached to the posted message, the subscription notification action fails.

System events (events that are posted by the system itself and processed by the `TMSYSEVT` server) may be digitally signed. The administrative policies regarding digital signature do *not* apply to the TMSYSEVT(5) server.

2.15.3.5 How the /Q Signature Policy Is Enforced

When digital signatures are attached to a queued buffer, the signatures are preserved in the queue and forwarded to the dequeuing process. Also, if a message is processed by TMQFORWARD(5) to invoke a service, signatures are preserved.

If the TMQUEUE(5)system server is running in a domain, machine, or server group that requires digital signatures, it rejects any incoming enqueue request without a `TPSIGN_OK` composite signature status—see Understanding the Composite Signature Status. In addition, the `TMQUEUE` server requires a digital signature if such a policy is in effect for the service name associated with the queue space.

2.15.3.6 How the Remote Client Signature Policy Is Enforced

If the workstation handler (WSH) is running in a domain, machine, or server group that requires digital signatures, it rejects any incoming message buffer containing application data without a `TPSIGN_OK` composite signature status—see Understanding the Composite Signature Status.

2.15.4 Setting Encryption Policy

As the administrator, you use the following configuration parameter to set the encryption policy for your ATMI application.

Parameter Name	Description	Setting
ENCRYPTION_REQUIRED in UBBCONFIG (TA_ENCRYPTION_REQUIRED in TM_MIB)	Determines whether a receiving process will accept *only* message buffers that are encrypted.	Y (yes—encryption is required) or N (no—encryption is not required). Default is N.

- Enforcing the Encryption Policy for Incoming Messages
- How the EventBroker Encryption Policy Is Enforced
- How the /Q Encryption Policy Is Enforced
- How the Remote Client Encryption Policy Is Enforced

2.15.4.1 Enforcing the Encryption Policy for Incoming Messages

ENCRYPTION_REQUIRED may be specified at any of the following four levels in the configuration hierarchy:

- RESOURCES section in UBBCONFIG or T_DOMAIN class in TM_MIB
- MACHINES section in UBBCONFIG or T_MACHINE class in TM_MIB
- GROUPS section in UBBCONFIG or T_GROUP class in TM_MIB
- SERVICES section in UBBCONFIG or T_SERVICE class in TM_MIB

Setting ENCRYPTION_REQUIRED to Y (yes) at a particular level means that encryption is required for all processes running at that level or below. For example, setting ENCRYPTION_REQUIRED to Y for a machine named mach1 means that all processes running on mach1 will accept only incoming messages that are encrypted.

- Set at the domain-wide level (RESOURCES section or T_DOMAIN class), this parameter covers all application services advertised within the domain, including those advertised by gateway processes. The default is N.
- Set at the machine level (MACHINES section or T_MACHINE class), this parameter covers all application services advertised on a particular machine, including those advertised by gateway processes. The default is N.
- Set at the group level (GROUPS section or T_GROUP class), this parameter covers all application services advertised by a particular group, including those advertised by gateway processes. The default is N.
- Set at the group level (GROUPS section or T_GROUP class), this parameter covers all application services advertised by a particular group, including those advertised by gateway processes. The default is N.
- Set at the service level (SERVICES section T_SERVICE class), this parameter covers all instances of a particular service advertised within the domain, including those advertised by gateway processes. The default is N.

You may specify both ENCRYPTION_REQUIRED=Y and SIGNATURE_REQUIRED=Y *together* at the domain-wide level, machine level, group level, or service level. See Enforcing the Signature Policy for Incoming Messages for a description of SIGNATURE_REQUIRED.

- Qualifier
- Example

2.15.4.1.1 Qualifier

The enforcement policy for `ENCRYPTION_REQUIRED` applies only to application services, application events, and application enqueue requests. It does not apply to system-generated service invocations and system event postings.

2.15.4.1.2 Example

To configure `ENCRYPTION_REQUIRED` for a server group named `STDGRP`, follow these steps:

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

```
*GROUPS
          STDGRP    LMID="machine_logical_name"
                    GRPNO="server_group_number"
                    ENCRYPTION_REQUIRED=Y
```

3. Load the configuration by running tmloadcf(1). The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the TUXCONFIG variable.

In the preceding example, when tmboot(1) starts the ATMI application, it passes the `ENCRYPTION_REQUIRED=Y` parameter to the server group named `STDGRP`. At that point, all application services advertised by `STDGRP`, including those advertised by gateway processes, are allowed to accept only messages protected by an encryption envelope. If a process controlled by `STDGRP` receives an unencrypted message, the system takes the following actions:

- Generates a userlog(3c)message (severity `ERROR`)
- Discards the buffer as if it were never received by the process

> **Note:**
>
> A NULL (empty) buffer cannot be encrypted, meaning that the system rejects any NULL buffer received by a process requiring encryption, in the manner stated in the preceding bullet list.

2.15.4.2 How the EventBroker Encryption Policy Is Enforced

When a posted message buffer is encrypted, encryption envelopes are preserved and forwarded, along with the encrypted message content, to subscribers for the relevant event.

If the TMUSREVT(5) system server is running in a domain, machine, or server group that requires encryption, it rejects any incoming posting message that is not encrypted.

Possible subscription notification actions that the `TMUSREVT` server might take include invoking a service or enqueuing a message. If the target service or queue requires encrypted input, but the posted message is not encrypted, the subscription notification action fails. Also, if the subscriber does not possess an appropriate decryption key, the event notification action fails.

System events (events that are posted by the system itself and processed by the TMSYSEVT server) may be encrypted. The administrative policies regarding encryption do *not* apply to the TMSYSEVT(5) server.

2.15.4.3 How the /Q Encryption Policy Is Enforced

When a queued message buffer is encrypted, this status is preserved in the queue, and the buffer is forwarded, in encrypted form, to the dequeuing process. Also, if a message is processed by TMQFORWARD(5) to invoke a service, encryption status is preserved.

If the TMQUEUE(5) system server is running in a domain, machine, or server group that requires encryption, it rejects any incoming enqueue request that is not encrypted. In addition, the TMQUEUE server requires encryption if such a policy is in effect for the service name associated with the queue space.

2.15.4.4 How the Remote Client Encryption Policy Is Enforced

If the workstation handler (WSH) is running in a domain, machine, or server group that requires encryption, it rejects any incoming message buffer containing an unencrypted application data buffer.

2.15.5 Initializing Decryption Keys Through the Plug-ins

As the administrator, you use the following configuration parameters to specify principal names and decryption keys for the system processes running in your ATMI application.

Parameter Name	Description	Setting
SEC_PRINCIPAL_NAME in UBBCONFIG (TA_SEC_PRINCIPAL_NAME in TM_MIB)	The name of the target principal, which becomes the identity of one or more system processes.	1-511 characters.
SEC_PRINCIPAL_LOCATION in UBBCONFIG (TA_SEC_PRINCIPAL_LOCATION in TM_MIB)	The location of the file or device where the decryption (private) key for the target principal resides.	0-1023 characters. If not specified, defaults to a NULL (zero length) string.
SEC_PRINCIPAL_PASSVAR in UBBCONFIG (SEC_PRINCIPAL_PASSVAR in TM_MIB)	The variable in which the password for the target principal is stored.	0-31 characters. If not specified, defaults to a NULL (zero length) string.

This trio of configuration parameters can be specified at any of the following four levels in the configuration hierarchy:

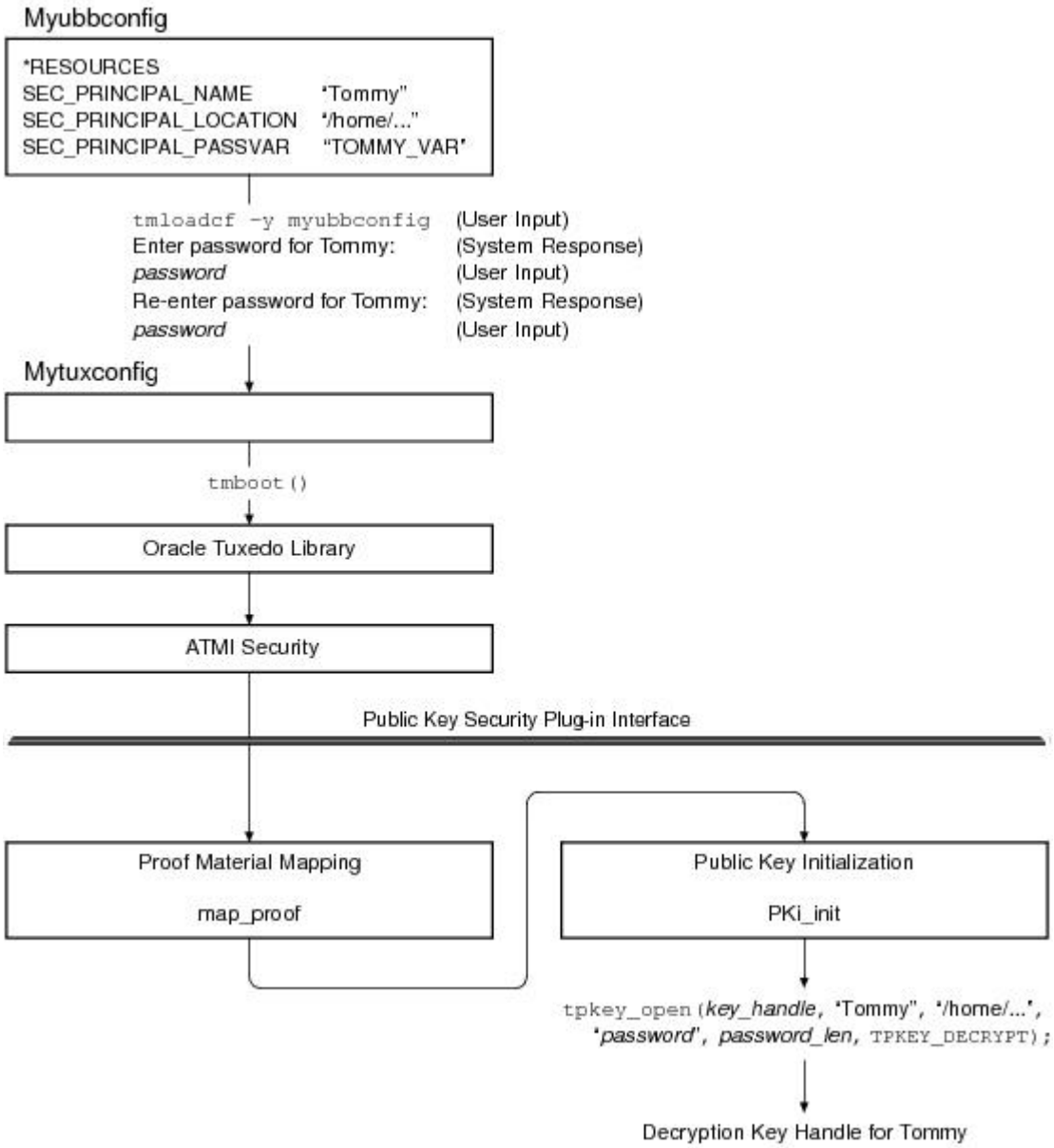
- RESOURCES section in UBBCONFIG or T_DOMAIN class in TM_MIB
- MACHINES section in UBBCONFIG or T_MACHINE class in TM_MIB
- GROUPS section in UBBCONFIG or T_GROUP class in TM_MIB
- SERVERS section in UBBCONFIG or T_SERVER class in TM_MIB

A principal name and decryption key at a particular configuration level can be overridden at a lower level. For example, suppose you configure a principal name and decryption key for machine mach1, and a principal name and decryption key for a server called serv1 running on mach1. The processes on mach1 behave as follows:

- All processes on `mach1` except `serv1` processes use the decryption key assigned to `mach1` to decrypt any received message buffer that is encrypted.
- All `serv1` processes use the decryption key assigned to `serv1` to decrypt any received message buffer that is encrypted.

Configured decryption keys are automatically opened when an ATMI application is booted. The following figure illustrates how the process works.

Figure 2-14 How a Decryption Key Is Initialized Example



The following is a detailed description of how the operation shown in the preceding figure is performed.

1. The administrator defines `SEC_PRINCIPAL_NAME`, `SEC_PRINCIPAL_LOCATION`, and `SEC_PRINCIPAL_PASSVAR` at a particular level in the ATMI application's `UBBCONFIG` file.

2. The administrator loads the configuration by running tmloadcf(1). The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.
3. When prompted, the administrator enters and then re-enters the password for the target principal.
4. The administrator enters the tmboot(1) command to boot the ATMI application.
5. During the boot process, the `map_proof` plug-in reads `SEC_PRINCIPAL_NAME`, `SEC_PRINCIPAL_LOCATION`, and `SEC_PRINCIPAL_PASSVAR`, analyzes their values, and then determines whether the calling process has proven its right to access the requested decryption key. (Having access to a decryption key, or private key, is equivalent to possessing the principal's identity.)
6. If the password associated with `SEC_PRINCIPAL_PASSVAR` matches the assigned password for the principal specified in `SEC_PRINCIPAL_NAME`, the `map_proof` plug-in passes the name, location, and password of the principal to the `PKi_init` plug-in.
7. The `PKi_init` plug-in calls tpkey_open(3c) with the name, location, and password of the principal as arguments. It returns a decryption key handle for the principal.

Each time you invoke `tmloadcf` to load the configuration, you are prompted to enter the password for each of the decryption keys configured with `SEC_PRINCIPAL_PASSVAR`. If you want to avoid having to enter each password manually, you can write a script that automatically enters the passwords. The script must include a definition of each password variable, and it must end with the following line:

```
tmloadcf -y ubbconfig_name < /dev/null
```

No application process has permission to close a decryption key opened during ATMI application booting. The decryption keys stay open until you run the tmshutdown(1)command to shut down the ATMI application.

- Example UBBCONFIG Entries for Principal Names and Decryption Keys

2.15.5.1 Example UBBCONFIG Entries for Principal Names and Decryption Keys

```
*RESOURCES
        SEC_PRINCIPAL_NAME     "Tommy"
        SEC_PRINCIPAL_LOCATION "/home/jhn/secsapp/cert/tommy.pvk"
        SEC_PRINCIPAL_PASSVAR  "TOMMY_VAR"
             .
             .
             .


*SERVERS
        "TMQUEUE"       SRVGRP="QUEGROUP"      SRVID=1
         CLOPT="-s secsdb:TMQUEUE"
         SEC_PRINCIPAL_NAME=    "TOUPPER"
         SEC_PRINCIPAL_LOCATION="/home/jhn/secsapp/cert/TOUPPER.pvk"
         SEC_PRINCIPAL_PASSVAR= "TOUPPER_VAR"
```

2.15.6 Failure Reporting and Auditing

This topic explains how the system manages errors found through digital signatures and message encryption.

- Digital Signature Error Handling
- Encryption Error Handling

2.15.6.1 Digital Signature Error Handling

If message tampering is detected (that is, if the composite signature status is either `TPSIGN_TAMPERED_MESSAGE` or `TPSIGN_TAMPERED_CERT`—see Understanding the Composite Signature Status), the system takes the following actions:

- Generates a userlog(3c) message (severity `ERROR`)
- Discards the buffer as if it were never received by the process

If any individual signature associated with an expired certificate, revoked certificate, expired signature, or postdated signature is detected, the system takes the following actions:

- Generates a userlog(3c) message (severity `WARN`)
- Discards the buffer as if it were never received by the process *unless* the buffer's composite signature status is `TPSIGN_OK` or `TPSIGN_UNKNOWN`

If a process that requires a valid digital signature (based on the `SIGNATURE_REQUIRED=Y` setting) receives a message with the composite signature status `TPSIGN_UNKNOWN`, the system takes the following actions:

- Generates a userlog(3c) message (severity `WARN`)
- Discards the buffer as if it were never received by the process

2.15.6.2 Encryption Error Handling

If a process receives an encrypted message but does not possess an open decryption key matching one of the message's encryption envelopes, the system takes the following actions:

- Generates a userlog(3c) message (severity `ERROR`)
- Discards the buffer as if it were never received by the process

If a process that requires encrypted input (based on the `ENCRYPTION_REQUIRED=Y` setting) receives an unencrypted message, the system takes the following actions:

- Generates a userlog(3c) message (severity `ERROR`)
- Discards the buffer as if it were never received by the process

Note:

- Public Key Security
- Public Key Implementation
- Security Administration Tasks
- Security Interoperability
- Security Compatibility

2.16 Administering Default Authentication and Authorization

Default authentication and authorization work in the same manner that authentication and authorization have worked since they were first made available with the Oracle Tuxedo system.

Default authentication provides three levels of security: no authentication (`NONE`), application password (`APP_PW`), and user-level authentication (`USER_AUTH`). Default authorization provides two levels of security: optional access control list (`ACL`) and mandatory access control list (`MANDATORY_ACL`). Only when users are authenticated to join an ATMI application does the access control list become active.

- Designating a Security Level
- Configuring the Authentication Server

2.16.1 Designating a Security Level

As the administrator, you can use one of three ways to designate a security level for an ATMI application: by editing the `UBBCONFIG` configuration file, by changing the `TM_MIB`.

- Establishing Security by Editing the Configuration File
- Establishing Security by Changing the TM_MIB

2.16.1.1 Establishing Security by Editing the Configuration File

In your `UBBCONFIG` file, set the `SECURITY` parameter to the appropriate value:

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

The default is `NONE`. If `SECURITY` is set to `USER_AUTH`, `ACL`, or `MANDATORY_ACL`, then a system-supplied authentication server named `AUTHSVR` is invoked to perform per-user authentication.

If you select any value other than `NONE`, make sure that the value of the `APPDIR` variable is unique for each ATMI application running on the `MASTER` site. Multiple ATMI applications cannot share the same application directory if security features are being used.

2.16.1.2 Establishing Security by Changing the TM_MIB

To designate a security level through the `TM_MIB`, you must assign a value to the `TA_SECURITY` attribute in the `T_DOMAIN` class. When an ATMI application is inactive, the administrator can `SET`

the value of TA_SECURITY to any of the values that are valid in UBBCONFIG. To complete this task, run the administrative interface tpadmcall(3c).

2.16.2 Configuring the Authentication Server

The Oracle Tuxedo server called AUTHSVR provides a single service, AUTHSVC, which performs authentication. AUTHSVC is advertised by the AUTHSVR server as ..AUTHSVC when the security level is set to ACL or MANDATORY_ACL.

To add AUTHSVC to an ATMI application, you need to define AUTHSVC as the authentication service and AUTHSVR as the authentication server in the UBBCONFIG file. For example:

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


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

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

As another example:

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


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

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

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

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

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

```
.
.

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

Note:

- To use the WebLogic Server as your security database to authenticate Tuxedo users, you must implement single point security administration using `LAUTHSVR` as your authentication server. For information about `LAUTHSVR` and single point security administration with WebLogic Server, refer to Implementing Single Point Security Administration.
- To use the LDAP repository as your security database to authenticate and authorize Tuxedo users, you must implement extensible security administration using `XAUTHSVR` as your authentication and authorization server. For information about `XAUTHSVR` and extensible security administration, refer to `XAUTHSVR(5)` in File Formats, Data Descriptions, MIBs, and System Processes Reference.

See Also:

- How to Enable Application Password Security
- How to Enable User-Level Authentication Security
- Enabling Access Control Security
- Default Authentication and Authorization
- Security Administration Tasks
- Implementing Single Point Security Administration
- AUTHSVR(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

2.17 How to Enable Application Password Security

Default authentication offers an *application password* security level that you invoke by specifying `SECURITY APP_PW` in your configuration file. This level requires that every client provide an application password as part of the process of joining the ATMI application. The administrator defines a single password for the entire ATMI application and gives the password only to authorized users.

To enable the `APP_PW` security level, follow these steps:

1. Ensure that you are working on the ATMI application `MASTER` machine and that the ATMI application is inactive.

2. Set the SECURITY parameter in the RESOURCES section of the UBBCONFIG file to APP_PW.
3. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.
4. The system prompts you for a password. The password you enter may be up to 30 characters long. It becomes the password for the ATMI application and remains in effect until you change it by using the passwd command of tmadmin.
5. Distribute the application password to authorized users of the ATMI application through an offline means such as telephone or letter.

Note:

- Default Authentication and Authorization
- Administering Default Authentication and Authorization
- Security Administration Tasks

2.18 How to Enable User-Level Authentication Security

Default authentication offers an *user-level authentication* security level that you invoke by specifying SECURITY USER_AUTH in your configuration file. This security level requires that in addition to the application password, each client must provide a valid username and user-specific data, such as a password, to join the ATMI application. The per-user password must match the password associated with the combination user-client name stored in a file named tpusr. The checking of per-user password against the password and user-client name in tpusr is carried out by the authentication service AUTHSVC, which is provided by the authentication server AUTHSVR.

To enable the USER_AUTH security level, follow these steps:

- Set up the UBBCONFIG file.
- Set up the user and group files.

Instructions for these steps are provided in the following two topics.

- Setting Up the UBBCONFIG File
- Setting Up the User and Group Files

2.18.1 Setting Up the UBBCONFIG File

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

```
*RESOURCES SECURITY
         USER_AUTH AUTHSVC    AUTHSVC
         .       .
         . *SERVERS AUTHSVR
```

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

CLOPT="-A" causes tmboot(1) to pass only the default command-line options (invoked by "-A") to AUTHSVR when tmboot starts the ATMI application. By default, AUTHSVR uses the client user information in a file named tpusr to authenticate clients that want to join the ATMI application. tpusr resides in the directory referenced by the first pathname defined in the ATMI application's APPDIR variable.

3. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.
4. The system prompts you for a password. The password you enter may be up to 30 characters long. It becomes the password for the ATMI application and remains in effect until you change it by using the passwd command of tmadmin.
5. Distribute the application password to authorized users of the ATMI application through an offline means such as telephone or letter.

2.18.2 Setting Up the User and Group Files

AUTHSVR and the access control checking feature available with the default authorization system require a user file named tpusr, which contains a list of client users allowed to join the ATMI application. tpusr is maintained by the application administrator using the tpusradd(1),tpusrdel(1), and tpusrmod(1) commands. The AUTHSVR server takes as input the client user information stored in the tpusr file; it uses this information to authenticate clients that want to join the ATMI application.

The following display is a sample entry in the tpusr file.

Figure 2-15 tpusr Sample Entry

```
user name      password          user identifier  group identifier  client name
smith:    86V7BzAdwrNVs:       9:          156:       TPCLTNM,*::
```

AUTHSVR and the access control checking feature also require a group file named tpgrp, which contains a list of groups associated with the client users allowed to join the ATMI application; tpgrp is maintained by the application administrator using the tpusradd(1), tpgrpdel(1), and tpgrpmod(1) commands.

AUTHSVC assigns an authenticated client user an *application key*, which contains a user identifier and associated group identifier for the USER_AUTH, ACL, or MANDATORY_ACL security level. (See Application Key for more information about application keys.)

The following display is a sample entry in the tpgrp file.

Figure 2-16 tpgrp Sample Entry

```
group name      group identifier
Administrators::      156:
```

As the administrator, you must define lists of users and groups in the `tpusr` and `tpgrp` files, both of which are located in the directory referenced by the first path name defined in the ATMI application's `APPDIR` variable. The files are colon-delimited, flat text files, readable and writable only by the application's administrator.

- Converting System Security Data Files to Oracle Tuxedo User and Group Files
- Adding, Modifying, or Deleting Users and Groups

2.18.2.1 Converting System Security Data Files to Oracle Tuxedo User and Group Files

You may already have files containing lists of users and groups on your host system. You can use them as the user and group files for your ATMI application, but only after converting them to the format required by the Oracle Tuxedo system. To convert your files, run the tpaclcvt(1) command, as shown in the following sample procedure. The sample procedure is written for a UNIX host machine

1. Ensure that you are working on the ATMI application `MASTER` machine and that the ATMI application is inactive.
2. To convert the `/etc/password` file into the format needed by the Oracle Tuxedo system, enter the following command.
   ```
   tpaclcvt -u /etc/password
   ```
   This command creates the `tpusr` file and stores the converted data in it. If the `tpusr` file already exists, `tpaclcvt` adds the converted data to the file, but it does *not* add duplicate user information to the file.

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

3. To convert the `/etc/group` file into the format needed by the Oracle Tuxedo system, enter the following command.
   `tpaclcvt -g /etc/group` This command creates the `tpgrp` file and stores the converted data in it. If the `tpgrp` file already exists, `tpaclcvt` adds the converted data to the file, but it does *not* add duplicate group information to the file.

2.18.2.2 Adding, Modifying, or Deleting Users and Groups

The Oracle Tuxedo system requires that you maintain a list of your application users in a file named `tpusr`, and a list of groups, in a file named `tpgrp`. There are two methods of modifying the entries in these files: by issuing commands or by changing the values of the appropriate attributes in the `ACL_MIB`.

- Changing Entries for Users and Groups Through Commands
- Changing Entries for Users and Groups Through the ACL_MIB

2.18.2.2.1 Changing Entries for Users and Groups Through Commands

You can add, modify, or delete entries in the `tpusr` and `tpgrp` files at any time by running one of the following commands.

Run . . .	To . . .	An Entry in This File
tpusradd(1)	Add	tpusr
tpusrmod(1)	Modify	
tpusrdel(1)	Delete	
tpgrpadd(1)	Add	tpgrp
tpgrpmod(1)	Modify	
tpgrpdel(1)	Delete	

To run any of these commands, follow these steps:

1. For an inactive ATMI application, make sure you are working from the application MASTER machine. For an active ATMI application, you may work from any machine in the configuration.
2. For specific instructions on running a command, see the entry for that command in *Oracle Tuxedo Command Reference*.

2.18.2.2.2 Changing Entries for Users and Groups Through the ACL_MIB

If you prefer not to use the command-line interface, you can add, modify, or delete user entries in tpusr by changing the appropriate attribute values in the T_ACLPRINCIPAL class in the ACL_MIB(5) . This method is more efficient than the command-line interface if you want to add several user entries simultaneously, since tpusradd(1) allows you to add only one user at a time.

Similarly, you can add, modify, or delete group entries in tpgrp by changing the appropriate attribute values in the T_ACLGROUP class in the ACL_MIB(5). This method is more efficient than the command-line interface if you want to add several group entries simultaneously, since tpgrpadd(1) allows you to add only one group at a time.

> **Note:**
>
> - Default Authentication and Authorization
> - Administering Default Authentication and Authorization
> - Security Administration Tasks

2.19 Enabling Access Control Security

Default authorization consists of an access control checking feature that determines which users can execute a service, post an event, or enqueue (or dequeue) a message on an application queue. There are two levels of access control security: optional access control list (ACL) and mandatory access control list (MANDATORY_ACL). Only when users are authenticated to join an ATMI application does the access control list become active.

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

- System administration is simplified. It is easier to give a group of people access to a new service than it is to give individual users access to the service.

- Performance is improved. Because access permission needs to be checked for each invocation of an entity, permission should be resolved quickly. Because there are fewer groups than users, it is quicker to search through a list of privileged groups than it is to search through a list of privileged users.

The access control checking feature is based on three files that are created and maintained by the application administrator:

- `tpusr` contains a list of users
- `tpgrp` contains a list of groups
- `tpacl` contains a list of mappings of groups to application entities (such as services) known as the access control list (ACL)

By parsing the client's *application key*, which contains information identifying the client as a valid user and valid group member, an entity (such as a service, event, or application queue) can identify the group to which the user belongs; by checking the `tpacl` file, an entity can determine whether the client's group has access permission.

The application administrator, application operator, and processes or service requests running with the privileges of the application administrator/operator are *not* subject to ACL permission checking.

If user-level ACL entries are needed, they may be implemented by creating a group for each user, and then mapping the group to the appropriate application entities in the `tpacl` file.

- How to Enable Optional ACL Security
- How to Enable Mandatory ACL Security
- How to Enable Generic LDAP Based Security
- How to Enable Security Service for OES

2.19.1 How to Enable Optional ACL Security

Default authentication offers an *optional ACL* (`ACL`) security level that you invoke by specifying `SECURITY ACL` in your configuration file. This security level requires that each client provide an application password, a username, and user-specific data, such as a password, to join the ATMI application. If there is no entry in the tpacl file associated with the target application entity, the user is permitted to access the entity.

This security level enables an administrator to configure access for only those resources that need more security. That is, there is no need to add entries to the `tpacl` file for services, events, or application queues that are open to everyone. Of course, if there *is* an entry in the `tpacl` file associated with the target application entity and a user attempts to access that entity, the user *must* be a member of a group that is allowed to access that entity; otherwise, permission is denied.

To enable the `ACL` security level, follow these steps:

1. Set up the `UBBCONFIG` file.
2. Set up the `ACL` file.

Instructions for these steps are provided in the following two topics.

- Setting Up the UBBCONFIG File
- Setting Up the ACL File

2.19.1.1 Setting Up the UBBCONFIG File

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

```
*RESOURCES
          SECURITY   ACL
          AUTHSVC
          ..AUTHSVC
          .
          .
          .
          *SERVERS
          AUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600
          MAXGEN=2 CLOPT="-A"
```

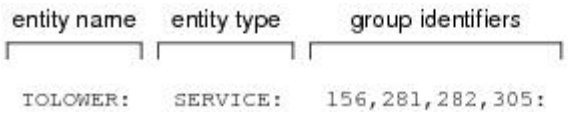
   CLOPT="-A" causes tmboot(1) to pass only the default command-line options (invoked by "-A") to AUTHSVR when tmboot starts the ATMI application. By default, AUTHSVR uses the client user information in a file named tpusr to authenticate clients that want to join the ATMI application. tpusr resides in the directory referenced by the first pathname defined in the ATMI application's APPDIR variable.
3. Load the configuration by running tmloadcf(1). The tmloadcf command parses UBBCONFIG and loads the binary TUXCONFIG file to the location referenced by the TUXCONFIG variable.
4. Distribute the application password to authorized users of the ATMI application through an offline means such as telephone or letter.

2.19.1.2 Setting Up the ACL File

The access control checking feature requires a user file named tpusr, a group file named tpgrp, and an ACL file named tpacl. The ACL file contains mappings of groups to application entities. An entity may be a service, event, or application queue.

The following display is a sample entry in the tpacl file.

Figure 2-17 tpacl Sample Entry



As the administrator, you must define the entries in the tpacl file, which is located in the directory referenced by the first pathname defined in the ATMI application's APPDIR variable. The file is a colon-delimited, flat text file, readable and writable only by the application's administrator.

There are two methods of modifying the ACL entries in the tpacl file: by issuing commands or by changing the values of the appropriate attributes in the ACL_MIB.

- Changing ACL Entries Through Commands
- Changing ACL Entries Through the ACL_MIB

2.19.1.2.1 Changing ACL Entries Through Commands

You can add, modify, or delete ACL entries in the `tpacl` file at any time by running one of the following commands.

Run . . .	To . . .
`tpacladd(1)`	Add an entry
`tpaclmod(1)`	Modify an entry
`tpacldel(1)`	Delete an entry

To run any of these commands, follow these steps:

1. For an inactive ATMI application, make sure you are working from the application `MASTER` machine. For an active ATMI application, you may work from any machine in the configuration.
2. For specific instructions on running a command, see the entry for that command in *Oracle Tuxedo Command Reference*.

2.19.1.2.2 Changing ACL Entries Through the ACL_MIB

If you prefer not to use the command-line interface, you can add, modify, or delete ACL entries in `tpacl` by changing the appropriate attribute values in the `T_ACLPERM` class in the ACL_MIB(5) . This method is more efficient than the command-line interface if you want to add several ACL entries simultaneously, since tpacladd(1) allows you to add only one ACL entry at a time.

2.19.2 How to Enable Mandatory ACL Security

Default authentication offers a *mandatory ACL* security level that you invoke by specifying `SECURITY MANDATORY_ACL` in your configuration file. This security level requires that each client provide an application password, a username, and user-specific data, such as a password, to join the ATMI application. If there is no entry in the `tpacl` file associated with the target application entity, the client is *not* permitted to access the entity. In other words, an entry *must* exist in the `tpacl` file for every application entity that a client needs to access. For this reason, this level is called *mandatory*.

Of course, if there *is* an entry in the `tpacl` file associated with the target application entity and a user attempts to access that entity, the user *must* be a member of a group that is allowed to access that entity; otherwise, permission is denied.

To enable the `MANDATORY_ACL` security level, follow these steps:

1. Set up the `UBBCONFIG` file.
2. Set up the ACL file.

Instructions for these steps are provided in the following two topics.

- Setting Up the UBBCONFIG File
- Setting Up the ACL File

2.19.2.1 Setting Up the UBBCONFIG File

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

```
*RESOURCES SECURITY   MANDATORY_ACL AUTHSVC
                  ..AUTHSVC      .
                    .
                    .
               *SERVERS AUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y
GRACE=600
               MAXGEN=2 CLOPT="-A"
```

   `CLOPT="-A"` causes tmboot(1) to pass only the default command-line options (invoked by "-A") to `AUTHSVR` when `tmboot` starts the ATMI application. By default, `AUTHSVR` uses the client user information in a file named `tpusr` to authenticate clients that want to join the ATMI application. `tpusr` resides in the directory referenced by the first pathname defined in the ATMI application's `APPDIR` variable.
3. Load the configuration by running tmloadcf(1). The `tmloadcf` command parses `UBBCONFIG` and loads the binary `TUXCONFIG` file to the location referenced by the `TUXCONFIG` variable.
4. Distribute the application password to authorized users of the ATMI application through an offline means such as telephone or letter.

2.19.2.2 Setting Up the ACL File

> **See Also:**
>
> Setting Up the ACL File.

> **Note:**
>
> - Default Authentication and Authorization
> - Administering Default Authentication and Authorization
> - Security Administration Tasks

2.19.3 How to Enable Generic LDAP Based Security

Generic LDAP based security includes the user-level authentication and access control security.

With this security mechanism, authentication and authorization are performed by invoking `TUXEDO "..ATNSVC"` and `"..ATZSVC"` administrative services. It provides flexibility for Oracle Tuxedo user to store their security information in independent repository and access these

security information from the "..ATNSVC" and "..ATZSVC" services. Oracle Tuxedo supplies a default implementation of XAUTHSVR server which advertises these two administrative services. With this implementation, the security information, including Tuxedo user ID, password, and service access privilege, are stored in LDAP repositories.

Each client must provide a valid user name and user-specific password, to join the ATMI application. The user password must match the password stored in LDAP repositories. Each client must be granted with proper privilege before it can access Tuxedo services successfully.

To enable the LDAP based security with default XAUTHSVR implementation, follow these steps:

1. Setting Up the UBBCONFIG File
2. Setting Up the XAUTHSVR Server Configuration File
3. Setting Up the LDAP Repository
4. Setting Up the Authorization Cache

Instructions for these steps are provided in the following topics.

- Setting Up the UBBCONFIG File
- Setting Up the XAUTHSVR Server Configuration File
- Setting Up the LDAP Repository
- Setting Up the Authorization Cache

2.19.3.1 Setting Up the UBBCONFIG File

1. Open UBBCONFIG with a text editor.
2. In the RESOURCES section, do the following:
   a. Set the SECURITY parameter to one of these values: USER_AUTH, ACL or MANDATORY_ACL.
   b. Set the OPTIONS parameter to EXT_AA.
   c. Do one of the following:
      - If the SECURITY parameter is set to ACL or MANDATORY_ACLAUTHSVC, set AUTHSVC to ..AUTHSVC, which is the service name advertised by the XAUTHSVR server.
      - If the SECURITY parameter is set to USER_AUTH, set AUTHSVC to AUTHSVC, which is the service name advertised by the XAUTHSVR server.
3. Set up XAUTHSVR in the SERVERS section.

```
* RESOURCES
            SECURITY
            ACL AUTHSVC ..AUTHSVC OPTIONS
                EXT_AA
                *SERVERS
                XAUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y
```

2.19.3.2 Setting Up the XAUTHSVR Server Configuration File

XAUTHSVR server configuration file is used for XAUTHSVR to locate the LDAP repository. By default, the configuration file named tpldap.xauth resides in $TUXDIR/udataobj directory. You

can specify a customized location with "-f" option to XAUTHSVR server. XAUTHSVR server allows you to store your authentication and authorization information in separate LDAP repositories. You can specify an ATN configuration file with "-n" option and "-z" option respectively. All these configuration files share the same format.

The following table describes the XAUTHSVR configuration file keywords.

Table 2-3 XAUTHSVR Configuration File Keywords

Keyword	Value Type	Usage
FILE_VERSION	numeric	The configuration file version. The default is 1. This should remains in 1.
LDAP_VERSION	numeric	The LDAP protocol version. Valid values are 2 and 3. The default is 3.
BINDDN	string	The DN used to bind to an LDAP server. Usually the DN represents a LDAP administrator. The default is " cn=Admin". The tpldapconf command can be used to create BINDDN.
BASE	string	LDAP search base. The default is " ou=people,ou=aa,dc=mydomain", where mydomain is the root node of the authentication or authorization security repository.
PASSWORD	string	The password for bind DN. This is a required keyword and the password is encrypted in clear text. The tpldapconf command can be used to create the encrypted password.
LDAP_ADDR	string	A comma-separated list of LDAP address containing hostnames and ports. The syntax is [//]hostname[:port]. The default value for port is 7001. If LDAP_ADDR is not specified, XAUTHSVR regards localhost:7001 as the location to contact the LDAP server.
UID_KW	string	The keyword used in user unique identification search in the authentication security repository. The default value is " uid".
PWD_KW	string	The keyword used in user password search in the authentication security repository. The default value is "userPassword".
MEMBEROF_KW	string	The keyword used in group membership search. The default value is "memberof". Different LDAP servers use different key name to identify the user's group membership. When using OVD with virtual member plugin enabled, the keyword is "memberof".

2.19.3.3 Setting Up the LDAP Repository

The security information in the LDAP repository follows below schema:

- **inetOrgPerson**: This object class holds the entries that represent people. The definition follows RFC 4519 & 2798 standard except that the attribute "uid" length is limited to up to 30 characters. Each Oracle Tuxedo user is saved as an entry in this class. The information including user identification and user password is used for user-level authentication.
- **groupOfUniqueNames**: This object classes holds the entries that represents a set of named objects including the information relevant to purpose or maintenance of the set. The definition follows RFC 4519 standard. This class groups a list of users that can be granted with certain sort of permissions. Groups can be nested. The permission granted to a parent group also applies to its child groups.
- **Orcljaznpermission**: This object class holds the tuxedo permission object consisting of the attributes shown in the following table. This object consists of two parts. One is the permission, which describes the resource types, target resource, and actions on the target resource. The other is the assigned groups or users, which are granted with this permission.

The following table describes the orcljaznpermission class attributes.

Table 2-4 orcljaznpermission Class Attributes

Attribute	Type	Constraints	Description
Cn	String:	Single-valued, unique, required	Permission name
Displayname	String:	String:	Display name
Description	String:	String:	Description
OrclJpsResourceTypeName	String:	Single valued	Type name of the resource to be protected. To define a Tuxedo service, this attribute should be specified to "SERVICE".
Orcljaznpermissiontarget	String:	Single valued	Name of the resource to be protected. To define a Tuxedo service, this attribute should be specified as the service name.
Orcljaznpermissionactions	String:	Single valued	List of the assigned actions, separated by comma. To define a Tuxedo service, this attribute should be specified to "EXECUTE".

2.19.3.4 Setting Up the Authorization Cache

In order to improve ATZ performance, the new ATZ mechanism introduces a roll-up cache, in which privileges of specific user identifiers are stored, to every Tuxedo server. To meet various ATZ requirements, the cache is configurable and flexible at each Tuxedo server level.

Three environment variables control the basic behaviors of the cache. After defining an ENVFILE parameter for a specific server entry in TUXCONFIG, these environment variables can be defined for each Tuxedo server entry in the SERVERS section in UBBCONFIG.

TMATZPRIVILEGEMAX
It specifies the maximum number of privileges entries. When the privileges number in the cache reaches this threshold, the new entry replaces an old one. Remaining time-to-live of privileges is evaluated for Tuxedo to choose the " most useless" entry in the ATZ cache. If this

environment variable is set to 0, ATZ cache in Tuxedo server is disabled and all ATZ requests are dispatched to ATZ service. If this environment variable is not defined explicitly, the default value is 100. The valid value range is from 0 to 32767. The size of one privilege entry in the ATZ cache is 50 bytes or so.

TMATZRESOURCEMAX
It specifies the maximum number of resource entries which can be allocated for a specific Tuxedo Server. When the resource number in the cache reaches this threshold, both the resource and new privilege are not added into the cache and the subsequent access requests to the resource are routed to the ATZ server until an available resource slot is found. Tuxedo keeps a reference number to each resource entry occupied by the cached privileges. When no privilege occupies the specific resource entry, it will be cleared from the cache.

If this environment variable is not defined explicitly, the value is set to the current number of advertised services. Meanwhile, the value of `TMATZPRIVILEGEMAX` must be set bigger than or equal to the value of `TMATZRESOURCEMAX`, otherwise `TMATZPRIVILEGEMAX` will be set to the equal value of `TMATZRESOURCEMAX`.The valid value range is from 0 to 32767. The size of one resource entry in the ATZ cache is 148 bytes or so. If this environment variable is set to 0, the ATZ cache in Tuxedo server is disabled and all ATZ requests are dispatched to ATZ service.

TMATZEXP
It specifies the maximum lifetime of a specific privilege in minutes. When the lifetime of a privilege reaches this threshold, the privilege is removed from the cache. If this environment variable of a Tuxedo server is set to 0, all privileges stored in the Tuxedo server have infinite lifetime and never expire. If this environment variable is not defined explicitly, the default value is 10. The valid value range is from 0 to 525600. 525600 indicates the privilege life in cache is 1 year.

The following sample demonstrates how to calculate the total memory size occupied by an ATZ cache in a specific Tuxedo server. Suppose there is a server accessing 10 /Q message queues, which correspond to 10 resource entries, and there are 100 potential users invoking services of this server, so we assume `TMATZRESOURCE` value is 10 and `MAXTMATZPRIVILEGEMAX` value is 1000.

According to the occupied memory size formula: [Max resource entry] * [Resource entry size] + [Max privilege entry]*[privilege entry size] the result of above case is:

(10*148 + 1000*50)= 51480 (51 KB)

> **Note:**
>
> - Default Authentication and Authorization
> - Administering Default Authentication and Authorization

2.19.4 How to Enable Security Service for OES

1. Install OES server and client (security module).

> **Note:**
>
> Oracle Entitlement Server (OES) client 11.1.2.0 is certified for use.

2. Configure OES java client to connect OES server.
3. Create an application in OES server; create resource type, resource, and policy to specify authorization. Note: Users are allowed to authorize different types of resources having the same name by defining different policies on OES side, each of which authorizes only one resource type. For example, in order to authorize a service named OES and a /Q queue named OES, users can define two policies on OES side to authorize them respectively.
4. Configure authorization template file (configure the application name in the configured OES server and the full path name of `jps-config.xml` to be precise) to indicate what you have configured in OES.
5. Configure `EAUTHSVR` server:
   - Run `tux.env` to set up `libjvm.so` in your library path.
   - Set `oes-client.jar` in `CLASSPATH`.
6. Configure authentication server:

Configure `EXT_AA` in `OPTIONS` and `..AUTHSVC` in `UBBCONFIG RESOURCES` to authenticate service you use.

For more information, please refer to *Installation Guide for Oracle Identity and Access Management*.

2.20 Using the Kerberos Authentication Plug-in

Kerberos is a network authentication protocol. It is designed to provide strong authentication for client/server applications by using secret-key cryptography. The Kerberos authentication protocol provides a mechanism for mutual authentication between a client and a server, or between one server and another, before opening a network connection between them. The protocol assumes that initial transactions between clients and servers take place on an open network where most computers are not physically secure. It also assumes that packets traveling along the network can be monitored and modified at will.

After using Kerberos to prove the identity of a client and server, their communications can be encrypted to ensure privacy and data integrity. Refer to the following *See Also* section for more information about Kerberos.

The following sections describe the Kerberos authentication plug-in feature included in Tuxedo:

- Kerberos Plug-In
- Kerberos Plug-In Pre-configuration
- Kerberos Plug-In Configuration

2.21 Kerberos Plug-In

Tuxedo provides a general security framework that can be customized. This framework is further enhanced with the inclusion of a Kerberos plug-in.

- Kerberos Supported Platforms
- Kerberos Plug-in Features

2.21.1 Kerberos Supported Platforms

Currently the Kerberos plug-in supports the following platforms:

- Microsoft Kerberos bundled with Windows 2000/2003 server
- Kerberos V systems on HP-UX(PA-RISC) provided by HP
- Kerberos V systems on Solaris 9 (SPARC) provided by Sun Microsystems

2.21.2 Kerberos Plug-in Features

The Kerberos Plug-in is a dynamic library that must be registered into the Tuxedo system, and a Kerberos authentication server (KAUTHSVR(5)). The Tuxedo implementation of the Kerberos plug-in supports the following:

- Authentication between Tuxedo native client and server
- Full support of Tuxedo ACL security mechanism

> **Note:**
>
> Authentication between the security protocols of Tuxedo workstation client and workstation handler, authentication between two domain gateways and CORBA components are not supported.

2.22 Kerberos Plug-In Pre-configuration

To use Kerberos authentication, you must make sure the following system requirements are set up properly:

- Supported systems run well with the correct Kerberos settings
- User/service accounts are set correctly
- The Kerberos authentication server key table is created correctly on UNIX
- Kerberos interoperability between UNIX and Windows is set correctly and verified if a heterogeneous (UNIX/Windows mixed) environment is needed.

2.23 Kerberos Plug-In Configuration

This section provides configuration information to get the Kerberos plug-in set up and running.

1. Configure the Kerberos Plug-in
2. Configure KAUTHSVR
3. Configure Tuxedo Native Client

Each of these steps are explained in more detail in the subsections that follow.

- Configure the Kerberos Plug-in
- Configure KAUTHSVR
- Configure Tuxedo Native Client
- Limitations

2.23.1 Configure the Kerberos Plug-in

You must first register the Kerberos plug-in on UNIX and Windows platforms.

The Kerberos plug-in must be configured using the EPIF commands `epifreg` and `epifregedt`. These commands will automatically add the plug-in to the Tuxedo registry in UNIX and Windows. For example:

Listing UNIX Registration

```
epifreg -r -p krb5/atn -i engine/security/authentication -o SYSTEM -v 1.0 \
         -f $TUXDIR/lib/libkrb5atn.so \
                -e krb5_plugin_entry \
                -u KRB5_CONFIG=/etc/krb5.conf \
                -u KRB5_KDC=/etc/krb5.kdc\
                -u KAUTHSVRPRINC="krbauth@host.yourcomany.com"
          epifregedt -s -k SYSTEM/interfaces/engine/security/authentication \
                -a Selector=native/security/authentication \
                -a native/security/authentication=krb5/atn
```

Listing Windows Registration

```
epifreg -r -p krb5/atn -i engine/security/authentication -o SYSTEM -v 1.0
\
                -f %TUXDIR%\bin\libkrb5atn.dll \
                -e krb5_plugin_entry \
                -u KAUTHSVRPRINC="krbauth/host.yourcomany.com@REALM"
        epifregedt -s -k SYSTEM/interfaces/engine/security/authentication \
                -a Selector=native/security/authentication \
                -a native/security/authentication=krb5/atn
```

> **Note:**
>
> On a Windows platform, the plug-in `KRB5_CONFIG` and `KRB5_KDC` parameters are not required. These parameters are used on a UNIX platform to locate the Kerberos-related configuration files. `KAUTHSVRPRINC` specifies the principal name for the `KAUTHSVR` server and Tuxedo clients use it as the server principal name.

On UNIX platforms, the GSS format is used. Because Microsoft does not support standard GSS name representation, the `KAUTHSVRPRINC` parameter must be given a complete Kerberos realm name.

The name format is illustrated as follows:

- A UNIX Tuxedo client must use GSS format to access `KAUTHSVR`.
- A Windows Tuxedo client always uses the *complete Kerberos realm name* to access `KAUTHSVR`.
  `KAUTHSVRPRINC` can also be set as an environment variable.
- Restore Default Plug-in

2.23.1.1 Restore Default Plug-in

The following commands restore the plug-in to its default state.

Listing 2-4 Restore Default Plug-In Settings

```
epifreg -r -p bea/native/atn
                \-i engine/security/authentication \-v 1.0
                 -f libtux.so -e_ep_dl_atnlcl
        epifregedt -s -k SYSTEM/interfaces/engine/security/authentication \
                 -a Selector=native/security/authentication \
                 -a native/security/authentication=bea/native/atn
```

> **Note:**
>
> In the above listing, `libtux.so` is used as an example. You must use the file name `libtux` plus your *platform specific* dynamic library extension.

2.23.2 Configure KAUTHSVR

`KAUTHSVR` is a Tuxedo server located in `TUXDIR/bin` directory and must be manually configured in the `UBBCONFIG` file. `KAUTHSVR` authenticates client identity by validating the client security token. It addresses the Tuxedo ACL mechanism when the security level is set above `"USER_AUTH"` in the `UBBCONFIG` file.

The following are examples of how `KAUTHSVR` is configured in the `UBBCONFIG` file for both UNIX and Windows:

Listing UNIX UBBCONFIG KAUTHSVR Configuration

```
*RESOURCES
         IPCKEY 66666
         MASTER SITE1
         MODEL MP

        SECURITY MANDATORY_ACL

        *SERVERS
         KAUTHSVR SRVGRP=SECGRP SRVID=100 GRACE=0 MAXGEN=2 CLOPT="-A -- -
k /etc/krbauth.kt -p krbauth@host.yourcomany.com"
```

> **Note:**
>
> The `-k` option allows you to provide the `KAUTHSVR` Kerberos key table file location.

The `-p` option indicates `KAUTHSVR` principal name.

`KAUTHSVR` running on UNIX platforms must use the GSS format.

Listing Windows UBBCONFIG KAUTHSVR Configuration

```
*RESOURCES
          IPCKEY 66666
          MASTER SITE1
          MODEL MP
```

```
          SECURITY MANDATORY_ACL

          *SERVERS

           KAUTHSVR SRVGRP=GROUP3 SRVID=100 GRACE=0 MAXGEN=2
           SEC_PRINCIPAL_NAME="kauthsvc" SEC_PRINCIPAL_PASSVAR=test CLOPT="-A
-- -p
           krbauth/host.yourcomany.com@REALM"
```

> **Note:**
>
> The `-p` option indicates `KAUTHSVR` principal name.

Instead of using the `-k` option, Windows platforms must use the following two arguments:

Instead of using the `-k` option, Windows platforms must use the following two arguments:

- `SEC_PRINCIPAL_NAME` represents `KAUTHSVR`, it does not represent the server principal name (which is represented by the -p option).
- `SEC_PRINCIPAL_PASSVAR` is the internal password variable. It is not the *true* password that is required when `tmloadcf` creates the `TUXCONFIG` file. The `tmloadcf` password input must be same as the `KAUTHSVR` account password in a Windows domain.

`KAUTHSVR` running on Windows platform must use the *complete Kerberos realm name*.

2.23.3 Configure Tuxedo Native Client

To use the Tuxedo native client with Kerberos enabled, you must first obtain a valid `TGT` from the `KDC` using `kinit` or other similar commands.

No programming APIs are required. Also, if `USER_AUTH` is specified, the Tuxedo user name is not required in the `tpusr` file. However, a user name is required for `ACL` and `MANDATORY_ACL` security level.

2.23.4 Limitations

- Kerberos Plug-In only works on systems where the plug-in is installed and registered to Tuxedo through `epif*` commands. If the Tuxedo administrator does not register the `libkrb5atn` to Tuxedo, the default plug-in still works and the default Tuxedo security mechanism takes effect. `KAUTHSVR` supports full function of `AUTHSVR` in addition to Kerberos authentication.
- Even if the Kerberos plug-in is configured on a system running WSH, the workstation clients connected to this system use the Tuxedo default security mechanism. This is because the protocol between workstation client and WSH is not affected using this feature.
- Although CORBA native clients can take advantage of Kerberos support, we do *not* support CORBA remote clients using Kerberos. ISH will report an error when the Kerberos plug-in is installed.

> **Note:**
>
> Authentication between the security protocols of Tuxedo workstation client and workstation handler, authentication between two domain gateways and CORBA components are unsupported.

> **Note:**
>
> - KAUTHSVR(5)
> - Kerberos Introduction from MIT (tap://web.mit.edu/kerberos/wow/)
> - Microsoft White Papers and Guide for Kerberos (tap://www.microsoft.com/windows2000/technologies/security/kerberos/default.asp)
> - RFC 1510, Kerberos protocol (tap://www.ietf.org/raft/rfc1510.txt)
> - RFC 2743, GSSAPI (tap://www.ietf.org/raft/rfc2743.txt)
> - RFC 1509, GSSAPI, c-bindings.(http://www.ietf.org/raft/rfc1509.txt)

2.24 Using the Cert-C PKI Encryption Plug-in

The Cert-C based PKI (public key infrastructure) plug-in utilizes the public key encryption algorithm to provide you with the ability to:

- sign - assign a signature to a Tuxedo typed buffer
- seal - encrypt a Tuxedo typed buffer, and
- envelope - provide access to the user signature and encryption information associated with the Tuxedo typed buffer

The following sections describe the Cert-C PKI encryption feature included in Tuxedo:

- Cert-C PKI Encryption Plug-In
- Cert-C PKI Encryption Plug-In Pre-configuration
- Cert-C PKI Encryption Plug-In Configuration

2.25 Cert-C PKI Encryption Plug-In

The Tuxedo Cert-C PKI encryption plug-in uses LDAP version 2 or higher as the storage mechanism for the publicly accessible user certificates. LDAP is a commonly used and deployed network directory service.

2.26 Cert-C PKI Encryption Plug-In Pre-configuration

To use the Tuxedo Cert-C PKI encryption plug-in, you must ensure to follow the system requirements:

- Access to a configured LDAP server
- User certificates stored in the LDAP are entered in the following format: `cn=user name`

2.27 Cert-C PKI Encryption Plug-In Configuration

To use this plug-in, you must run a command script to configure Tuxedo in order to use this plug-in as the default PKI plug-in.

The Tuxedo Cert-C plug-in utilizes four interface groups in the Tuxedo Security PIF and is configured using PIF registry commands. The required interface groups are:

- Configure Certificate Lookup
- Configure Key Management
- Configure Certificate Parsing
- Configure Certificate Validation

In the Tuxedo environment, only user names are available in the plug-in at runtime. In order to get the proper search information, it assumes that a certificate stored in the LDAP with a `cn=user name` entry is a Tuxedo user name.

2.27.1 Configure Certificate Lookup

This interface group expects a user certificate to be located on an LDAP server and it has access permission to read these certificates. The certificate lookup interface has four parameters that must be configured. The parameters are described as follows:

`ldapUserCertificate`
LDAP server configuration parameter that identifies where the plug-in can obtain user certificates. The network address for the LDAP host is specified in this parameter as a string variable. It also contains the TCP LDAP port number. The syntax of this parameter is `LDAP:URL`. For example: `ldapUserCertificate=ldap://sagamore:389`

This example tells the Cert-C plug-in that the LDAP server is located on a machine called “sagamore”, and it is listening on port 389.

`ldapBaseObject`
LDAP server configuration parameter that identifies the base DN where the LDAP search should start. For example: `ldapBaseObject="ou=Engineer Test,o=ABC Company,c=US"` This example initiates a search from the directory information tree `"ou=Engineer Test, o=ABC Company, c=US"`

`ldapFilterAttribute`
LDAP server configuration parameter that identifies the search filter used in an LDAP search when retrieving a certificate by subject name. This parameter is a string variable and follows the same syntax as `ldapBaseDNAttribute`. For example: `ldapFilterAttribute="cn"`

This example tells the Cert-C plug-in to use `"cn"` as a filter.

ldapBaseDNAttribute
LDAP server configuration parameter that is used in an LDAP search to build the base DN. This parameter is a string variable consisting of a comma-separated list of DN attributes, such as c, o. An optional blank space can follow the commas. For example:
`ldapBaseDNAttribute="c, o, ou, cn"`

This example tells the Cert-C plug-in to use the `"c"`, `"o"`, `"ou"`, `"cn"` attributes when constructing the DN for a search.

- OpenLDAP for X.509 Certificate Lookup

2.27.1.1 OpenLDAP for X.509 Certificate Lookup

To enable OpenLDAP for X.509 certificate lookup, execute the command shown in the following Listing modifies Tuxedo PKI plug-in information:

Listing OpenLDAP Command

```
epifreg -r -p security/BEA/certificate_lookup -i engine/security/
certificate_lookup -v 1.0 -f 'libplugin.<suffix>' -e _ep_dl_certlookup -u
        userCertificateLdap=ldap:/<ldap_host_name>:<ldap_port>/-u
ldapBaseObject='<your_ldap_base>' -u binaryCertificate='YES'.
```

Where:

- `<suffix>` is the proper suffix for the shared library (for example, `libplugin.dll` for Windows, and `libplugin.so.71` for Solaris).
- `ldap_host_name` is the name of the host where the LDAP server is running
- `ldap_port` is the LDAP server port number (for example, `userCertificateLdap=ldap:/cerebrum:389/)`.
- `your_ldap_base` is the base of your LDAP DIT (for example,

```
ldapBaseObject='ou=Accounting,o=ABC
        Company,c=US')
```

> **Note:**
>
> You may also need to modify the `bea_ldap_filter.dat` file which is located in `$TUXDIR/udataobj/security`.

Listing displays a filter example.

Listing Filter Example

```
"BEA_person_lookup"
        ".*' " " "(&(objectClass=inetOrgPerson)(cn=%v))" "username"
                "(&(objectClass=inetOrgPerson)(cn=%v*))" "start of
            username"

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

```
on
        sn, cn"
```

2.27.2 Configure Key Management

The location of the private key is the only configuration parameter that must be specified for key management interface.

- decPassword
- privateKeyDir

2.27.2.1 decPassword

Optional parameter. It is a string variable that gives the Cert-C PKI encryption plug-in the password to decrypt the private key wrapped in encrypted private key information format. For example:

```
decPassword="abc123"
```

The plug-in assumes the private key information file follows the `"<subject_name>.epk"` naming scheme.

> **Note:**
>
> `decPassword` and `privateKeyDir` can be overridden by using the tpkey_open(3c) `identity_proof` and `location` parameters.

2.27.2.2 privateKeyDir

A string variable parameter in file URL format. It indicates the default location of the private key. For example:

```
privateKeyDir=file:///c:\home\certs\
```

This example tells the Cert-C PKI encryption plug-in to look for a private key in the `c:\home\certs` directory. The private key can be a binary file that conforms with PKCS #8. It must have a `.pvt` or `.epk` extension.

If the password is given in the `"decPassword"` path or `tpkey_open(..., identity_proof, ...)`, then the `.epk` file will be searched first, if not found then it will try `.pvt` file. If the password is not given in the `"decPassword"` path or `tpkey_open(..., identity_proof, ...)`, then only `.pvt` file is searched.

2.27.3 Configure Certificate Parsing

No special configuration parameter is needed to utilize the certificate parsing interface. It is initialized automatically.

> **Note:**
>
> Certificates must be X.509-compatible in DER format.

2.27.4 Configure Certificate Validation

This interface group allows the Cert-C PKI encryption plug-in to examine a certificate and to determine its validity based on trusted certificate authorities, chains of trust, certificate revocation list. There are two configuration parameters associated wither certificate validation:

- caCertificateFile
- crlFile

2.27.4.1 caCertificateFile

A string variable configuration parameter in file `URL` format. It points to a single certificate whose public key is trusted by the user. The certificate can be self-signed. If the certificate chain validates this trusted certificate the certificate is deemed a “good” certificate. For example:

> **Note:**
>
> There is only one certificate validation chain level. That is, all user certificates are issued directly by the root `CA` configured in `caCertificateFile`.

```
caCertificateFile=file:///c:\home\certs\root.cer
```

This example indicates that the trusted root certificate is located at directory called `c:\home\certs` and is named `root.cer`.

2.27.4.2 crlFile

A string variable configuration parameter in file `URL` format. It points to a single `CRL` that is to be used to verify the resulting certificate path; in another word, it determines whether the certificate in question is being revoked by its issuer or not. For example:

```
crlFile=file:///c:\home\certs\revoke.crl
```

This example indicates which `CRL` is used to determine if the certificate has not been revoked by its issuer.

2.27.5 Sample Registry Command File

The following is a sample command for modifying the Tuxedo registry database on a Windows platform using the Cert-C PKI encryption plug-in.

> **Note:**
>
> On a UNIX platform, you must:
>
> - use the file name `libcertctux` plus your *platform specific* dynamic library extension instead of `certctux.dll` used in Windows. For example:
>
> Solaris: `libcertctux.so.71`
>
> HP-UX: `libcertctux.sl`
> - change the file `URL` to UNIX format

Listing Sample Command for Modifying Tuxedo Registry Database on Windows

```
REM ********************************************************
         REM ** Modify Validation Interface **
         REM ********************************************************
         epifreg -r -p bea/cert-c/certificate_validation -i engine/security/
certificate_validation -v 1.0 -f certctux.dll -e
         _ep_dl_certc_validate_certificate -u caCertificateFile=file:///
c:\home\certs\root.cer -u crlFile=file:///c:\home\certs\revoke.crl

         epifreg -s -k SYSTEM/impl/bea/valfile -a InterceptionSeq=bea/cert-c/
certification_validation
         epifregedt -s -k SYSTEM/interfaces/engine/security/
certificate_validation -a DefaultImpl=bea/valfile
         REM ********************************************************
         REM ** Modify Lookup Interface **
         REM ********************************************************
         epifreg -r -p bea/cert-c/certificate_lookup -i engine/security/
certificate_lookup -v 1.0 -f certctux.dll -e_ep_dl_certc_certificate_lookup -u
         ldapUserCertificate=ldap://sagamore:389 -u
ldapBaseObject="ou=Engineer Test,o=ABC Company,c=US" -u
ldapFilterAttribute="cn" -u ldapBaseDNAttribue="c,o,ou,cn"
         epifregedt -s -k SYSTEM/interfaces/engine/security/
certificate_lookup -a DefaultImpl=bea/cert-c/certificate_lookup
         REM ********************************************************
         REM ** Modify Key Management Interface **
         REM ********************************************************
         epifreg -r -p bea/cert-c/key_management -i engine/security/
key_management -v 1.0 -f certctux.dll -e_ep_dl_certc_key_management -u
privateKeyDir=file:///c:\home\certs\
         epifregedt -s -k SYSTEM/interfaces/engine/security/key_management -a
DefaultImpl=bea/cert-c/key_management
         REM ********************************************************
         REM ** Modify Certificate Parsing Interfaces **
         REM ********************************************************
         epifreg -r -p bea/cert-c/certificate_parsing -i engine/security/
certificate_parsing -v 1.0 -f certctux.dll -
e_ep_dl_certc_certificate_parsing
         epifregedt -s -k SYSTEM/interfaces/engine/security/
certificate_parsing -a DefaultImpl=bea/cert-c/certificate_parsing
```

2.27.6 Limitations

- The "cn" attribute of distinguished name is used as key for certificate lookup, so the DN must contains the "cn=" attribute.
- There are two possible places to put a name in an X.509 v3 KC:
  - One is the subject field in the base PKC, often called the Distinguished Name or DN field.
  - The other is the subjectAltName extension. This plug-in does not support subjectAltName extension.

> **Note:**
>
> Wildcards used in a name are not supported. Empty subject fields are not allowed.

- The following tpkey_getinfo() attributes cannot retrieve ENCRYPT_ALG, ENCRYPT_BITS, SIGNATURE_ALG, or SIGNATURE_BITS information using the Cert-C PKI encryption plug-in:
  - TPKEY_SIGNATURE: cannot retrieve ENCRYPT_ALG, ENCRYPT_BITS
  - TPKEY_ENCRYPT: cannot retrieve SIGNATURE_BITS
  - TPKEY_AUTOSIGN: cannot retrieve ENCRYPT_ALG, ENCRYPT_BITS
  - TPKEY_AUTOENCRYPT: cannot retrieve SIGNATURE_BITS

> **Note:**
>
> TPKEY_DECRYPT: *can* retrieve ENCRYPT_ALG, ENCRYPT_BITS, SIGNATURE_ALG, or SIGNATURE_BITS information

TPKEY_AUTOSIGN|TPKEY_DECRYPT: *can* retrieve ENCRYPT_ALG, ENCRYPT_BITS, SIGNATURE_ALG, or SIGNATURE_BITS information

> **See Also:**
>
> tpkey_open(3c)

3
Programming Security

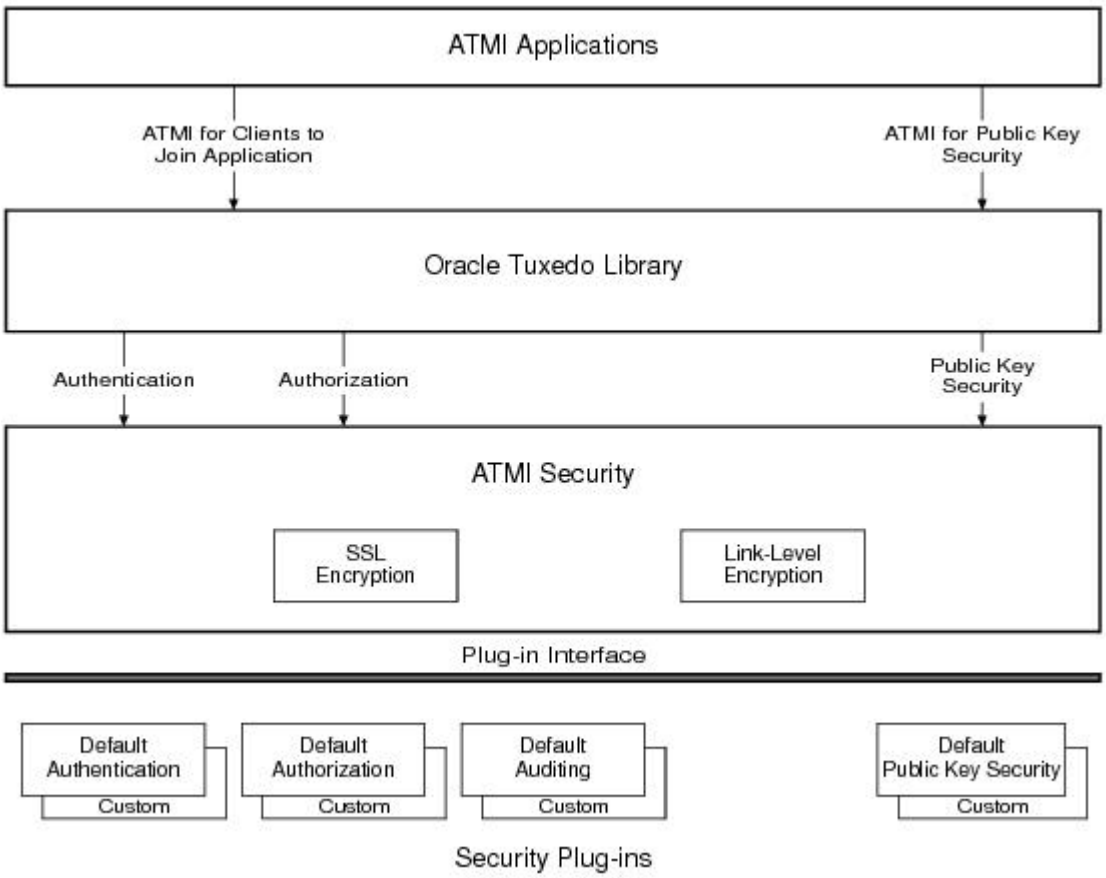
The following sections describe how to build security for your Oracle Tuxedo ATMI application into your code.

3.1 What Programming Security Means

Programming security is the task of writing security code for Application-to-Transaction Monitor Interface (ATMI) applications. In addition to the code that expresses the logic of the program, application programmers use ATMI to link their application code with the Oracle Tuxedo transaction monitor. The ATMI programming interfaces enable communication among application clients and servers running under the control of the Oracle Tuxedo transaction monitor. C and COBOL implementations of the ATMI are available.

The following figure illustrates application programmers have access to the ATMI functions for authenticating users and controlling user access, and for incorporating public key encryption techniques into their applications. Also shown is the absence, at the application level, of ATMI functions for auditing or link-level encryption. Auditing is accessed at the Oracle Tuxedo system level, and link-level encryption is configured by the application administrator.

Figure 3-1 Programming Oracle Tuxedo Security



> **Note:**
>
> - Programming an ATMI Application with Security
> - What Security Means
> - What Administering Security Means

3.2 Programming an ATMI Application with Security

The Oracle Tuxedo system offers various ATMI functions for different security needs.

If You Are Writing Security Code for . . .	Then You Use the ATMI Functions Available for . . .
Client programs so that clients can join a ATMI application and access application services.	Clients joining an ATMI application, which in turn invoke system-level calls to the authentication and authorization plug-ins.
Both client and server programs to protect the integrity and privacy of the data they exchange.	Public key security, which supports end-to-end digital signing and data encryption.

> **Note:**
>
> - Setting Up the Programming Environment

3.3 Setting Up the Programming Environment

To be able to write security code, an application programmer needs:

- Access to Oracle Tuxedo libraries and commands
- Read and execute permissions on the directories and files in the Oracle Tuxedo system directory structure

To obtain access to the required libraries and commands, you must set the `TUXCONFIG`, `TUXDIR`, `APPDIR`, and other environment variables in your environment. For details, see “How to Set Your Environment” in *Administering an Oracle Tuxedo Application at Run Time*.

The application administrator is responsible for setting the permissions on directories and files. See your administrator to get the permissions you need.

> **Note:**
>
> - Writing Security Code So Client Programs Can Join the ATMI Application
> - Writing Security Code to Protect Data Integrity and Privacy

3.4 Writing Security Code So Client Programs Can Join the ATMI Application

Client programs are responsible for gathering data from outside the application or computer, bundling the data into messages, and forwarding the messages to servers for processing. Client programs are made available to users through devices such as automatic teller machines (ATMs), data entry terminals, and graphics devices.

For default authentication and authorization, application security may be set to one of five levels. At the lowest level, no authentication is performed. At the highest level, an access control checking feature determines which users can execute a service, post an event, or enqueue (or dequeue) a message on an application queue. Setting the security level for an ATMI application is the responsibility of the application administrator.

An application programmer needs to perform two tasks so that a client program can join an ATMI application:

- Get the security data for the specific client process
- Pass that data to the Oracle Tuxedo system

The following pseudo-code in the following listing summarizes the operation of a basic client program. The security-related statements are highlighted in bold.

Listing Pseudo-code for a Client

```
main()
          {
               call tpchkauth() to check security level of ATMI application
                get usrname, cltname
                prompt for application password
                prompt for per-user password
                allocate a TPINIT buffer
                place initial client identification into TPINIT buffer
                call tpinit() to enroll as a client of the ATMI application
                allocate buffer
                do while true {
                     place user input in buffer
                     send service request
                     receive reply
                     pass reply to user }
                leave application  }
```

Most of the statements in the preceding listing are implemented by ATMI functions in either C or COBOL. The preceding listing shows only the C language implementation.

A client program written in C uses tpinit(3c) to comply with the level of security set for the ATMI application and to join the application. The argument to `tpinit()` is a pointer to a `TPINIT` buffer. To perform the same tasks in a COBOL application, a client program calls TPINITIALIZE(3cbl), specifying a pointer to a `TPINFDEF-REC` record as an argument.

> **Note:**
>
> - Getting Security Data
> - Joining the ATMI Application
> - "Writing Clients" on page 4-1 in *Programming an Oracle Tuxedo ATMI Application Using C* and *Programming an Oracle Tuxedo ATMI Application Using COBOL*
> - `tpinit(3c)` in *Oracle Tuxedo ATMI C Function Reference*
> - TPINITIALIZE(3cbl) in the *Oracle Tuxedo ATMI COBOL Function Reference*
> - Administering Public Key Security
> - Administering Authorization
> - Default Authentication and Authorization
> - Programming an ATMI Application with Security

3.5 Getting Security Data

For general-purpose client programs that are written to work with a variety of applications, the Oracle Tuxedo system provides an ATMI function that enables a client to determine the level of security required by the ATMI application that the client is trying to join. This ATMI function, implemented as tpchkauth(3c) for C and `TPCHKAUTH(3cbl)` for COBOL, is designed to work with ATMI applications using default authentication and authorization. The `tpchkauth()` and

TPCHKAUTH() functions can also be used in ATMI applications in which custom authentication and/or authorization is used. How they are used, however, depends on how the custom security features are implemented. For the most part, this discussion focuses on default authentication and authorization.

An application programmer writing in C uses tpchkauth() to check the ATMI application's security level before calling tpinit(3c), so that the client program can prompt for the application password and the user authentication data needed for the tpinit() call; tpchkauth() is called without arguments.

An application programmer writing in COBOL uses TPCHKAUTH() for the same purpose before calling TPINITIALIZE(3cbl). The syntax and functionality of TPCHKAUTH(3cbl) and TPINITIALIZE(3cbl) are the same as those of tpchkauth(3c) and tpinit(3c).

The tpchkauth() function (or TPCHKAUTH() routine) returns one of the following values.

TPNOAUTH
Nothing is required beyond the normal operating system login and file permission security. TPNOAUTH is returned for security level NONE.

TPSYSAUTH
An application password is required. The client program should prompt the user to provide the password, and should put it in the password field of the TPINIT buffer for C, or TPINFDEF-REC record for COBOL. TPSYSAUTH is returned for security level APP_PW.

The application administrator informs users of the application password, and the application programmer writes client-program code to prompt users for the application password and to put the user-supplied password, as plain text, in the password field of the TPINIT buffer or TPINFDEF-REC record. The password should not be displayed on the user's screen.

Oracle Tuxedo system-supplied client programs, such as ud, wud(1), prompt for an application password. ud() allows fielded buffers to be read from standard input and sent to a service.

TPAPPAUTH
The application password is required. The client is expected to provide a value to be passed to the authentication service in the data field of the TPINIT buffer for C, or the TPINFDEF-REC record for COBOL. TPAPPAUTH is returned for security level USER_AUTH, ACL, or MANDATORY_ACL.

The application programmer writes client-program code to furnish additional information for the application authentication service, which is provided by the AUTHSVR server for default authentication and authorization. AUTHSVR is configured by the administrator to validate the per-user authentication information with client and usernames, indicating whether the client program is allowed to join the ATMI application.

See Also:

- Joining the ATMI Application
- Writing Clients in *Programming an Oracle Tuxedo ATMI Application Using C* and *Programming an Oracle Tuxedo ATMI Application Using COBOL*
- `tpinit(3c)` and tpchkauth(3c) in the *Oracle Tuxedo ATMI C Function Reference*
- `TPINITIALIZE(3cbl)` and `TPCHKAUTH(3cbl)` in the *Oracle Tuxedo ATMI COBOL Function Reference*
- Default Authentication and Authorization
- Programming an ATMI Application with Security

3.6 Joining the ATMI Application

In a secure ATMI application, it is necessary to pass security information to the Oracle Tuxedo system via a `TPINIT` buffer for C, or a `TPINFDEF-REC` record for COBOL. The `TPINIT` buffer is a special typed buffer used by a client program to pass client identification and authentication information to the system as the client attempts to join the ATMI application. The `TPINFDEF-REC` record serves the same purpose in a COBOL application.

The header file `atmi.h`defines `TPINIT`, and the`COBOL COPY` file defines `TPINFDEF-REC`. Their structures are as follows:

TPINIT Structure		TPINFDEF-REC Structure	
char char char char long long long	usrname[MAXTIDENT+2]; cltname[MAXTIDENT+2]; passwd[MAXTIDENT+2]; grpname[MAXTIDENT+2]; flags; datalen; data;	05 USRNAME 05 CLTNAME 05 PASSWD 05 GRPNAME 05 NOTIFICATION-FLAG 88 TPU-SIG 88 TPU-DIP 88 TPU-IGN 05 ACCESS-FLAG 88 TPSA-FASTPATH 88 TPSA-PROTECTED 05 DATLEN	PIC X(30). PIC X(30). PIC X(30). PIC X(30). PIC S9(9) COMP-5. VALUE 1. VALUE 2. VALUE 3. PIC S9(9) COMP-5. VALUE 1. VALUE 2. PIC S9(9) COMP-5.

Note: MAXTIDENT may contain up to 30 characters.

The following table describes the fields in the `TPINIT` buffer/`TPINFDEF-REC` record:

Table 3-1 Fields in TPINIT Buffer/ TPINFDEF-REC Record

TPINIT Fields	TPINFDEF-REC Fields	Description
usrname	USRNAME	Username.* A null-terminated string of up to 30 characters. The username represents the caller; writers of client programs might use the same login names used to log in to the host operating system.
cltname	CLTNAME	Client name.* A null-terminated string of up to 30 characters. The client name represents the client program; writers of client programs might use this field to indicate the job function or role of the user when executing the client program.
passwd	PASSWD	Application password.* A null-terminated string of up to eight characters. tpinit() or TPINITIALIZE() validates this password by comparing it to the configured application password stored in the TUXCONFIG file.**
grpname	GRPNAME	Group name. A null-terminated string of up to 30 characters. This field is not related to security. The group name allows a client to be associated with a resource manager group that is defined in the UBBCONFIG file.
flags	NOTIFICATION-FLAG TPU-SIG TPU-DIP TPU-IGN ACCESS-FLAG TPSA-FASTPATH TPSA-PROTECTED	Notification and access flags. This field is not related to security. The flag settings specify the notification mechanism and system access mode to be used for the client. Selections override (with some exceptions) the values set in the RESOURCES section of the UBBCONFIG file.
datalen	DATALEN	Length of the user-specific data*** that follows.* To get a size value for this field, writers of client programs written in C can call TPINITNEED with the number of bytes of user-specific data expected to be sent. TPINITNEED is a macro provided in the atmi.h header file.
data	N/A	User-specific data*** of no fixed length.* tpinit() or TPINITIALIZE() forwards the user-specific data to the authentication server for validation. For default authentication, the authentication server is AUTHSVR.

* This field is required for the USER_AUTH, ACL, and MANDATORY_ACL security levels provided by default authentication and authorization. ** The binary equivalent of the UBBCONFIG file; created using tmloadcf(1). *** Usually a user password.

The client program calls tpalloc(3c) to allocate a TPINIT buffer. The following sample code in the following listing prepares to pass eight bytes of application-specific data to tpinit() and enables the client to join an ATMI application.

Listing Allocating a TPINIT Buffer and Joining an ATMI Application

```
.
        .
        .
TPINIT *tpinfo;
        .
        .
        .
if ((tpinfo = (TPINIT *)tpalloc("TPINIT",(char *)NULL,
     TPINITNEED(8))) == (TPINIT*)NULL){
     Error Routine
}
        .
        .
        .
tpinit(tpinfo)  /* join an ATMI application */
        .
        .
        .
```

When a Workstation client calls the `tpinit()` function or the `TPINITIALIZE()` routine to join an ATMI application, the following major events occur.

1. The *initiator* Workstation client and the *target* workstation listener (WSL) exchange link-level encryption (LLE) *min-max* values to be used to set up LLE on the link between the initiator Workstation client and the *target* WSH. LLE is described in Link-Level Encryption.
2. The initiator Workstation client and target WSH authenticate one another through the exchange of security tokens. For default authentication, a successful authentication ends with the transfer of client security data from the `TPINIT` buffer or `TPINFDEF-REC` record to the target WSH.
3. After a successful authentication, the initiator Workstation client sends another buffer to the target WSH containing the values of the `usrname`, `cltname`, and `flags` fields, to ensure that the target WSH receives this information for the authenticated Workstation client.

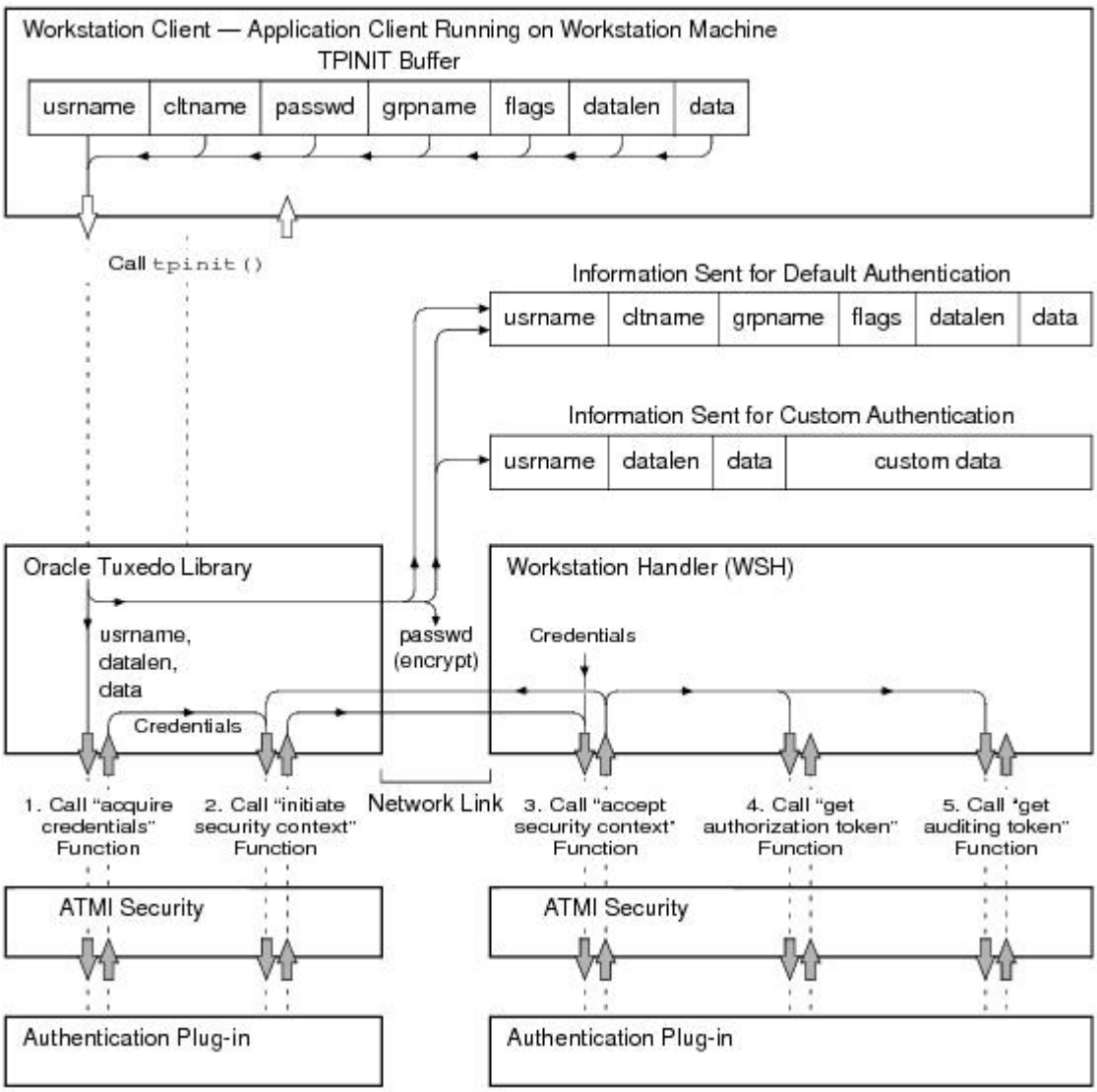
When a native client calls the `tpinit()` function or the `TPINITIALIZE()` routine to join an ATMI application, only authentication occurs. In essence, the native client authenticates with itself.

- Transferring the Client Security Data
- Calling a Service Request Before Joining the ATMI Application

3.6.1 Transferring the Client Security Data

The following listing demonstrates the transfer of data from the `TPINIT` buffer for a Workstation client. This figure illustrates the process of transferring data from the `TPINFDEF-REC` record.

Figure 3-2 Transferring Data from the TPINIT Buffer for a Workstation Client



> **Note:**
>
> The authorization procedure shown in the preceding figure is essentially the same for a native client attempting to join an ATMI application except that no network link or WSH is involved. A native client authenticates with itself.

In the preceding diagram, notice that the information sent to the Oracle Tuxedo system differs between default and custom authentication. For default authentication, the values of the `cltname`, `grpname`, and `flags` fields are delivered to the default authentication plug-in at the Workstation client by a means *other* than through the plug-in interface. However, for custom authentication, writers of client programs can include these values as well as any other values they so choose in the variable length `data` field.

For a Workstation client *and* assuming default authentication, the authentication plug-in at the Workstation client uses the `passwd`/ `PASSWD` field to encrypt the information when transmitting

the information over the network. The encryption algorithm used is 56-bit DES, where DES is an acronym for the Data Encryption Standard. The authentication plug-in at the target WSH uses the application password stored in the TUXCONFIG file to decrypt the information. For a native client, the system simply compares the passwd/ PASSWD field with the application password stored in the TUXCONFIG file.

> **Note:**
>
> At the Workstation client, the passwd/ PASSWD field is delivered to the authentication plug-in by a means *other* than through the authentication plug-in interface. At the WSH, the application password in the TUXCONFIG file is delivered to the authentication plug-in through the authentication plug-in interface during application booting.

After a successful authentication of a Workstation client, the tpinit() function ends with the sending of another buffer to the WSH containing the values of the usrname, cltname, and flags fields, to ensure that the WSH receives this information for the authenticated Workstation client. Similarly, the TPINITIALIZE() routine ends with the sending of another buffer containing the same information. A custom authentication plug-in might not send this information to the WSH during the authentication procedure, and the WSH needs this information for reporting purposes, that is, during an invocation of the tmadmin(1) printclient (pclt) command.

When a Workstation or native client passes the security check, it may initiate service requests and receive replies.

3.6.2 Calling a Service Request Before Joining the ATMI Application

If a client calls a service request (or any ATMI function) before invoking tpinit() or TPINITIALIZE() *and* assuming the SECURITY configuration for the target ATMI application is *not* set or is set to NONE, the Oracle Tuxedo system automatically invokes tpinit()/ TPINITIALIZE() with a NULL parameter. This behavior has the following consequences:

- The TPINIT/ TPINFDEF-REC feature cannot be used.
- Default values are used for client naming, unsolicited notification type, and system access mode.
- The client cannot be associated with a resource manager group.
- An application password cannot be specified.

If a client calls a service request (or any ATMI function) before invoking tpinit() or TPINITIALIZE() *and* assuming the SECURITY configuration for the target ATMI application is set to APP_PW, USER_AUTH, ACL, or MANDATORY_ACL, the Oracle Tuxedo system rejects the service request.

See Also:

- "Writing Clients" in *Programming an Oracle Tuxedo ATMI Application Using C* and *Programming an Oracle Tuxedo ATMI Application Using COBOL*
- tpinit(3c)and tpalloc(3c) in the *Oracle Tuxedo ATMI C Function Reference*
- `TPINITIALIZE(3cbl)` in the *Oracle Tuxedo ATMI COBOL Function Reference*
- Default Authentication and Authorization
- Programming an ATMI Application with Security

3.7 Writing Security Code to Protect Data Integrity and Privacy

Public key security comprises end-to-end digital signing and data encryption. Both features are supported by Oracle Tuxedo ATMI functions. ATMI applications protected by public key security are much safer for use across the Internet than programs in which this type of security is not used.

The capabilities that make end-to-end digital signing and data encryption possible are message-based digital signature and message-based encryption. Both capabilities are built upon the *PKCS-7 standard*, which is one of a set of Public-Key Cryptography Standards (PKCS) developed by RSA Laboratories in cooperation with several other leading communications companies.

Message-based digital signature ensures data integrity and non-repudiation by having the sending party bind proof of its identity to a specific message buffer. Message-based encryption protects the confidentiality of messages; only parties for whom messages are intended can decrypt and read them.

Because the unit of digital signing and encryption is an ATMI message buffer, both capabilities are compatible with existing ATMI programming interfaces and communication paradigms. It is possible for a message buffer to be both signed and encrypted. There is no required relationship between the number of digital signatures and the number of *encryption envelopes* associated with a message buffer.

Note:

Each encryption envelope identifies a recipient of the message, and contains information needed by the recipient to decrypt the message.

- ATMI Interface for Public Key Security
- Recommended Uses of Public Key Security

3.7.1 ATMI Interface for Public Key Security

The ATMI interface for public key security is a compact set of functions used to:

- Open and close key resources
- View and change key optional parameters
- Sign and seal (encrypt) message buffers

- Access the digital signature and encryption information associated with a message buffer
- Convert a typed message buffer into an exportable, machine-independent string representation, which includes the generation of any digital signatures or encryption envelopes associated with the buffer

The ATMI interfaces for public key security are available in both C and COBOL implementations. The ATMI COBOL language binding, however, does not support *message buffers*; thus, explicit signature, encryption, and query operations on individual buffers cannot be used in a COBOL application. However, key management interfaces do have a COBOL language binding, which enables signature generation in the `AUTOSIGN` mode and encryption-envelope generation in the `AUTOENCRYPT` mode. All operations related to automatic signature verification or automatic decryption apply to COBOL client and server processes.

> **Note:**
>
> The COBOL `TPKEYDEF` record is used to manage public-private keys for performing message-based digital signature and encryption operations. See “COBOL Language ATMI Return Codes and Other Definitions” in the introduction part of the *Oracle Tuxedo ATMI COBOL Function Reference* for a description of the `TPKEYDEF` record.

The following tables summarize the ATMI interfaces for public key security. Each function is also documented in the *Oracle Tuxedo ATMI C Function Reference* and the *Oracle Tuxedo ATMI COBOL Function Reference*.

Table 3-2 Functions in ATMI Interface for Public Key Security

Use This Function	To...
tpkey_open(3c)	Open a key handle for digital signature generation, message encryption, or message decryption. Keys are represented and manipulated via handles. A handle has data associated with it that is used by the ATMI application to locate or access the item named by the handle. A key may play one or more of the following roles: • Signature Generation The key identifies the calling process as being authorized to generate a digital signature under the principal's identity. (A principal may be a person or a process.) Calling `tpkey_open()` with the principal's name and either the `TPKEY_SIGNATURE` or `TPKEY_AUTOSIGN` flag returns a handle to the principal's private key and digital certificate. • Signature Verification The key represents the principal associated with a digital signature. Signature verification does not require a call to `tpkey_open()`; the verifying process uses the public key specified in the digital certificate accompanying the digitally signed message to verify the signature. • Encryption The key represents the intended principal of an encrypted message. Calling `tpkey_open()` with the principal's name and either the `TPKEY_ENCRYPT` or `TPKEY_AUTOENCRYPT` flag returns a handle to the principal's public key via the principal's digital certificate. • Decryption The key identifies the calling process as being authorized to decrypt a private message for the intended principal. Calling `tpkey_open()` with the principal's name and the `TPKEY_DECRYPT` flag returns a handle to the principal's private key and digital certificate.

Table 3-2 (Cont.) Functions in ATMI Interface for Public Key Security

Use This Function	To...
tpkey_getinfo(3c)	Get information associated with a key handle. Some information is specific to a cryptographic service provider, but the following set of attributes is supported by all providers: • PRINCIPAL The name of the principal associated with the specified key (key handle). A principal may be a person or a process, depending on how an application developer sets up public key security. Any principal specified in an ATMI application's UBBCONFIG file using the SEC_PRINCIPAL_NAME parameter become the identity of one or more system processes. (See "Specifying Principal Names" on page 2-11 and "Initializing Decryption Keys Through the Plug-ins" on page 2-56 for more detail.) • PKENCRYPT_ALG An ASN.1 Distinguished Encoding Rules (DER) object identifier of the public key algorithm used by the key for public key encryption. See the tpkey_getinfo(3c) reference page for details. • PKENCRYPT_BITS The key length of the public key algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. • SIGNATURE_ALG An ASN.1 DER object identifier of the digital signature algorithm used by the key for digital signature. See the tpkey_getinfo(3c) reference page for details. • SIGNATURE_BITS The key length of the digital signature algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. • ENCRYPT_ALG An ASN.1 DER object identifier of the symmetric key algorithm used by the key for bulk data encryption. See the tpkey_getinfo(3c) reference page for details. • ENCRYPT_BITS The key length of the symmetric key algorithm. The value must be within the range of 40 to 128 bits, inclusive. • DIGEST_ALG An ASN.1 DER object identifier of the message digest algorithm used by the key for digital signature. See the tpkey_getinfo(3c) reference page for details. • PROVIDER The name of the cryptographic service provider. • VERSION The version number of the cryptographic service provider's software.
tpkey_setinfo(3c)	Set optional attribute parameters associated with a key handle. A core set of key handle attributes is identified in the preceding description of tpkey_getinfo(). Other attributes, specific to a certain cryptographic service provider, may also be available.
tpkey_close(3c)	Close a previously opened key handle. A key handle may be opened explicitly using tpkey_open(), or implicitly (automatically) using tpenvelope().
tpsign(3c)	Mark a typed message buffer for digital signature. The public key software generates the digital signature just before the message is sent.

Table 3-2 (Cont.) Functions in ATMI Interface for Public Key Security

Use This Function	To...
`tpseal(3c)`	Mark a typed message buffer for encryption. The public key software encrypts the message just before the message is sent.
`tpenvelope(3c)`	Access the digital signature and encryption information associated with a typed message buffer. `tpenvelope()` returns status information about the digital signatures and encryption envelopes attached to a particular message buffer. It also returns the key handle associated with each digital signature or encryption envelope. The key handle for a digital signature identifies the signer, and the key handle for an encryption envelope identifies the recipient of the message.
`tpexport(3c)`	Convert a typed message buffer into an exportable, machine-independent (externalized) string representation. `tpexport()` generates any digital signatures or encryption envelopes associated with a typed message buffer just before it converts that buffer into an externalized string representation. An externalized string representation can be transmitted between processes, machines, or domains through any communication mechanism. It can be archived on permanent storage.
`tpimport(3c)`	Convert an externalized string representation back into a typed message buffer. During the conversion, `tpimport()` decrypts the message, if necessary, and verifies any associated digital signatures.

Table 3-3 COBOL Routines in ATMI Interface for Public Key Security

Use This Routine . . .	To . . .
TPKEYOPEN(3cbl)	Open a key handle for digital signature generation, message encryption, or message decryption. Keys are represented and manipulated via handles. A handle has data associated with it that is used by the ATMI application to locate or access the item named by the handle. A key may play one or more of the following roles: • Signature Generation The key identifies the calling process as being authorized to generate a digital signature under the principal's identity. (A principal can be a person or a process.) Calling `TPKEYOPEN()` with the principal's name and the `TPKEY-SIGNATURE`and `TPKEY-AUTOSIGN` settings returns a handle to the principal's public key and enables signature generation in `AUTOSIGN` mode. The public key software generates and attaches the digital signature to the message just before the message is sent. • Signature Verification The key represents the principal associated with a digital signature. Signature verification does not require a call to `TPKEYOPEN()`; the verifying process uses the public key specified in the digital certificate accompanying the digitally signed message to verify the signature. • Encryption The key represents the intended principal of an encrypted message. Calling `TPKEYOPEN()` with the principal's name and the `TPKEY-ENCRYPT` and `TPKEY-AUTOENCRYPT` settings returns a handle to the principal's public key (via the principal's digital certificate) and enables encryption in `AUTOENCRYPT` mode. The public key software encrypts the message and attaches an encryption envelope to the message; the encryption envelope enables the receiving process to decrypt the message. • Decryption The key identifies the calling process as being authorized to decrypt a private message for the intended principal. Calling `TPKEYOPEN()` with the principal's name and the `TPKEY-DECRYPT`setting returns a handle to the principal's private key and digital certificate.

Table 3-3 (Cont.) COBOL Routines in ATMI Interface for Public Key Security

Use This Routine . . .	To . . .
TPKEYGETINFO(3cbl)	Get information associated with a key handle. Some information is specific to a cryptographic service provider, but the following set of attributes is supported by all providers: • PRINCIPAL The name of the principal associated with the specified key (key handle). A principal may be a person or a process, depending on how an ATMI application developer sets up public key security. Any principal specified in an ATMI application's UBBCONFIG file using the SEC_PRINCIPAL_NAME parameter become the identity of one or more system processes. (See "Specifying Principal Names" on page 2-11 and "Initializing Decryption Keys Through the Plug-ins" on page 2-56 for more detail.) • PKENCRYPT_ALG An ASN.1 Distinguished Encoding Rules (DER) object identifier of the public key algorithm used by the key for public key encryption. See the TPKEYGETINFO(3cbl) reference page for details. • PKENCRYPT_BITS The key length of the public key algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. • SIGNATURE_ALG An ASN.1 DER object identifier of the digital signature algorithm used by the key for digital signature. See the TPKEYGETINFO(3cbl) reference page for details. • SIGNATURE_BITS The key length of the digital signature algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. • ENCRYPT_ALG An ASN.1 DER object identifier of the symmetric key algorithm used by the key for bulk data encryption. See the TPKEYGETINFO(3cbl) reference page for details. • ENCRYPT_BITS The key length of the symmetric key algorithm. The value must be within the range of 40 to 128 bits, inclusive. • DIGEST_ALG An ASN.1 DER object identifier of the message digest algorithm used by the key for digital signature. See the TPKEYGETINFO(3cbl) reference page for details. • PROVIDER The name of the cryptographic service provider. • VERSION The version number of the cryptographic service provider's software.
TPKEYSETINFO(3cbl)	Set optional attribute parameters associated with a key handle. A core set of key handle attributes is identified in the preceding description of TPKEYGETINFO(). Other attributes, specific to a certain cryptographic service provider, may also be available.
TPKEYCLOSE(3cbl)	Close a key handle previously opened using TPKEYOPEN().

3.7.2 Recommended Uses of Public Key Security

- Use tpkey_close() to release key handles used for digital signature generation or for data decryption as soon as they are no longer needed.

- To inhibit replay attacks, generate digital signatures only on message buffers that contain details identifying a specific operation. For example, a buffer that contains the message "Your deposit is confirmed" is dangerously vague. An attacker who intercepts such a message can easily reuse it. On the other hand, a message that contains many operation-specific details is much safer. An attacker who intercepts a message such as the one that follows will not be able to reuse it easily: "John Smith's deposit of $100.00, account 987654321, confirmation code 123456789, 7/31/2001, is confirmed."

See Also:

- Sending and Receiving Signed Messages
- Sending and Receiving Encrypted Messages
- Examining Digital Signature and Encryption Information
- Externalizing Typed Message Buffers
- Public Key Security
- Administering Public Key Security
- Programming an ATMI Application with Security

3.8 Sending and Receiving Signed Messages

Message-based digital signature provides end-to-end authentication and message integrity protection. For a diagram that illustrates how it works, see the figure "ATMI PKCS-7 End-to-End Digital Signing" on page 1-37.

To add a digital signature to an ATMI message buffer, the originating process or user signs the message buffer. This signature contains a cryptographically secure check sum of the message buffer's content and a timestamp based on the signer's local clock.

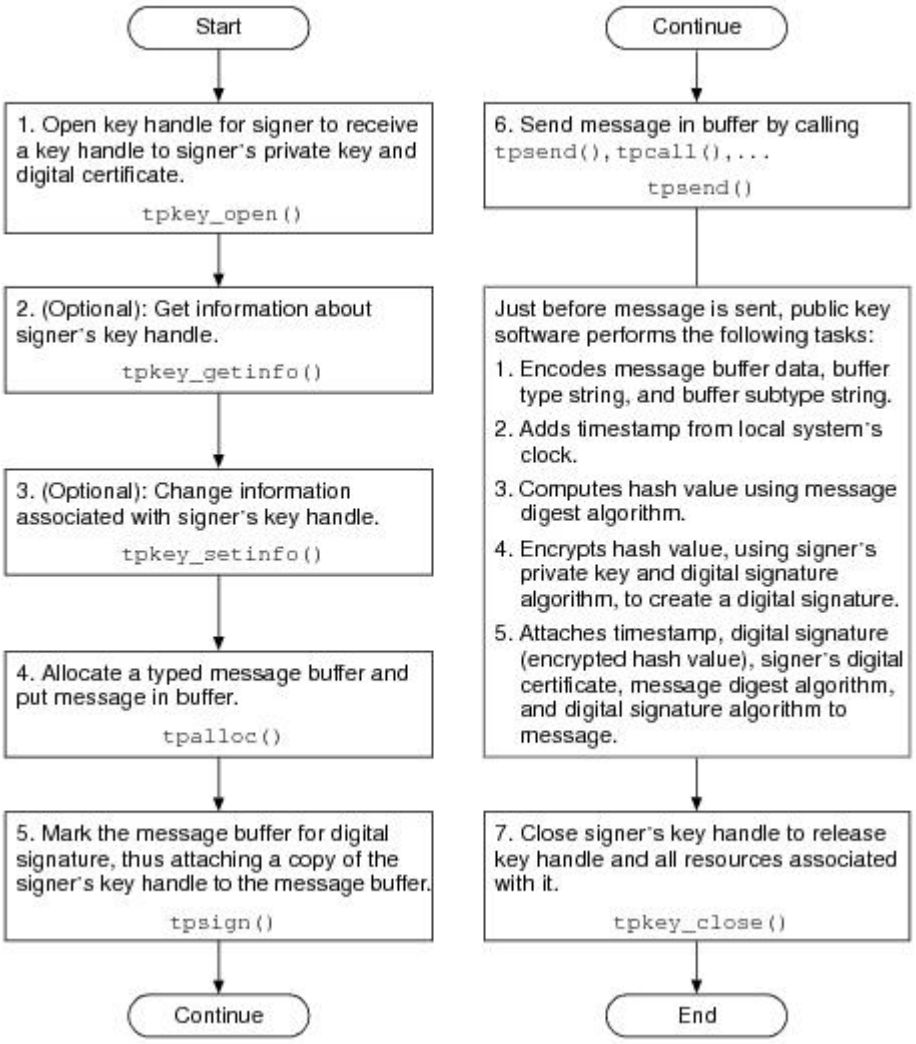
Any party with access to the message buffer can verify that the signing party's signature is authentic, that the message buffer content is unchanged, and that the timestamp is within a configured tolerance of the verifier's local clock. In addition, time-independent verification by a third party guarantees non-repudiation: the originating process or user cannot later deny authorship or claim the message was altered.

- Writing Code to Send Signed Messages
- How a Signed Message Is Received

3.8.1 Writing Code to Send Signed Messages

The following figure illustrates the procedure for writing code to send signed messages.

Figure 3-3 Procedure for Sending Signed Messages



For details about these steps and insight into how the system signs a message buffer, see the following topics.

- Step 1: Opening a Key Handle for Digital Signature
- Step 2 (Optional): Getting Key Handle Information
- Step 3 (Optional): Changing Key Handle Information
- Step 4: Allocating a Buffer and Putting a Message in the Buffer
- Step 5: Marking the Buffer for Digital Signature
- Step 6: Sending the Message
- Step 7: Closing the Signer's Key Handle
- How the System Generates a Digital Signature

3.8.1.1 Step 1: Opening a Key Handle for Digital Signature

Call the tpkey_open(3c) function or `TPKEYOPEN(3cbl)` routine to make the private key and the associated digital certificate of the signer available to the originating process. The private key is highly protected, and possession of it is equivalent to possessing the signer's identity.

In order to access the signer's private key, the originating process must prove its right to act as the signer. Proof requirements depend on the implementation of the public key plug-in interface. The default public key implementation requires a secret password from the calling process.

When the originating process calls `tpkey_open()` to open the key handle, it specifies either the `TPKEY_SIGNATURE` or `TPKEY_AUTOSIGN` flag to indicate that the handle will be used to digitally sign a message buffer. Typically, a client makes this call after calling `tpinit()`, and a server makes this call as part of initializing through `tpsvrinit()`.

Opening a key handle with the `TPKEY_AUTOSIGN` flag enables automatic signature generation: subsequently, the originating process signs message buffers automatically whenever they are sent. Using the `TPKEY_AUTOSIGN` flag is beneficial for three reasons:

- Less work is required from application programmers because fewer ATMI calls are required when operating in a secure ATMI application.
- Existing ATMI applications can leverage digital signature technology with minimal coding changes.
- The possibility of programming errors that might result in an unsigned buffer being sent over an insecure network is reduced.

The following listing describes how to open a signer's key handle. `TPKEY` is a special data type defined in the `atmi.h` header file.

Listing Opening a Signer's Key Handle Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
            TPKEY sdo_key;
            char *sdo_location;
           .
           .
           .
            if (tpkey_open(&sdo_key, "sdo",
         sdo_location,
               NULL, 0, TPKEY_SIGNATURE) ==
         -1) {
               (void) fprintf(stderr,
         "tpkey_open sdo failed
                  tperrno=%d(%s)\n",
         tperrno, tpstrerror(tperrno));
               exit(1);
            }
           .
           .
```

```
        .
    }
```

3.8.1.2 Step 2 (Optional): Getting Key Handle Information

You may want to get information about a signer's key handle to establish the validity of the key. To do so, call the tpkey_getinfo(3c) function or `TPKEYGETINFO(3cbl)` routine. While some of the information returned may be specific to a cryptographic service provider, a core set of attributes is common to all providers.

The default public key implementation supports the following signature modes for computing signatures on a message buffer:

- MD5 message digest algorithm with RSA public key signature
- SHA-1 message digest algorithm with RSA public key signature

The message digest algorithm is controlled by the `DIGEST_ALG` key attribute, and the public key signature is controlled by the `SIGNATURE_ALG` key attribute. Public key sizes from 512 to 2048 bits are supported, to allow a wide range of safety and performance options. The public key size is controlled by the `SIGNATURE_BITS` key attribute.

The default public key implementation recognizes only those digital certificate signatures that are created with these algorithm and key size choices.

The following listing describes how to get information about a signer's key handle.

Listing Getting Information About a Signer's Key Handle Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
            TPKEY sdo_key;
            char principal_name[PNAME_LEN];
            long pname_len = PNAME_LEN;
           .
           .
           .
            if (tpkey_getinfo(sdo_key, "PRINCIPAL",principal_name,&pname_len,
0) == -1) {(void) fprintf(stdout,
         "Unable to get information about principal: %d(%s)\n",
                     tperrno,
         tpstrerror(tperrno));
           .
           .
           .
               exit(1);
            }
           .
           .
           .
         }
```

3.8.1.3 Step 3 (Optional): Changing Key Handle Information

To set optional attributes associated with a signer's key handle, call the tpkey_setinfo(3c) function or TPKEYSETINFO(3cbl) routine. Key handle attributes vary, depending on the cryptographic service provider.

The following listing example code describes how to change information associated with a signer's key handle.

Listing Changing Information Associated with a Signer's Key Handle Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
            TPKEY sdo_key;
            static const unsigned char sha1_objid[] = {
            0x06, 0x05, 0x2b, 0x0e, 0x03,0x02, 0x1a
            };
           .
           .
           .
            if (tpkey_setinfo(sdo_key, "DIGEST_ALG", (void *) sha1_objid,
        sizeof(sha1_objid), 0) == -1){
        (void) fprintf(stderr, "tpkey_setinfo failed
         tperrno=%d(%s)\n",
         tperrno, tpstrerror(tperrno));
         return(1);
         }
           .
           .
           .
        }
```

3.8.1.4 Step 4: Allocating a Buffer and Putting a Message in the Buffer

To allocate a typed message buffer, call the tpalloc(3c) function. Then put a message in the buffer.

3.8.1.5 Step 5: Marking the Buffer for Digital Signature

To mark, or register, the message buffer for digital signature, call the tpsign(3c) function. By calling this function, you attach a copy of the signer's key handle to the message buffer. If you open the key with the `TPKEY_AUTOSIGN` flag, each message that you send is automatically marked for digital signature without an explicit call to `tpsign()`; signature parameters are stored and associated with the buffer for later use.

> **Note:**
>
> In COBOL applications, use the `AUTOSIGN` settings member to create a digital signature. See TPKEYOPEN(3cbl).

The following example code shows how to mark a message buffer for digital signature.

Listing Marking a Message Buffer For Digital Signature Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
           TPKEY sdo_key;
           char *sendbuf, *rcvbuf;
           .
           .
           .
          if (tpsign(sendbuf, sdo_key, 0) == -1) {
         (void) fprintf(stderr, "tpsign failed tperrno=%d(%s)\n",
            tperrno,tpstrerror(tperrno));
          tpfree(rcvbuf);
          tpfree(sendbuf);
          tpterm();
         (void) tpkey_close(sdo_key,0);
         exit(1);
            }
           .
           .
           .
         }
```

3.8.1.6 Step 6: Sending the Message

After the message buffer has been marked for digital signature, transmit the message buffer using one of the following C functions or COBOL routines:

- `tpcall()` or `TPCALL`
- `tpbroadcast()` or `TPBROADCAST`
- `tpconnect()` or `TPCONNECT`
- `tpenqueue()` or `TPENQUEUE`
- `tpforward()`
- `tpnotify()` or `TPNOTIFY`
- `tppost()` or `TPPOST`
- `tpreturn()` or `TPRETURN`
- `tpsend()` or `TPSEND`

3.8.1.7 Step 7: Closing the Signer's Key Handle

Call the tpkey_close(3c) function or `TPKEYCLOSE(3cbl)` routine to release the signer's key handle and all resources associated with it.

3.8.1.8 How the System Generates a Digital Signature

The public key software digitally signs a message buffer before sending it. If a signed buffer is transmitted more than once, the software generates a new signature for each communication. This process makes it possible to modify a message buffer after marking the buffer to be digitally signed.

The public key software generates a digital signature by performing the following three-step procedure.

1. digest [message_buffer_data + buffer_type_string + buffer_subtype_string] = hash1
2. digest [hash1 + local_timestamp + PKCS-7_message_type] = hash2
3. {hash2}signer's_private_key = encrypted_hash2 = digital_signature

The notation *digest*[*something*] means that a hash value has been computed for *something* using a message digest algorithm—in this case, MD5 or SHA-1. The notation {*something*}*key* means that *something* has been encrypted or decrypted using *key*. In this case, the computed hash value is encrypted using the signer's private key.

- Signature Timestamp
- Multiple Signatures
- Signed Message Content

3.8.1.8.1 Signature Timestamp

A digital signature includes a timestamp from the local system's clock. Inclusion of such a timestamp ensures that any tampering with the timestamp value will be detected when the recipient verifies the signature. In addition, a copy of the timestamp accompanies the digitally signed message when the message is routed to its destination. Time resolution is to the second. Timestamps are stored in PKCS-9 `SigningTime` format.

3.8.1.8.2 Multiple Signatures

More than one signature can be associated with a message buffer, which means that any number of signers can sign a message buffer in parallel. A signer can be a person or a process. Each signer signs the message buffer using his, her, or its private key.

Different signatures may be based on different message digest or digital signature algorithms. If two signers use the same message digest and digital signature algorithm, the hash value is computed for only one of them.

3.8.1.8.3 Signed Message Content

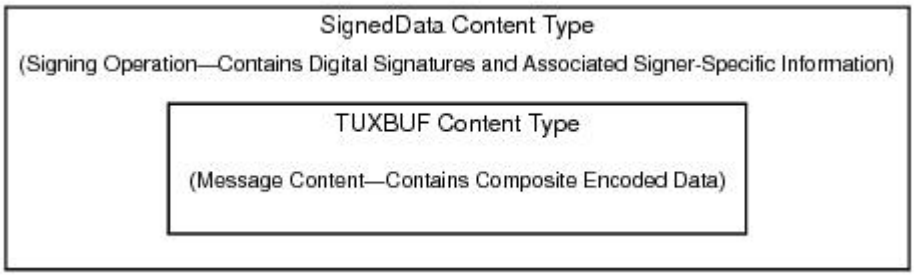
A digitally signed message buffer is represented in the PKCS-7 format as a version 1 `SignedData` content type. The `SignedData` content type, as used by the Oracle Tuxedo system, consists of the following items:

- One or more digital signatures, each with its own set of signer-specific information, such as:
- Signer's X.509v3 certificate
- Message digest and digital signature algorithm identifiers
- Timestamp based on the local clock

- Message content, which is a composite of message buffer data, buffer type string, and buffer subtype string represented in the Oracle Tuxedo encoded format. The encoded format allows a message buffer's signature to be verified on any machine architecture.

The following figure illustrates the message content is enveloped by `SignedData` content type.

Figure 3-4 SignedData Content Type



3.8.2 How a Signed Message Is Received

No ATMI application code is needed to receive a signed message buffer. The public key software automatically verifies the attached digital signatures and passes the message to the receiving process.

Upon receiving a signed message buffer, the public key software, operating on behalf of the receiving process, performs the following tasks.

1. Reads the digital signature information attached to the received message, including the signer's digital certificate, message digest algorithm, digital signature algorithm, and signature timestamp.
2. Decrypts the attached digital signature (encrypted hash value) using the signer's public key (found in the signer's digital certificate) and the digital signature algorithm.
3. Recomputes the hash value for the received message, as shown in the following two-step procedure.
   a. digest[message_buffer_data + buffer_type_string + buffer_subtype_string] = hash1
   b. digest[hash1 + received_timestamp + PKCS-7_message_type] = hash2

   The notation *digest*[*something*] means that a hash value has been computed for *something* using a message digest algorithm—in this case, MD5 or SHA-1.
4. Compares the recomputed hash value with the received hash value; if the two are not identical, discards the message buffer.
5. Compares the received timestamp with the local system's clock; if the timestamp is not within a configured tolerance, discards the message buffer.
6. If the message buffer successfully passes the checks performed in Steps 4 and 5, the public key software decodes the message buffer data, buffer type string, and buffer subtype string, and then passes the message to the receiving process. This step reverses the encoding performed by the originating process. (The Oracle Tuxedo encoded format allows a message buffer's signature to be verified on any machine architecture.)

> **Note:**
>
> If none of the attached digital signatures can be verified, the receiving process does *not* receive the message buffer. Moreover, the receiving process has no knowledge of the message buffer.

- Verifying Digital Signatures
- Verifying and Transmitting an Input Buffer's Signatures
- Replacing an Output Buffer's Signatures

3.8.2.1 Verifying Digital Signatures

The public key software automatically verifies digital signatures whenever a signed message buffer enters a client process, server process, or any system process that needs to access the content of the message buffer. If a system process is acting as a *conduit* (that is, if it is not reading the content of the message), then the attached digital signatures need not be verified. Bridges and workstation handlers (WSHs) are examples of system processes acting as conduits.

The signature timestamp is based on an unsynchronized clock, and therefore cannot be fully trusted, especially if the signature is performed on a PC or personal workstation. However, a server may reject requests with timestamps that are too old or dated too far into the future. The capability to reject a request based on the timestamp provides a measure of protection against replay attacks.

3.8.2.2 Verifying and Transmitting an Input Buffer's Signatures

If a message buffer is passed to an ATMI function (such as `tpacall()`) as an input parameter, the public key software verifies any signatures previously attached to the message and then forwards the message. This behavior enables a secure, verified transfer of information with signatures from multiple processes.

If a server modifies a received message buffer and then forwards the buffer, the original signature is no longer valid. In this case, the public key software detects the invalid signature and silently discards it. For an example of the process, see Discarding an Input Buffer's Encryption Envelopes.

3.8.2.3 Replacing an Output Buffer's Signatures

If a message buffer is passed to an ATMI function (such as `tpgetreply()`) as an output parameter, the public key software deletes any signature information associated with the buffer. This information includes any *pending* signatures and signatures from previous uses of the buffer. (A pending signature is a signature that is registered with a message buffer.)

New signature information might be associated with the new buffer content after successful completion of this operation.

Note:

- Sending and Receiving Encrypted Messages
- Examining Digital Signature and Encryption Information
- Externalizing Typed Message Buffers
- Public Key Security
- Administering Public Key Security
- Programming an ATMI Application with Security

3.9 Sending and Receiving Encrypted Messages

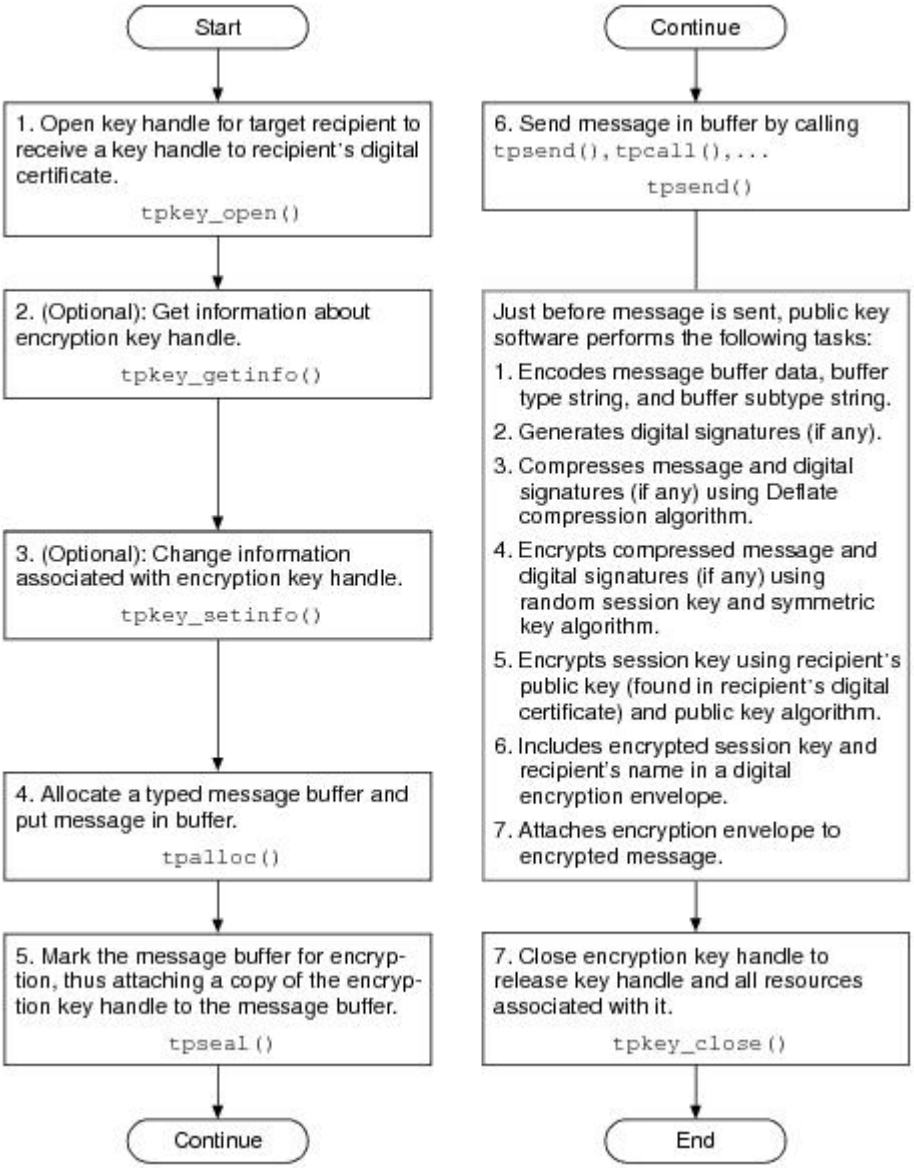
Message-based encryption provides end-to-end data privacy. For a diagram that illustrates how it works, see the figure “ATMI PKCS-7 End-to-End Encryption” on page 1-42.

A message is encrypted just before it leaves the originating process, and remains encrypted until it is received by the final destination process. It is opaque at all intermediate transit points (including operating system message queues, system processes, and disk-based queues) and during network transmission over inter-server network links.

- Writing Code to Send Encrypted Messages
- Writing Code to Receive Encrypted Messages

3.9.1 Writing Code to Send Encrypted Messages

Figure 3-5 Procedure for Sending Encrypted Messages



For details about these steps and insight into how the system encrypts a message buffer, see the following topics.

- Step 1: Opening a Key Handle for Encryption
- Step 2 (Optional): Getting Key Handle Information
- Step 3 (Optional): Changing Key Handle Information
- Step 4: Allocating a Buffer and Putting a Message in the Buffer
- Step 5: Marking the Buffer for Encryption

3.9.1.1 Step 1: Opening a Key Handle for Encryption

Call the tpkey_open(3c) function or TPKEYOPEN(3cbl) routine to make the digital certificate of the target recipient available to the originating process. The target recipient might be a client, a service, a server group, a gateway group, a server machine, or an entire domain of servers.

When the originating process calls `tpkey_open()` to open the key handle, it specifies either the `TPKEY_ENCRYPT` or `TPKEY_AUTOENCRYPT` flag to indicate that the handle will be used to encrypt a message buffer. Typically, a client makes this call after calling `tpinit()`, and a server makes this call as part of initializing through `tpsvrinit()`.

Opening a key handle with the `TPKEY_AUTOENCRYPT` flag enables automatic encryption: subsequently, the originating process encrypts message buffers automatically whenever they are sent. Using the `TPKEY_AUTOENCRYPT` flag is beneficial for three reasons:

- Less work is required from application programmers because fewer ATMI calls are required when operating in a secure ATMI application.
- Existing ATMI applications can leverage encryption technology with minimal coding changes.
- The possibility of programming errors that might result in an unencrypted (plaintext) buffer being sent over an insecure network is reduced.

Listing describes how to open an encryption key handle. `TPKEY` is a special data type defined in the `atmi.h` header file.

Listing Opening an Encryption Key Handle Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
         TPKEY tu_key;
           .
           .
           .
         if (tpkey_open(&tu_key,
         "TOUPPER", NULL,
          NULL, 0, TPKEY_ENCRYPT) == -1)
         {
         (void) fprintf(stderr,
         "tpkey_open tu failed
          tperrno=%d(%s)\n",tperrno, tpstrerror(tperrno));
          exit(1);
            }
           .
           .
           .
         }
```

3.9.1.2 Step 2 (Optional): Getting Key Handle Information

You may want to get information about an encryption key handle to establish the validity of the key. To do so, call the tpkey_getinfo(3c) function or `TPKEYGETINFO(3cbl)` routine. While some of the information returned may be specific to a cryptographic service provider, a core set of attributes is common to all providers.

The default public key implementation supports three algorithms for bulk data encryption of message content:

- DES (DES-CBC)—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) and is exportable outside the United States. (DES stands for the Data Encryption Standard.)
- 3DES (two-key triple-DES)—a 128-bit block cipher run in Encrypt-Decrypt-Encrypt (EDE) mode. 3DES provides two 56-bit keys (in effect, a 112-bit key) and is *not* exportable outside the United States.
- RC2—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, RC2 can be used with keys of virtually unlimited length, but the public key software restricts the key length to 128 bits. (RC2 stands for Rivest's Cipher 2.)

Encryption strength is controlled by the `ENCRYPT_BITS` key attribute, and the algorithm is controlled by the `ENCRYPT_ALG` key attribute. When an algorithm with fixed key length is set in `ENCRYPT_ALG`, the value of `ENCRYPT_BITS` is automatically adjusted to match.

The following listing describes how to get information about an encryption key handle.

Listing Getting Information About an Encryption Key Handle Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
         TPKEY tu_key;
         char principal_name[PNAME_LEN];
         long pname_len = PNAME_LEN;
           .
           .
           .
          if (tpkey_getinfo(tu_key,
         "PRINCIPAL",
          principal_name,
         &pname_len, 0) == -1) {
         (void) fprintf(stdout,
         "Unable to get information
          about
         principal: %d(%s)\n",
         tperrno,
         tpstrerror(tperrno));
           .
           .
           .
          exit(1);
```

```
    }
  .
  .
  .
}
```

3.9.1.3 Step 3 (Optional): Changing Key Handle Information

To set optional attributes associated with an encryption key handle, call the tpkey_setinfo(3c) function or TPKEYSETINFO(3cbl) routine. Key handle attributes vary, depending on the cryptographic service provider.

Listing describes how to change information associated with an encryption key handle.

Listing Changing Information Associated with an Encryption Key Handle Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
         TPKEY tu_key;
         static const unsigned char rc2_objid[] = {
         0x06, 0x08, 0x2a, 0x86, 0x48,0x86, 0xf7, 0x0d, 0x03, 0x02
            };
           .
           .
           .
         if (tpkey_setinfo(tu_key, "ENCRYPT_ALG", (void *) rc2_objid,
         sizeof(rc2_objid), 0) == -1){
         (void) fprintf(stderr, "tpkey_setinfo failed
          tperrno=%d(%s)\n",
          tperrno,tpstrerror(tperrno));
         return(1);
            }
           .
           .
           .
         }
```

3.9.1.4 Step 4: Allocating a Buffer and Putting a Message in the Buffer

To allocate a typed message buffer, call the tpalloc(3c) function. Then put a message in the buffer.

3.9.1.5 Step 5: Marking the Buffer for Encryption

To mark, or register, the message buffer for encryption, call the tpseal(3c) function. By calling this function, you attach a copy of the encryption key handle to the message buffer. If you open the key with the `TPKEY_AUTOENCRYPT` flag, each message that you send is automatically marked for encryption without an explicit call to `tpseal()`.

> **Note:**
>
> In COBOL applications, use the `AUTOENCRYPT` settings member to encrypt a message buffer. See TPKEYOPEN(3cbl) .

The following listing describes how to mark a message buffer for encryption.

Listing Marking a Message Buffer for Encryption Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
         TPKEY tu_key;
         char *sendbuf, *rcvbuf;
           .
           .
           .
         if (tpseal(sendbuf, tu_key, 0) == -1) {
         (void) fprintf(stderr,
         "tpseal failed tperrno=%d(%s)\n",
          tperrno,tpstrerror(tperrno));
          tpfree(rcvbuf);
          tpfree(sendbuf);
          tpterm();
         (void) tpkey_close(tu_key,
         0);
         exit(1);
            }
           .
           .
           .
         }
```

3.9.1.6 Step 6: Sending the Message

After the message buffer has been marked for encryption, transmit the message buffer using one of the following C functions or COBOL routines:

- `tpcall()` or `TPCALL`
- `tpbroadcast()` or `TPBROADCAST`
- `tpconnect()` or `TPCONNECT`
- `tpenqueue()` or `TPENQUEUE`
- `tpforward()`
- `tpnotify()` or `TPNOTIFY`
- `tppost()` or `TPPOST`
- `tpreturn()` or `TPRETURN`
- `tpsend()` or `TPSEND`

3.9.1.7 Step 7: Closing the Encryption Key Handle

Call the tpkey_close(3c) function or TPKEYCLOSE(3cbl) routine to release the encryption key handle and all resources associated with it.

3.9.1.8 How the System Encrypts a Message Buffer

Just before a message buffer is sent, the public key software encrypts the message and attaches an encryption envelope; the encryption envelope enables the target recipient to decrypt the message. If a sealed buffer is transmitted more than once, encryption is performed for each transmission. This process makes it possible to modify a message buffer after marking the buffer to be encrypted.

The public key software encrypts the content of the message buffer and generates an encryption envelope for the recipient of the encrypted message by performing the following two-step procedure.

1. {message_buffer_data + buffer_type_string + buffer_subtype_string}session_key = encrypted_message
2. {session_key}recipient's_public_key = encrypted_session_key = encryption_envelope_for_recipient

The notation {*something*}*key* means that *something* has been encrypted or decrypted using *key*. In Step 1, a message buffer is encrypted using the session key, and in step 2, the session key is encrypted using the recipient's public key.

- Multiple Message Recipients
- Encrypted Message Content

3.9.1.8.1 Multiple Message Recipients

More than one encryption envelope can be associated with a message buffer, which means that multiple recipients, with different private keys, can receive and decrypt an encrypted message. A recipient can be a person or a process. When a message is encrypted for multiple recipients, it is encrypted only once, but the session key is encrypted with the public key of each recipient. All encryption envelopes are attached to the encrypted message.

If several encryption envelopes are associated with one message buffer, all of them must use the same symmetric key algorithm and the same key size for that algorithm.

3.9.1.8.2 Encrypted Message Content

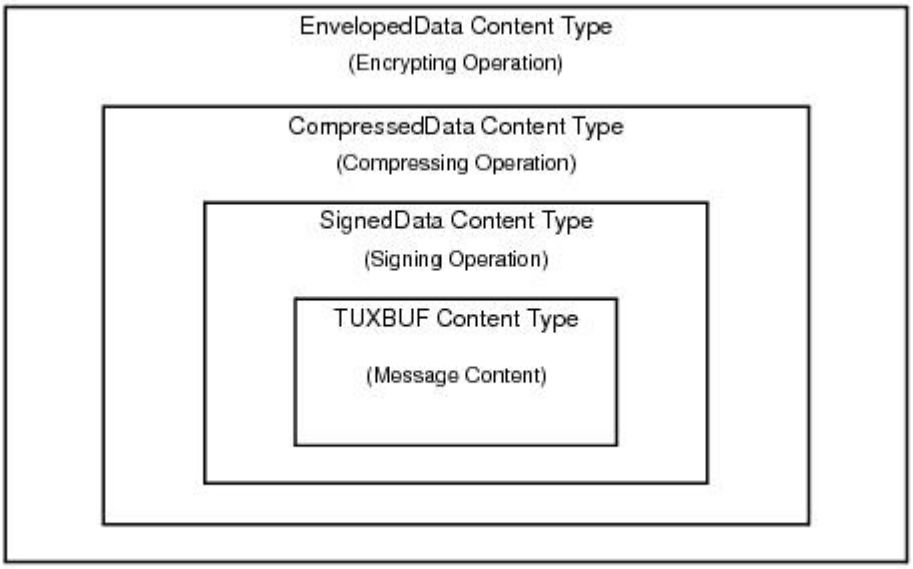
An encrypted message buffer is represented in the PKCS-7 format as a version 0 `EnvelopedData` content type. The `EnvelopedData` content type, as used by the Oracle Tuxedo system, consists of the following items:

- A list of recipients (in plaintext) that can be read by any ATMI process
- Encryption envelopes for one or more recipients
- Public key algorithm (and any associated parameters) under which the session key was encrypted
- Symmetric key algorithm (and any associated parameters) under which the bulk data was encrypted

- Encrypted bulk data, which is a composite of message buffer data, buffer type string, buffer subtype string, and digital signatures (if any) that have undergone the following transformations:
  - Conversion of the message buffer data, buffer type string, and buffer subtype string into the Oracle Tuxedo encoded format to form the composite encoded data. (The Oracle Tuxedo encoded format allows a message buffer to be decrypted on any machine architecture.)
  - Compression of the composite encoded data and digital signatures (if any) using the Deflate compression algorithm to form the composite compressed data.
  - Encryption of the composite compressed data under a randomly generated session key and symmetric key algorithm (identified earlier in this list) to form the encrypted bulk data.

The following figure illustrates the envelope hierarchy for the `EnvelopedData` content type. The `SignedData` content type is part of the hierarchy *only* if the message to which it belongs has one or more associated digital signatures.

Figure 3-6 EnvelopedData Content Type



As shown in the preceding figure, a message buffer may be both signed and encrypted. No relationship is required between the number of digital signatures and the number of encryption envelopes associated with a message buffer.

When both processes are performed on a message buffer, signatures are generated first, on unencrypted data. The number of attached signatures and the identity of signing parties are then obscured by the bulk data encryption.

> **Note:**
>
> A suitable decryption key must be available to access message data before signatures can be verified.

3.9.2 Writing Code to Receive Encrypted Messages

The procedure for writing code to receive encrypted messages consists of the following steps:

1. Call `tpkey_open()` to open a key handle for the target recipient. `tpkey_open` returns a key handle to the recipient's private key and digital certificate.
2. (Optional): Call `tpkey_getinfo()` to get information about the decryption key handle.
3. (Optional): Call `tpkey_setinfo()` to change information associated with the decryption key handle.
4. Call `tpkey_close()` to close the decryption key handle. `tpkey_close()` releases the key handle and all resources associated with it.

For details about these steps and insight into how the system decrypts a message buffer, see the following topics.

- Step 1: Opening a Key Handle for Decryption
- Step 2 (Optional): Getting Key Handle Information
- Step 3 (Optional): Changing Key Handle Information
- Step 4: Closing the Decryption Key Handle
- How the System Decrypts a Message Buffer

3.9.2.1 Step 1: Opening a Key Handle for Decryption

Call the tpkey_open(3c) function or `TPKEYOPEN(3cbl)` routine to make the private key and the associated digital certificate of the target recipient available to the receiving process. The receiving process might be a client, a service, a server group, a gateway group, a server machine, or an entire domain of servers.

An application administrator can configure the ATMI application's `UBBCONFIG` file such that decryption key handles are opened automatically when the ATMI application is booted. No more than one decryption key handle per server may be used with this method. See "Initializing Decryption Keys Through the Plug-ins" on page 2-56 for details.

If an ATMI application is not configured to open a decryption key handle for the receiving process during booting, the receiving process initiates its own `tpkey_open()` call. Or, if the receiving process wants to open another decryption key handle, the receiving process makes an additional `tpkey_open()` call.

In order to access the target recipient's private key, the receiving process must prove its right to act as the target recipient. Proof requirements depend on the implementation of the public key plug-in interface. The default public key implementation requires a secret password from the calling process.

When the receiving process calls `tpkey_open()` to open the key handle, it specifies the `TPKEY_DECRYPT` flag to indicate that the handle will be used to decrypt a message buffer. Typically, a client makes this call after calling `tpinit()`, and a server makes this call as part of initializing through `tpsvrinit()`.

The following listing describes how to open a decryption key handle. `TPKEY` is a special data type defined in the `atmi.h` header file.

Listing Opening a Decryption Key Handle Example

```
TPKEY tu_key;
         tpsvrinit(argc, argv)
         int argc;
         char **argv;
         #endif
         {
         char  *tu_location;
           .
           .
           .
          if (tpkey_open(&tu_key,
         "TOUPPER", tu_location,
          NULL, 0, TPKEY_DECRYPT) == -1)
         {
         userlog("Unable to open
         private key: %d(%s)",
         tperrno,
         tpstrerror(tperrno));
         return(-1)
         }
           .
           .
           .
         }
```

3.9.2.2 Step 2 (Optional): Getting Key Handle Information

You may want to get information about a decryption key handle to establish the validity of the key. To do so, call the tpkey_getinfo(3c) function or TPKEYGETINFO(3cbl) routine. While some of the information returned may be specific to a cryptographic service provider, a core set of attributes is common to all providers.

The following listing describes hows how to get information about a decryption key handle.

Listing Getting Information About a Decryption Key Handle Example

```
TPKEY tu_key;
         tpsvrinit(argc, argv)
         int argc;
         char **argv;
         #endif
         {
         char principal_name[PNAME_LEN];
         long pname_len = PNAME_LEN;
           .
           .
           .
         if (tpkey_getinfo(tu_key,
         "PRINCIPAL",
          principal_name,
         &pname_len, 0) == -1) {
         (void) fprintf(stdout,
         "Unable to get information
         about
```

```
principal: %d(%s)\n”,
tperrno,
tpstrerror(tperrno));
  .
  .
  .
exit(1);
   }
  .
  .
  .
}
```

3.9.2.3 Step 3 (Optional): Changing Key Handle Information

To set optional attributes associated with a decryption key handle, call the tpkey_setinfo(3c) function or TPKEYSETINFO(3cbl) routine. Key handle attributes vary, depending on the cryptographic service provider.

The following listing describes how to change information associated with a decryption key handle.

Listing Changing Information Associated with a Decryption Key Handle Example

```
TPKEY tu_key;
         tpsvrinit(argc, argv)
         int argc;
         char **argv;
         #endif
         {
         TM32U mybits = 128;
           .
           .
           .
         if (tpkey_setinfo(tu_key,
         “ENCRYPT_BITS”, &mybits,
         sizeof(mybits), 0) == -1)
         {
         (void) fprintf(stderr,
         “tpkey_setinfo failed
          tperrno=%d(%s)\n”,
          tperrno,
         tpstrerror(tperrno));
          return(1);
           }
           .
           .
           .
         }
```

3.9.2.4 Step 4: Closing the Decryption Key Handle

Call the tpkey_close(3c) function or TPKEYCLOSE (3cbl) routine to release the decryption key handle and all resources associated with it.

3.9.2.5 How the System Decrypts a Message Buffer

The public key software automatically decrypts an encrypted message buffer whenever it enters an Oracle Tuxedo client process, server process, or any system process that needs to access the content of the message buffer. For automatic decryption to succeed, the receiving process must have opened a decryption key (type `TPKEY_DECRYPT`) corresponding to a recipient identified in one of the attached encryption envelopes.

Upon receiving an encrypted message, the public key software, operating on behalf of the receiving process, performs the following tasks.

1. Reads the target recipient's name on the attached encryption envelope.
2. To recover the session key, decrypts the recipient's encryption envelope using the recipient's private key and the public key algorithm.
3. Decrypts the message using the recovered session key and the symmetric key algorithm.
4. Uncompresses the message.
5. Verifies digital signatures if any. (See How a Signed Message Is Received.)
6. If the message buffer successfully passes the check performed in step 5, the public key software decodes the message buffer data, buffer type string, and buffer subtype string, and then passes the plaintext message to the receiving process. This step reverses the encoding performed by the originating process. (The Oracle Tuxedo encoded format allows a message buffer to be decrypted on any machine architecture.)
7. If none of the attached digital signatures can be verified or the message buffer cannot be decrypted, the receiving process does *not* receive the message buffer. Moreover, the receiving process has no knowledge of the message buffer.

If a system process is acting as a *conduit* (that is, if it is not reading the content of the message), then the message need not be decrypted. Bridges and workstation handlers (WSHs) are examples of system processes acting as conduits.

The WSH is a special example of a conduit. If a WSH is configured for data-dependent routing, it needs to read the received message buffer to determine how to route the buffer. The public key software makes a copy of the received message buffer, decrypts the copy, and then passes the decrypted copy to the WSH. The WSH analyzes the decrypted copy to determine how to route the buffer, and then routes the original message buffer *unchanged* to the appropriate server. (For more detail about the interaction between data-dependent routing and public key security, see Compatibility/Interaction with Data-dependent Routing.)

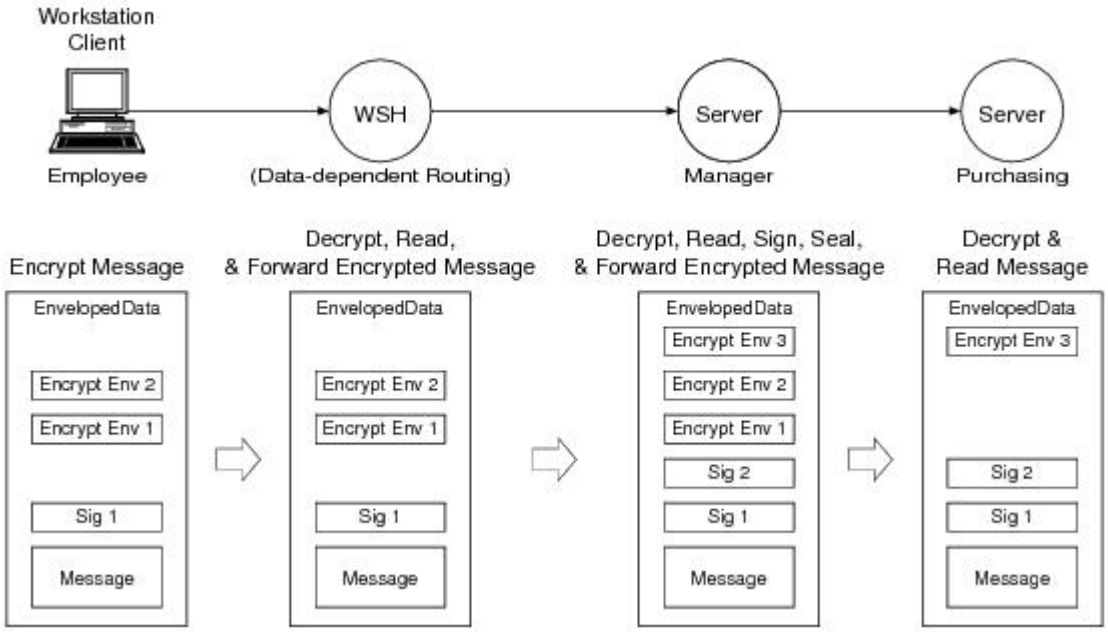
- Discarding an Input Buffer's Encryption Envelopes
- Replacing an Output Buffer's Encryption Envelopes

3.9.2.5.1 Discarding an Input Buffer's Encryption Envelopes

If a message buffer is passed to an ATMI function (such as `tpacall()`) as an input parameter, the public key software discards any encryption envelopes previously attached to the message. This behavior prevents the target recipients for the original message from receiving any modifications made by an intermediate process.

As an example of this process, consider the scenario shown in the following figure.

Figure 3-7 Forwarding a Signed and Encrypted Message Example



A server process named `Manager` receives a signed and encrypted message buffer from a client process named `Employee`, decrypts and reads the received message buffer, signs and seals it for a service named `Purchasing`, and then forwards the message to `Purchasing`.

The following is a detailed description of how this operation is performed.

1. The workstation handler (WSH) receives the signed and encrypted message buffer from the employee and forwards it *as is*.
   The WSH process is configured for data-dependent routing, which is briefly described in "How the System Decrypts a Message Buffer" on page 3-46. The public key software uses a decryption key previously opened for the WSH process to decrypt a copy of the received message buffer, and then passes the decrypted copy to the WSH. After analyzing the decrypted copy, the WSH routes the received message buffer to the `Manager` process *as is*.

   If the WSH process is *not* configured for data-dependent routing, the `Employee` process does not need to `tpseal()` the message buffer for the WSH process, and the WSH process does not need to open a decryption key.
   Regardless of how it is configured, the WSH does not verify digital signatures.
2. When the message buffer arrives at the `Manager` process, the public key software:
   a. Decrypts the message buffer using a decryption key previously opened for the `Manager` process.
   b. Verifies the employee's signature.
   c. Passes the message *without* digital signature or encryption information to the `Manager`.
3. The `Manager` calls `tpenvelope()` *repeatedly* to find out about the digital signature and encryption information associated with the message buffer. `tpenvelope()` returns:
   - Digital signature information, including the signer's public key and a digital-signature status of `TPSIGN_OK`

- Encryption information, including the public keys of the WSH process and the `Manager` process itself

4. The `Manager` calls `tpkey_getinfo()` with the signer's public key as an argument, to obtain more information about the signer, including the signer's principal name.
5. If the `Manager` determines that the signer is a known employee and that the employee's request (as stated in the message content) is valid, the `Manager` proceeds as follows.
   a. Calls `tpsign()` to mark the message buffer for digital signature by the `Manager`.
   b. Calls `tpseal()` to mark the message buffer to be encrypted for `Purchasing`.
   c. Calls `tpforward()` (or some other function used to transmit data) to send the message to `Purchasing`.

Prior to transmission, the public key software performs the following tasks.

1. Generates a digital signature for the Manager.
2. Verifies the employee's digital signature.
3. Encrypts the message content and associated digital signatures.
4. Creates an encryption envelope for Purchasing.

3.9.2.5.2 Replacing an Output Buffer's Encryption Envelopes

If a message buffer is passed to an ATMI function (such as `tpgetrply()`) as an output parameter, the public key software deletes any encryption information associated with the buffer. This information includes any *pending* seals, or seals from previous uses of the buffer. (A pending seal is a recipient's seal that is registered with a message buffer.)

New encryption information might be associated with the new buffer content after successful completion of the operation.

See Also:

- Examining Digital Signature and Encryption Information
- Externalizing Typed Message Buffers
- Public Key Security
- Administering Public Key Security
- Programming an ATMI Application with Security

3.10 Examining Digital Signature and Encryption Information

The public key software maintains the order in which:

- Digital-signature registration requests and digital signatures are attached to a message buffer
- Encryption registration requests and encryption envelopes are attached to a message buffer

A process obtains this information by calling the `tpenvelope()` function with the target message buffer as an argument. `tpenvelope()` is described on the tpenvelope(3c) reference page in the *Oracle Tuxedo ATMI C Function Reference*.

There may be multiple occurrences of digital-signature registration requests, digital signatures, encryption registration requests, and encryption envelopes associated with a message buffer. The occurrences are stored in sequence, with the first item at the zero position and subsequent items in consecutive positions. The `occurrence` input parameter for `tpenvelope()` indicates which item is being requested. When the value of `occurrence` is beyond the position of the last item, `tpenvelope()` fails with the `TPENOENT` error condition. A process can examine all items by calling `tpenvelope()` repeatedly until `TPENOENT` is returned.

In an originating process, digital signature and encryption information is generally in a pending state, waiting until the message is sent. In a receiving process, digital signatures have already been verified, and encryption and decryption have already been performed.

- What Happens When an Originating Process Calls tpenvelope
- What Happens When a Receiving Process Calls tpenvelope
- Understanding the Composite Signature Status
- Example Code for tpenvelope

3.10.1 What Happens When an Originating Process Calls tpenvelope

When an originating process calls `tpenvelope()` with the originating message buffer as an argument, `tpenvelope()` reports:

- Any digital signature request *explicitly* registered with the message buffer as being in the `TPSIGN_PENDING` state. The originating process explicitly registers a digital signature request by calling the ftpsign(3c)unction.
- Any digital signature request *implicitly* registered with the message buffer as also being in the `TPSIGN_PENDING` state. The originating process implicitly registers a digital signature request by calling tpkey_open(3c) with the `TPKEY_AUTOSIGN` flag specified.
- Any encryption (seal) request *explicitly* registered with the message buffer as being in the `TPSEAL_PENDING` state. The originating process explicitly registers an encryption request by calling the tpseal(3c) function.
- Any encryption (seal) request *implicitly* registered with the message buffer as also being in the `TPSEAL_PENDING` state. The originating process implicitly registers an encryption request by calling `tpkey_open()` with the `TPKEY_AUTOENCRYPT` flag specified.

In addition to the status, `tpenvelope()` returns the key handle associated with a digital signature or encryption registration request. A process can call the tpkey_getinfo(3c) function with the key handle as an argument, to get more information about the key handle.

3.10.2 What Happens When a Receiving Process Calls tpenvelope

When a process receives a message buffer, it receives *only* the message content. Any digital signatures or encryption envelopes associated with the message buffer are not included. The receiving process must call `tpenvelope()` to obtain information about any attached digital signatures or encryption envelopes.

When a receiving process calls `tpenvelope()` with the received message buffer as an argument, `tpenvelope()` reports:

- Any digital signature attached to the message buffer. A digital signature has one of the following states:

  TPSIGN_OK
  Digital signature has been verified.

  TPSIGN_TAMPERED_MESSAGE
  Digital signature is not valid because the content of the message buffer has been altered.

  TPSIGN_TAMPERED_CERT
  Digital signature is not valid because the signer's digital certificate has been altered.

  TPSIGN_REVOKED_CERT
  Digital signature is not valid because the signer's digital certificate has been revoked.

  TPSIGN_POSTDATED
  Digital signature is not valid because its timestamp is too far into the future.

  TPSIGN_EXPIRED_CERT
  Digital signature is not valid because the signer's digital certificate has expired.

  TPSIGN_EXPIRED
  Digital signature is not valid because its timestamp is too old.

  TPSIGN_UNKNOWN
  Digital signature is not valid because the signer's digital certificate was issued by an unknown Certification Authority (CA).

- Any encryption envelope attached to the message buffer. An encryption envelope has one of the following states:

  TPSEAL_OK
  Encryption envelope is valid.

  TPSEAL_TAMPERED_CERT
  Encryption envelope is not valid because the target recipient's digital certificate has been altered. (Target recipient will not receive the message buffer.)

  TPSEAL_REVOKED_CERT
  Encryption envelope is not valid because the target recipient's digital certificate has been revoked. (Target recipient will not receive the message buffer.)

  TPSEAL_EXPIRED_CERT
  Encryption envelope is not valid because the target recipient's digital certificate has expired. (Target recipient will not receive the message buffer.)

  TPSEAL_UNKNOWN
  Encryption envelope is not valid because the target recipient's digital certificate was issued by an unknown CA. (Target recipient will not receive the message buffer.)

In addition to the status, `tpenvelope()` returns the key handle associated with a digital signature or encryption envelope. A process can call the tpkey_getinfo(3c) function with the key handle as an argument, to get more information about the key handle.

If a receiving process calls `tpsign()` to register a digital signature request after receiving the message buffer, `tpenvelope()` reports the status of the registration as `TPSIGN_PENDING`. Similarly, if a receiving process calls `tpseal()` to register an encryption (seal) request after receiving the message buffer, `tpenvelope()` reports the status of the registration as `TPSEAL_PENDING`.

If a receiving process modifies the content of a *signed* message buffer after receiving it, the attached signatures are no longer valid. As a result, `tpenvelope()` cannot verify the signatures, and reports a signature status of `TPSIGN_TAMPERED_MESSAGE`.

3.10.3 Understanding the Composite Signature Status

For a message buffer with multiple digital signatures, the public key software calls an internal function equivalent to `tpenvelope()` to examine the state of each digital signature. Then, by observing certain rules, the public key software forms a *composite signature status*. The rules for forming a composite signature status are shown in the following table.

Table 3-4 Composite Signature Status

If Any Status Is . . .	And There Is No Status of . . .	Then the Composite Status Is . . .
TPSIGN_TAMPERED_MESSAGE	. . .	TPSIGN_TAMPERED_MESSAGE
TPSIGN_TAMPERED_CERT	TPSIGN_TAMPERED_MESSAGE	TPSIGN_TAMPERED_CERT
TPSIGN_REVOKED_CERT	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT	TPSIGN_REVOKED_CERT
TPSIGN_POSTDATED	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT	TPSIGN_POSTDATED
TPSIGN_EXPIRED_CERT	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED	TPSIGN_EXPIRED_CERT
TPSIGN_OK	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED TPSIGN_EXPIRED_CERT	TPSIGN_OK
TPSIGN_EXPIRED	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED TPSIGN_EXPIRED_CERTTPSIGN_OK	TPSIGN_EXPIRED
TPSIGN_UNKNOWN	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED TPSIGN_EXPIRED_CERT TPSIGN_OK TPSIGN_EXPIRED	TPSIGN_UNKNOWN

Any incoming message buffer *without* a composite signature status of `TPSIGN_OK` or `TPSIGN_UNKNOWN` is discarded as if it were never received. If the `SIGNATURE_REQUIRED` parameter is set to `Y` (yes) in the ATMI application's `UBBCONFIG` file, then any incoming message buffer *without* a composite signature status of `TPSIGN_OK` is discarded as if it were never received. See Enforcing the Signature Policy for Incoming Messages for more detail.

An exception to the handling of signed message buffers described in the previous paragraph is the tpimport(3c) function. The tpimport(3c) function delivers an incoming message buffer regardless of the composite signature status.

3.10.4 Example Code for tpenvelope

The following listing describes how to use `tpenvelope()` to examine the digital signature and encryption information associated with a message buffer.

Listing Using tpenvelope Example

```
main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
           TPKEY tu_key;
           TPKEY sdo_key;
           TPKEY output_key;
           char *sendbuf, *rcvbuf;
           int ret;
           int occurrence = 0;
           long status;
           char principal_name[PNAME_LEN];
           long pname_len = PNAME_LEN;
           int found = 0;
           .
           .
           .
          output_key = NULL;
          ret = tpenvelope(rcvbuf, 0, occurrence, &output_key,
             &status, NULL, 0);
         while (ret != -1) {
         if (status == TPSIGN_OK) {
           if (tpkey_getinfo(output_key, "PRINCIPAL",
               principal_name, &pname_len, 0) == -1) {
               (void) fprintf(stdout, "Unable to get information
                   about principal: %d(%s)\n",
                       tperrno, tpstrerror(tperrno));
               tpfree(sendbuf);
               tpfree(rcvbuf);
               tpterm();
              (void) tpkey_close(tu_key, 0);
              (void) tpkey_close(sdo_key, 0);
              (void) tpkey_close(output_key, 0);
               exit(1);
          }
          /* Do not forget to free resources */
          (void) tpkey_close(output_key, 0);
          output_key = NULL;
          found = 1;
          break;
          }
          /* Do not forget to free resources */
         (void) tpkey_close(output_key, 0);
         output_key = NULL;
         occurrence++;
         ret = tpenvelope(rcvbuf, 0, occurrence, &output_key,
             &status, NULL, 0);
```

```
}
.
.
.
}
```

> **Note:**
>
> - Externalizing Typed Message Buffers
> - Public Key Security
> - Administering Public Key Security
> - Programming an ATMI Application with Security

3.11 Externalizing Typed Message Buffers

An externalized representation is a message buffer that does not include any ATMI header information that is normally added to a message buffer just before the buffer is transmitted. An externalized representation of a signed message buffer enables “pass through” transmission of signed data and long-term storage of the signed buffer for non-repudiation. It also enables an encrypted message buffer to be transported through intermediate processes without access to a decryption key.

- How to Create an Externalized Representation
- How to Convert an Externalized Representation
- Example Code for tpexport and tpimport

3.11.1 How to Create an Externalized Representation

An ATMI process converts a typed message buffer into an externalized representation by calling the tpexport(3c) function. Pending signatures associated with a message buffer are generated at the time `tpexport()` is called, just as if the buffer were being transmitted to another process by an ATMI function. Similarly, pending seals associated with a message buffer are generated at the time `tpexport()` is called, just as if the buffer were being transmitted to another process by an ATMI communication function.

The externalized representation of a message buffer is stored in the PKCS-7 format, which is a binary format. If a string format is required, the calling process must call `tpexport()` with the `TPEX_STRING` flag specified.

> **Note:**
>
> The ability to create an externalized representation of a typed message buffer is not unique to public key security. A process may call `tpexport()` to externalize a typed message buffer regardless of whether a message buffer is marked for digital signature or encryption.

3.11.2 How to Convert an Externalized Representation

A receiving process calls the tpimport(3c) function to convert the externalized representation of a message buffer into a typed message buffer. The `tpimport()` function also performs decryption, if necessary, and verifies any associated digital signatures.

3.11.3 Example Code for tpexport and tpimport

The following listing describes how to use `tpexport()` to convert a typed message buffer into an externalized representation, and how to use `tpimport()` to convert the externalized representation back into a typed message buffer.

Listing Using tpexport and tpimport Example

```
static void hexdump _((unsigned char *, long));
         #define MAX_BUFFER 80000
         main(argc, argv)
         int argc;
         char *argv[];
         #endif
         {
             char  *databuf;
             char  exportbuf[MAX_BUFFER];
             long  exportbuf_size = 0;
             char  *importbuf = NULL;
             long  importbuf_size = 0;
             int go_on = 1;
         .
         .
         .
         exportbuf_size = 0;
         while (go_on == 1) {
         if (tpexport(databuf, 0, exportbuf, &exportbuf_size, 0)
             == -1) {
             if (tperrno == TPELIMIT) {
                 printf("%d tperrno is TPELIMIT, exportbuf_size=%ld\n",
                   __LINE__, exportbuf_size);
             if (exportbuf_size > MAX_BUFFER) {
                return(1);
             }
         }
         else {
         printf("tpexport(%d) failed: tperrno=%d(%s)\n",
         __LINE__, tperrno, tpstrerror(tperrno));
         return(1);
               }
           }
           else {
               go_on = 0;
           }
         }
         .
         .
         .
```

```
hexdump((unsigned char *) exportbuf, (long) exportbuf_size);
if (tpimport(exportbuf, exportbuf_size, &importbuf,
    &importbuf_size, 0) == -1) {
printf("tpimport(%d) failed: tperrno=%d(%s)\n",
__LINE__, tperrno, tpstrerror(tperrno));
return(1);
}
.
.
.
}
```

Note:

- Public Key Security
- Administering Public Key Security
- Programming an ATMI Application with Security

4

Quick Reference for TLS Support

The following is a quick reference for TLS support in Oracle Tuxedo.

All network connections within a Tuxedo domain support TLS. Tuxedo 22.1.0.0.0 supports TLS versions: 1.0, 1.1, and 1.2. Tuxedo 22.1.1.0.0 introduces support for TLS version 1.3 and removes support for TLS versions 1.0 and 1.1. The following information outlines the TLS support behavior in Tuxedo 22.1.1.0.0.

4.1 Overview

TLS is supported on all network connections within a Tuxedo domain. In Tuxedo 22.1.0.0.0, TLS versions 1.0, 1.1, and 1.2 are supported. Tuxedo 22.1.1.0.0 introduces support for TLS version 1.3 and removes support for TLS versions 1.0 and 1.1. The following information outlines the TLS support behavior in Tuxedo 22.1.1.0.0.

4.2 Supported Tuxedo Components

The following Tuxedo components support versions 1.2 and 1.3 of TLS and can function as either a TLS client or a TLS server.

> **Note:**
>
> Some Tuxedo components, such as `BRIDGE`, `GWTDOMAIN`, and `GWWS`, can operate as both a TLS client and TLS server.

Table 4-1 Supported Tuxedo Components

Product	TLS Client	TLS Server
Tuxedo	Workstation client	Workstation Listener/Handler
JOLT	JOLT client	JOLT Listener/Handler
Tuxedo	CORBA client	CORBA Listener/Handler
Tuxedo	BRIDGE	BRIDGE
Tuxedo	tmboot/tmshutdown	tlisten
Tuxedo	GWTDOMAIN	GWTDOMAIN
SALT	External Web service client	GWWS

Table 4-1 (Cont.) Supported Tuxedo Components

Product	TLS Client	TLS Server
SALT	GWWS	External Web service server
TASM Plus	LMS	TSAM Plus Manager

4.3 TLS Version Configuration

The following table summarizes the default TLS version for each Tuxedo component and provides configuration options for changing it.

Table 4-2 TLS Version Configurations

TLS Client			TLS Server		
Component	**Default Version**	**Configurations Option**	**Component**	**Default Version**	**Configuration Option**
Workstation client	TLSv1.3_1.2	Environment variable `WSNADDR`. For example: `WSNADDR="//host1:8810;TLSv1.3"`	Workstation Listener/Handler	TLSv1.3_1.2	Environment variable `TM_TLS_FORCE_VER`
JOLT client	TLSv1.3_1.2	Java property `bea.jolt.tls.version`. For example: `System.setProperty("bea.jolt.tls.version" "TLSv1.3");`	JOLT Listener/Handler	TLSv1.3_1.2	Environment variable `TM_TLS_FORCE_VER`
CORBA client	TLSv1.3_1.2	The `Tobj_Bootstrap` constructor `naddress` parameter or the environment variable `TOBJADDR`	CORBA Listener/Handler	TLSv1.3_1.2	Environment variable `TM_TLS_FORCE_VER`
BRIDGE	TLSv1.3_1.2	Not Supported	BRIDGE	TLSv1.3_1.2	Environment variable `TM_TLS_FORCE_VER`
tmboot/tmshutdown	TLSv1.3_1.2	Not Supported	tlisten	TLSv1.3_1.2	Environment variable `TM_TLS_FORCE_VER`
GWTDOMAIN	TLSv1.3_1.2	Configuration option `DM_TDOMAIN/TLSversion` in `DMCONFIG` file. For example:`TLSversion=TLSv1.3`	GWTDOMAIN	TLSv1.3_1.2	Environment variable `TM_TLS_FORCE_VER`
External Web service client	N/A	N/A	GWWS	TLSv1.3_1.2	Environment variable `TM_TLS_FORCE_VER`
GWWS	TLSv1.3_1.2	Attribute `tlsversion` in element `Endpoint` in `SALTDEPLOY` file. For example:`<Endpoint address="..." tlsversion="TLSv1.3"/>`	External Web service server	N/A	N/A

Table 4-2 (Cont.) TLS Version Configurations

TLS Client			TLS Server		
Component	**Default Version**	**Configurations Option**	**Component**	**Default Version**	**Configuration Option**
LMS	TLSv1.3_1.2	Not Supported	TSAM Plus Manager	See WebLogic Server document	See WebLogic Server document

> **Note:**
>
> Configuration options such as environment variable `TM_TLS_FORCE_VER` supports the following values. Any other values are ignored, and the default TLS version will be used.
>
> - TLSv1.2
> - TLSv1.3
> - TLSv1.3_1.2

4.4 Supported Cipher Suites

TLS 1.3 supports the following cipher suites:

- `TLS_AES_128_GCM_SHA256`
- `TLS_AES_256_GCM_SHA384`
- `TLS_CHACHA20_POLY1305_SHA256`
- `TLS_AES_128_CCM_SHA256`
- `TLS_AES_128_CCM_8_SHA256`

TLS 1.2 supports the following cipher suites:

- `TLS_RSA_WITH_AES_256_CBC_SHA256`
- `TLS_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`
- `TLS_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`

JDK determines the default cipher suites for JOLT client. The Java property `bea.jolt.tls.ciphersuites` can be used to customize the cipher suites used by JOLT client. Customize the cipher suites for other Tuxedo components using the environment variable `TM_CIPHERSUITES`.

4.5 Upgrade from Previous Versions to TLS 1.3

When upgrading to TLS 1.3 from previous versions, Oracle recommends the following these steps:

- Certain old key generation algorithms are no longer supported in the TLS 1.3. If you require to regenerate TLS keys and certificates.
- TLS 1.3 has removed support for re-negotiation, so any re-negotiation configurations will be ignored in TLS 1.3.
- JOLT client depends on JDK to support TLS 1.3. For example, for JDK 8, 8u261 or later must be used.

4.6 Interoperability

The Tuxedo 22.1.1.0.0 release no longer supports TLS 1.1 and older versions. Now, if TLS is enabled, this Tuxedo release cannot interoperate with previous Tuxedo releases that do not support TLS 1.2.

5

Implementing Single Point Security Administration

The following sections explain how to implement single point security administration for Tuxedo and WebLogic Server from the Tuxedo point of view:

> **Note:**
>
> Before setting up single point security, ensure you are familiar with the Tuxedo security architecture and requirements. You may also want to coordinate this effort with your WebLogic or LDAP Administrator.

5.1 What Single Point Security Administration Means

If you have both Tuxedo and WebLogic Server deployed in your environment, then you have to manage two sets of security information. Single point security administration allows you to leverage the WebLogic Server security to manage your security database by eliminating user and group information from Tuxedo. You can use WebLogic Server as your security database to authenticate Tuxedo users.

> **Note:**
>
> The Tuxedo ACL information will continue to reside in Tuxedo and is not currently integrated with WebLogic Server 7.0.

If you are specifying `SECURITY=ACL` or `SECURITY=MANDATORY_ACL` in the `RESOURCES` section of the `UBBCONFIG` file, then you must continue to maintain *tpgrp* and *tpacl* files in Tuxedo.

The single point security administration feature leverages the enhanced WebLogic Server 7.0 security and the LDAP to allow single point security administration. You can maintain user security information in WebLogic Server embedded LDAP server and use the WebLogic Server Console to administer the security information from a single system. You must modify the UBBCONFIG file to enable single point security.

5.1.1 Single Point Security Administration Tasks

To set up single point security, you must provide the Tuxedo security information to the WebLogic Server-embedded LDAP server. This includes migrating or setting up the Tuxedo user (UID) and group (GID) information in WebLogic Server LDAP server so that authentication can be successful. For Tuxedo UID and GID values to be available to WebLogic Server, you must use the `tpmigldap` utility, modify the *tpusr* file manually with a text editor, or enter the user information via the WebLogic Administration Console.

> **Note:**
>
> The WebLogic Administration Console may be the method used when adding one or two users after the security database is set up. For efficiency and time management, you may prefer using the `tpmigldap` utility or the *tpusr* file as a general rule.

Single point security administration consists of the following tasks:

- Setting up LAUTHSVR as the Authentication Server
- Using tpmigldap to Migrate User Information to WebLogic Server
- Setting up GAUTHSVR as the Authentication Server
- Using tpmigldif to Migrate User Information
- Adding New Tuxedo User Information

> **See Also:**
>
> - Security information for Oracle WebLogic Server

5.2 Setting up LAUTHSVR as the Authentication Server

`LAUTHSVR` is a System /T provided server that offers the authentication service while the user security information is located in WebLogic Server. To enable the single security administration feature, you must configure `LAUTHSVR` as the authentication server. At runtime, the `LAUTHSVR` will retrieve the user information from the WebLogic Server-embedded LDAP and authenticate users. If the authentication is successful, an *appkey* is returned to the user, otherwise, authentication fails.

> **Note:**
>
> Tuxedo 10 and greater allows you to configure WebLogic authentication using a more general authentication server, `GAUTHSVR` (which can be used along with `LAUTHSVR` or replace it).

For more GAUTHSVR information, see Setting up GAUTHSVR as the Authentication Server and GAUTHSVR(5), in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

To define LAUTHSVR as the authentication server, you must define the following parameters in the UBBCONFIG file:

- SECURITY must be set to USER_AUTH, ACL, or MANDATORY_ACL in the RESOURCES section.
- LAUTHSVR must be specified in the SERVERS section.
- If LAUTHSVR cannot find a valid configuration file or the file does not exist, it will log an error message in USERLOG and fail to boot. The default LAUTHSVR configuration file is $TUXDIR/udataobj/tpldap and is provided with the product.

5.2.1 LAUTHSVR Command Line Interface

The LAUTHSVR is the LDAP-based authentication server for Tuxedo. It requires a configuration file, that by default is $TUXDIR/udataobj/tpldap. You can create your own LAUTHSVR configuration file or use the default tpldap file that is available with the product.

The command line interface syntax for LAUTHSVR is as follows:

-f full_pathname
Specifies the full pathname of the LAUTHSVR configuration file.

> **Note:**
>
> If -f option is omitted, the default LAUTHSVR configuration file tpldap is used.

The following example instructs LAUTHSVR to use the default configuration file, *tpldap*, in the $TUXDIR/udataobj directory.

```
LAUTHSVR SRVGRP=GROUP1 SRVID=2 CLOPT="-A-"
```

In the following example, LAUTHSVR uses the myauthsvr.conf configuration file in the /home/tuxedo/bankapp directory.

```
LAUTHSVR SRVGRP=GROUP1 SRVID=2
         CLOPT="-A--
         -f/home/tuxedo/bankapp/myauthsvr.conf"
```

5.2.2 Setting Up the LAUTHSVR Configuration File

LAUTHSVR supports an input configuration file that contains information such as bind DN and an unencrypted password for bind DN. This configuration file is a plain text file and can be edited using any text editor and must be protected by the system using file permissions. By default the configuration file, named *tpldap*, is located in $TUXDIR/udataobj directory. You can overwrite this file in the command line for LAUTHSVR. The LAUTHSVR configuration file contains keyword and value pairs as defined in the following table.

- Syntax Requirements for LAUTHSVR Configuration File
- LAUTHSVR Configuration File Keywords
- Example LAUTHSVR Configuration File

5.2.2.1 Syntax Requirements for LAUTHSVR Configuration File

Although the default values for the LAUTHSVR configuration file are usually sufficient, a system administrator may choose to configure it with different names. Therefore, you should be aware of the following requirements for the LAUTHSVR configuration file:

- The LAUTHSVR configuration file is a plain text file.
- Keyword order does not matter; however, there must be at least one space character between the keyword and its value.
- Comments begin with the pound symbol (#). Text after the # is ignored.
- The upper limit of a line is 255 characters. If a line exceeds this upper limit, it will be truncated.
- The bind DN must have privileges to access the LDAP database (usually this is the LDAP administrator).

> **Note:**
>
> Before an administrator can set up and use the Tuxedo LDAP-based security authentication server, the administrator must change the LDAP administrator password through the WebLogic Administration Console.

5.2.2.2 LAUTHSVR Configuration File Keywords

The following table defines the LAUTHSVR configuration file keywords.

> **Note:**
>
> The only required keyword in the LAUTHSVR configuration file is PASSWORD, which specifies the password for bind DN. All other keywords are optional.

Table 5-1 LAUTHSVR Configuration File Keywords

Keyword	Value Type	Usage
FILE_VERSION	numeric	The configuration file version. This should always be 1. The default is 1.
LDAP_VERSION	numeric	The LDAP protocol version. Valid values are 2 or 3. The default is 3.
BINDDN	string	The DN used to bind to an LDAP server, usually the DN for the LDAP administrator. The default is "cn=admin".
BASE	string	LDAP search base. The default is "ou=people,ou=myrealm, dc=mydomain", where my realm is the name of the security realm and my domain is the name of the WebLogic Server domain.
UID	string	The user id attribute that is used to logon to WebLogic Server and Tuxedo. The default is uid.
PASSWORD	string	The password for bind DN. This is a required keyword and the password is in clear text format or encrypted format. The tpldapconf command can be used to create the encrypted password.
LDAP_ADDR	string	A comma separated list of WebLogic hostnames and ports. The syntax is [//]hostname[:port][, [//]hostname[:port]...]. The default value for port is 7001. If LDAP_ADDR is not specified, LAUTHSVR assumes localhost:7001 is the location to contact the LDAP server. For more information about specifying multiple network addresses, refer to"Using Multiple Network Addresses for High Availability."
EXPIRE	numeric	A numeric value that represents the number of seconds the cached entry is available in the local process memory. A value other than zero will enable caching. A value of zero specifies no caching. The default is zero. For more information about enabling caching,refer to"Using Multiple Network Addresses for High Availability."
SRCH_ORDER	string	Valid values are LDAP or LOCAL, or both separated by a comma. If you specify LOCAL, the search order will use the tpusr file. The default is LDAP. For more information about database search order, refer to"Configuring the Database Search Order."

Table 5-1 (Cont.) LAUTHSVR Configuration File Keywords

Keyword	Value Type	Usage
LOCAL_FILE	string	The full pathname of the tpusr file to be used if LOCAL search order is enabled. The default value is $APPDIR/tpusr. For more information about database search order, refer to“ Configuring the Database Search Order..” **Note:** If a directory path is specified other than the default $APPDIR/*tpusr*, the file must be generated using Tuxedo MIB or tpusradd command line utility. Failure to do this may cause authentication failure.
WLS_DOMAIN	string	The WebLogic Server domain name. The default value is *mydomain*.
WLS_REALM	string	The WebLogic Server security realm name. The default is *myrealm*.
ADM_GROUP	string	The WebLogic Server administrator group name. The default is Administrators.
OP_GROUP	string	The WebLogic Server operators group name. The default is Operators.
TUX_UID_KW	string	The keyword used in the description to identify the Tuxedo userid. The default is TUXEDO_UID.
TUX_GID_KW	string	The keyword used in the description to identify the Tuxedo group ID. The default is TUXEDO_GID.

5.2.2.3 Example LAUTHSVR Configuration File

The following listing describes an example of a `LAUTHSVR` configuration file.

Listing Example LAUTHSVR Configuration File

```
#
        # Tuxedo LDAP Authentication Server configuration file.
        #
        # created: Thu May 26 15:36:59 2002
        #
        FILE_VERSION         1
        LDAP_VERSION         3
        BINDDN               cn=Admin
        BASE                 ou=people,ou=myrealm,dc=mydomain
        UID                  uid
        PASSWORD             secret
        LDAP_ADDR           //PLUTO:7001,//Saturn:7001
        EXPIRE               0
        SRCH_ORDER           LDAP
        WLS_DOMAIN           mydomain
        WLS_REALM            myrealm
        ADM_GROUP            Administrators
        OP_GROUP             Operators
        TUX_UID_KW           TUXEDO_UID
        TUX_GID_KW           TUXEDO_GID
        # end of file
```

> **WARNING:**
>
> It is recommended that the system administrator secures this file with the correct access permissions, as the `PASSWORD` for the LDAP administrator is in clear text.

5.2.3 Example UBBCONFIG Using LAUTHSVR

The following listing describes an example UBBCONFIG file with `SECURITY` set to `ACL` and `LAUTHSVR` defined.

Listing Example UBBCONFIG File Using LAUTHSVR

```
*RESOURCES

        IPCKEY        51002
        MASTER        site1
        MAXACCESSERS  50
        MAXSERVERS    20
        MAXSERVICES   20
        MODEL        SHM
        LDBAL        N
        BLOCKTIME     10
        SECURITY      ACL
        AUTHSVC       "..AUTHSVC"
```

```
*MACHINES
DEFAULT:
        APPDIR="/home/tuxedo/application"
        TUXCONFIG="/home/tuxedo/application/TUXCONFIG"
        TUXDIR="/home/tuxedo/tux81"
Server1         LMID=site1
                     MAXWSCLIENTS=20

*GROUPS
GROUP1         LMID=site1 GRPNO=1
GROUP2         LMID=site1 GRPNO=2
GROUP3         LMID=site1 GRPNO=3
GROUP4         LMID=site1 GRPNO=4

*SERVERS
DEFAULT:
        CLOPT="-A" RESTART=N  MAXGEN=5
LAUTHSVR        SRVGRP=GROUP1 SRVID=10
CLOPT="-A -- -F /home/tuxedo/application/lauthsvr.conf "
DMADM          SRVGRP=GROUP2  SRVID=20
GWADM          SRVGRP=GROUP3  SRVID=30
GWTDOMAIN       SRVGRP=GROUP3 SRVID=31
Simpserv        SRVGRP=GROUP4 SRVID=40

*SERVICES
TOUPPER
```

5.2.4 Using Multiple Network Addresses for High Availability

It is possible to configure more than one network address for a WebLogic Server domain. This may be a favorable configuration in order to provide high availability for user authentication. The user security information is replicated to all WebLogic Server-embedded LDAP servers in a WebLogic domain. `LAUTHSVR` can only connect to one server at a time; however, when a network error occurs, `LAUTHSVR` will try to connect to the next available address.

To configure multiple network addresses for `LAUTHSVR`, use the `LDAP_ADDR` keyword in the `LAUTHSVR` configuration file. The order in which the hostnames are specified is the order in which `LAUTHSVR` will try to connect. To use caching during authentication, specify the `EXPIRE` keyword. The value in this keyword will determine the number of seconds the cached entry is available in the local process memory.

> **Note:**
>
> It is not required to have WebLogic Server available when you boot Tuxedo using `tmboot`; however, without the availability of at least one WebLogic Server, `LAUTHSVR`s ability to authenticate users is limited.

Without the availability of WebLogic Server, you can boot Tuxedo and authenticate users using `SRCH_ORDER LOCAL`. In this case, the user authentication is verified against the *tpusr* file. For more information about search order, refer to Configuring the Database Search Order.

- Example LAUTHSVR Configuration of Multiple Network Addresses

5.2.4.1 Example LAUTHSVR Configuration of Multiple Network Addresses

The following example specifies multiple network addresses in the LDAP_ADDR keyword.

```
LDAP_ADDR //Pluto:8000,//Saturn,Jupiter
```

The previous example specifies three WebLogic Server hostnames. The first server runs on Pluto and uses address 8000. The second server runs on Saturn and uses the default address 7001. The third server runs on Jupiter and also uses the default address 7001.

5.2.5 Configuring the Database Search Order

By default the LAUTHSVR authentication server will search the user information in the WebLogic Server-embedded LDAP server. To enable the use of the *tpusr* file in the database search, you must specify LOCAL in the SRCH_ORDER keyword. The order that the comma separated values are defined in the SRCH_ORDER keyword will specify the order in which LAUTHSVR searches for user information. LAUTHSVR will search the LDAP server or the *tpusr* file or both (according to the order of the values specified).

If there are two or more SRCH_ORDER entries specified in the LAUTHSVR configuration file, only the last entry takes effect. In this case a warning message is logged in USERLOG as well. A warning message also results if you specify a value other than LDAP or LOCAL in the SRCH_ORDER keyword. In this case, the invalid entry is discarded and the default value or a previous valid SRCH_ORDER entry is used.

- Example LAUTHSVR Configuration for Database Search Order

5.2.5.1 Example LAUTHSVR Configuration for Database Search Order

The following example specifies that LAUTHSVR should search the WebLogic Server-embedded LDAP server first for user information. If the user information is not found in the LDAP server, then LAUTHSVR should look in the *tpusr* file.

```
SRCH_ORDER LDAP,LOCAL
```

The following example specifies that LAUTHSVR should search the *tpusr* file first for user information. If the user information is not found in the tpusr file, then LAUTHSVR should look in the WebLogic Server-embedded LDAP server for the information.

```
SRCH_ORDER LOCAL,LDAP
```

The following example specifies that LAUTHSVR should search the *tpusr* file only for user information.

```
SRCH_ORDER LOCAL
```

> **Note:**
>
> - LAUTHSVR(5) and GAUTHSVR(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

5.2.6 Using tpmigldap to Migrate User Information to WebLogic Server

You must use the `tpmigldap` command utility to migrate Tuxedo user and group information to WebLogic Server.

- Assigning New Passwords for the tpusr File
- tpmigldap Command Line Options

5.2.6.1 Assigning New Passwords for the tpusr File

Before migrating the user and group information, the administrator must assign new passwords for each user so the migration can be successful. This step is required because the passwords in the tpusr file are encrypted with one-way encryption; therefore, it is impossible to retrieve the original password from the file.

There are two ways to handle this password situation:

- Modify the `tpusr` file.
  You can modify the `tpusr` file using a text editor and change the user password for each user in the file. The password field is the second field in the `tpusr` file. The field delimiter is a colon (:). Each user takes up a line in the `tpusr` file.

  The following example:

  ```
  TuxedoUser1:ADdg0w8nfGMag:6001:601:TPCLTNM,*:
  TuxedoUser2:0Yq2s6FjbvuU2:6002:601:TPCLTNM,*::
  ```

  could be modified to:

  ```
  TuxedoUser1:User1Password:6001:601:TPCLTNM,*::
  TuxedoUser2:User2Password:6002:601:TPCLTNM,*::
  ```

- Use the `-f` option with the `tpmigldap` utility to define a default password for all users.
  If a -f option is used, then the argument that follows will be used as a substitute for the password field in the `tpusr` file for every user in the file.

  The following example command:

  ```
  tpmigldap -f userpassword -c
  ```

  causes “userpassword” to be assigned to every user in the `tpusr` file. After the migration, all users can use “userpassword” as their password in order to join the Tuxedo application.

5.2.6.2 tpmigldap Command Line Options

The following table describes the command line options for the `tpmigldap` utility. The order of the command line options does not matter.

> **Note:**
>
> The `tpmigldap` command requires the use of `-w` or `-c` so the user or group can be added to the WebLogic Server-embedded LDAP database.

Table 5-2 tpmigldap Command Line Options

Command Line Option	Option Argument	Default Value	Usage
`-h`	hostname	localhost	Hostname of WebLogic Server.
`-p`	port	7001	Port number for WebLogic Server Administration Console
`-d`	domain	mydomain	WebLogic Server domain name.
`-r`	realm	myrealm	WebLogic Server security realm name.
`-i`	TUXEDO_UID keyword string	TUXEDO_GID	The keyword string for Tuxedo UID that the administrator wants to use in the WebLogic Server user “description” attribute.
`-e`	TUXEDO_GID keyword string	TUXEDO_GID	The keyword string for Tuxedo GID that the administrator wants to use in the WebLogic Server user “description”.
`-f`	user password	No default.	The default user password for every user in the tpusr file.
`-b`	binddn	cn=Admin	LDAP connection bind DN.
`-w`	password	No default.	The password for bind DN.
`-c`	Not applicable.	No default.	A prompt for entering a password for bind DN.
`-u`	full path name	`$APPDIR/tpusr`	The full directory path for the *tpusr* file.
`-g`	full path name	`$APPDIR/tpgrp`	The full directory path for the *tpgrp* file.

> **See Also:**
>
> - tpmigldap(1) in the *Oracle Tuxedo Command Reference*.

5.2.7 Adding New Tuxedo User Information

There are two methods for adding new user and group information to the single security LDAP database:

- Add new information to the *tpusr* text file and then specify the updated file when using the migration utility `tpmigldap`. Refer to Adding New User Information in tpusr or tpgrp.

- Use the WebLogic Administration Console to add user or group information. Refer to Adding New User Information Using the WebLogic Administration Console.
- Using the WebLogic Administration Console may not be efficient for adding large numbers of users to the LDAP database. In the case of adding several users, you may want to use the `tpmigldap` utility.
- Adding New User Information in tpusr or tpgrp
- Adding New User Information Using the WebLogic Administration Console

5.2.7.1 Adding New User Information in tpusr or tpgrp

To add new user information to the single point security LDAP database:

1. Use your existing *tpusr* file and *tpgrp* file to add the new user and group information. Be sure to use the same format previously defined in the file. Be sure to use clear text passwords to add to the LDAP database.
2. Run the `tpmigldap` utility using the `-u` option and specify the updated *tpusr* file and the`-g` option and specify the updated *tpgrp* file. For example:

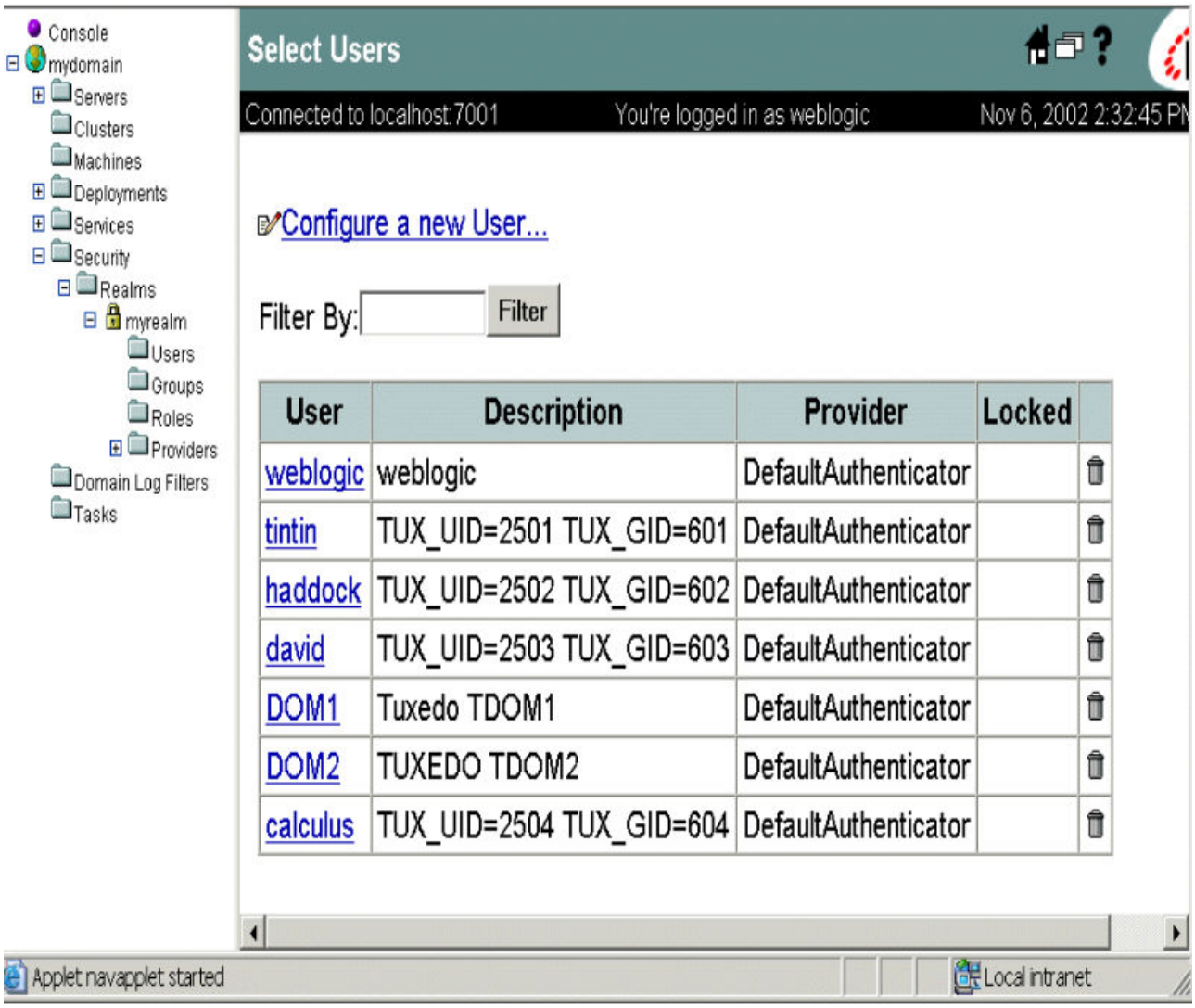
```
tpmigldap -u $APPDIR/tpusr -g $APPDIR/tpgrp
```

5.2.7.2 Adding New User Information Using the WebLogic Administration Console

To add new user information to the single point security LDAP database using the WebLogic Administration Consol

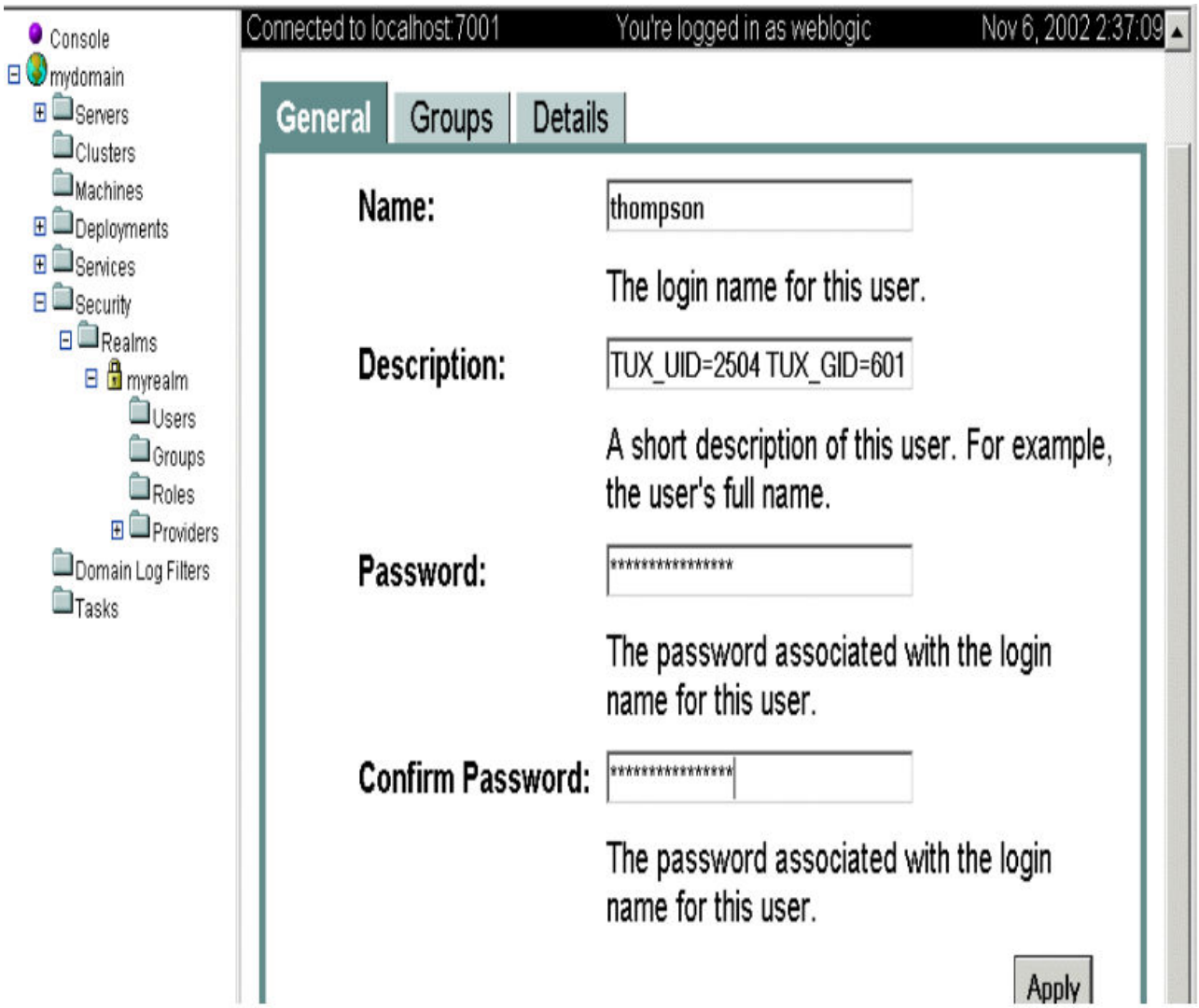
1. Access the WebLogic Administration Console and select `Security` → `Realms` → *myrealm* where *myrealm* represents the LDAP security realm.

Figure 5-1 WebLogic Administration Console Select Users



2. Click Configure a new User... and access the General tab.

Figure 5-2 WebLogic Administration Console Create Users



Enter the user information:

In the Name field specifies the user name.

In the Description field specify the Tuxedo UID and GID values as a string in the following syntax:

```
<TUXEDO UID
         KEYWORD>=<decimal value>
        <TUXEDO GID KEYWORD>=<decimal value>
```

where by default, the `TUXEDO UID KEYWORD` is `TUXEDO_UID` and `TUXEDO GID KEYWORD` by default is `TUXEDO_GID`. For example:`TUXEDO_UID=2504 TUXEDO_GID=601`

In the Password field, specify the password for the user. Then confirm the password by entering the password again in the Confirm Password field.

3. Click Apply to update the LDAP database with the new user information.

5.3 Setting up GAUTHSVR as the Authentication Server

GAUTHSVR is a System /T provided server usage is similar to LAUTHSVR, but with the following differences:

- GAUTHSVR can access user security information located in a wide variety of LDAP servers (for example, WebLogic, OpenLDAP, Netscape/IPlanet, Microsoft Active Directory, z/OS LDAP, and so on), using LDAP (Lightweight Directory Access Protocol).

  > **Note:**
  >
  > You can also configure WebLogic authentication using LAUTHSVR. GAUTHSVR can be used along with an existing LAUTHSVR or replace it.

  For more LAUTHSVR information, see Setting up LAUTHSVR as the Authentication Server and "LAUTHSVR(5)," in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

- GAUTHSVR syntax does not support multiple network addresses for high availability. For more information, see Using Multiple Network Addresses for High Availability.
- GAUTHSVR does not support user security information stored in a local file. For more information, see Configuring the Database Search Order.
  To enable the single security administration feature, GAUTHSVR must be configured as the authentication server. GAUTHSVR authenticates user security information against LDAP server. It returns *appkey* if SECURITY is set to ACL or MANDATORY_ACL when authentication success.

  To configure GAUTHSVR as the authentication server, you must define the following parameters in the UBBCONFIG file:

- SECURITY must be set to USER_AUTH, ACL, or MANDATORY_ACL in the RESOURCES section.
- GAUTHSVR must be specified in the SERVERS section.

> **Note:**
>
> If GAUTHSVR cannot find a valid configuration file or the file does not exist, it will log an error message in USERLOG and fail to boot. The default GAUTHSVR configuration file is $TUXDIR/udataobj/tpgauth and is provided with the product.

If you use GAUTHSVR with JDK1.6 or later, please specify the following: JAVA_OPTS=-Djavax.xml.stream.XMLInputFactory=com.bea.xml.stream.MXParserFactory in your environment before booting GAUTHSVR.

If you use GAUTHSVR with Java 1.6, please add the 64-bit JRE library path to LIBPATH. The default library is located in /usr/java6_64/jre/lib/ppc64. Run export LIBPATH=/usr/java6_64/jre/lib/ppc64:$LIBPATH to set the correct LIBPATH.

- GAUTHSVR Command Line Interface
- Setting Up the GAUTHSVR Configuration File

- Example UBBCONFIG Using GAUTHSVR
- Using tpmigldif to Migrate User Information
- Supported LDAP Server Template Example

5.3.1 GAUTHSVR Command Line Interface

GAUTHSVR is an LDAP-based authentication server for Tuxedo. It requires a configuration file, that by default is $TUXDIR/udataobj/tpgauth.

The command line interface syntax for GAUTHSVR is as follows:

-f *config*
Specifies the full pathname of the GAUTHSVR configuration file.

-o *gaconfig.xml*
Specifies the full pathname of the GAUTHSVR internal configuration file generated from customer configuration file specified by -f option. The default value is $APPDIR/gaconfig.xml.

-k *gakey.dat*
Specifies the full pathname of the GAUTHSVR internal configuration file generated from the configuration file (specified in the -f option). The default value is $APPDIR/gakey.dat.

-v
Verbose mode. Logs more detailed messages to ULOG.

The following example instructs GAUTHSVR to use the default configuration file, *tpgauth*, in the $TUXDIR/udataobj/tpgauth directory.

```
GAUTHSVR SRVGRP=GROUP1 SRVID=2 CLOPT="-A
         --"
```

In the following example, GAUTHSVR use the myauthsvr.conf configuration file in the /home/tuxedo/bankapp directory.

```
GAUTHSVR SRVGRP=GROUP1 SRVID=2 CLOPT="-A --
            -f/home/tuxedo/bankapp/myauthsvr.conf"
```

GAUTHSVR updates the generated XML file if tpgauth is newer than the generated XML and key files. Only changed or newly added tpgauth items are updated in the generated XML file.

If the XML and key file are not present when GAUTHSVR is booted, GAUTHSVR creates them automatically.

5.3.2 Setting Up the GAUTHSVR Configuration File

GAUTHSVR supports an input configuration file that contains information such as bind DN and an unencrypted password for bind DN. This configuration file is a plain text file and can be edited using any text editor and must be protected by the system using file permissions. By default the configuration file, named *tpgauth*, is located in $TUXDIR/udataobj/tpgauth directory. You can overwrite this file in the command line for GAUTHSVR. The following table lists keywords and value pairs contained in the GAUTHSVR configuration file.

- Syntax Requirements for GAUTHSVR Configuration File

5.3.2.1 Syntax Requirements for GAUTHSVR Configuration File

Although the default values for the GAUTHSVR configuration file are usually sufficient, you can choose to configure it with different names. Therefore, you must be aware of the following requirements for the GAUTHSVR configuration file:

- The GAUTHSVR configuration file is a plain text file.
- Keywords are case-sensitive, but their order does not matter. The keyword format is "keyword=value".
- Blank lines or lines starting with a # sign are treated as comments, and are ignored.
- The upper limit of a line is 255 characters. If a line exceeds this upper limit, it is truncated.
- The Principal must have privileges to access the LDAP database (usually the LDAP administrator).

5.3.2.2 GAUTHSVR Configuration File Keywords

GAUTHSVR keywords are divided into three groups: basic, advanced, and LDAP schema. The following tables describe the GAUTHSVR configuration file keywords accordingly.

Table 5-3 Basic GAUTHSVR Configuration File Keywords

Configuration Keyword	Value Type	Description
UserCacheExpire	numeric	A numeric value that represents the number of seconds the cached entry is available in the local process memory. A value other than zero will enable caching. A value of zero specifies no caching. The default value is 0.
UserCacheSize	numeric	Maximum number of entries for user cache where one entry is required for each user. A0 value of zero specifies no limit. The default value is 0 (indicating no limit).
SYSADM	string	The user name for the Tuxedo SYSADM.
SYSOP	string	The user name for the Tuxedo SYSOP.
Host	string	The host name or IP address of the LDAP server. The default value is localhost.
Port	numeric	The port number on which the LDAP server is listening. The default value is 389.
Principal		The Distinguished Name (DN) of the LDAP user that is used to connect to the LDAP server.

Table 5-3 (Cont.) Basic GAUTHSVR Configuration File Keywords

Configuration Keyword	Value Type	Description
Credential		The credential (generally a password) used to authenticate the LDAP user that is defined in the Principal attribute. The credential can be in clear text format or encrypted format. The tpldapconf command can be used to create the encrypted credential.
RetrieveUIDAndGID	boolean	Specifies whether the UID and GID information are retrieved from the LDAP server. It must be set to true when SECURITY is ACL or MANDATORY_ACL. The default value is false.

Table 5-4 Advanced GAUTHSVR Configuration File Keywords

Configuration Keyword	Value Type	Description
TuxedoUIDKey	string	Used to identify the Tuxedo UID. The default value is TUXEDO_UID.
TuxedoGIDKey	string	Used to identify the Tuxedo GID. The default value is TUXEDO_GID.
ConnectTimeout	numeric	The maximum number of seconds to wait for the LDAP connection to be established. If set to 0, there is no maximum time limit. The default value is 0.
ConnectionRetryLimit	numeric	The number of times to attempt to connect to the LDAP server if the initial connection failed. The default value is 1.
ResultsTimeLimit	numeric	The maximum number of milliseconds to wait for results before timing out. If set to 0, there is no maximum time limit. The default value is 0.
SSLEnabled	boolean	Specifies that TLS is used to connect to the LDAP server. The default value is false.
KeepAliveEnabled	boolean	Specifies whether to prevent LDAP connections from timing out or not. The default value is false.

Table 5-4 (Cont.) Advanced GAUTHSVR Configuration File Keywords

Configuration Keyword	Value Type	Description
`ParallelConnectDelay`	`numeric`	The number of seconds to delay when making concurrent attempts to connect to multiple servers. If set to 0, connection attempts are serialized. An attempt is made to connect to the first server in the list. The next entry in the list is tried only if the attempt to connect to the current host fails. This might cause your application to block for unacceptably long time if a host is down. If set to greater than 0, another connection setup thread is started after this number of delay seconds has passed. The default value is `0`.
`FollowReferrals`	`boolean`	Specifies whether referrals are automatically followed within the LDAP Directory or not. If set to false, a referral exception is sent when referrals are encountered during LDAP requests. The default value is `true`.
`BindAnonymouslyOnReferrals`	`boolean`	Specifies to anonymously bind when following referrals within the LDAP directory. If set to false, then the current Principal and Credential are used. The default value is `false`.
`UseZOSRACF`	`boolean`	Specifies whether the LDAP server is z/OS RACF LDAP server. The default value is `false`.

Table 5-4 (Cont.) Advanced GAUTHSVR Configuration File Keywords

Configuration Keyword	Value Type	Description
ControlFlag	string	Specifies how Tuxedo LDAP Authentication provider fits into the login sequence. The Control Flag determines how the login sequence uses the Authentication provider. A REQUIRED value specifies this LoginModule must succeed. Even if it fails, authentication proceeds down the list of LoginModules for the configured Authentication providers. This setting is the default. A REQUISITE value specifies this LoginModule must succeed. If other Authentication providers are configured and this LoginModule succeeds, authentication proceeds down the list of LoginModules. Otherwise, control is return to the application. A SUFFICIENT value specifies this LoginModule need not succeed. If it does succeed, return control to the application. If it fails and other Authentication providers are configured, authentication proceeds down the LoginModule list. An OPTIONAL value specifies this LoginModule need not succeed. Whether it succeeds or fails, authentication proceeds down the LoginModule list. The default value is REQUIRED.

Table 5-5 LDAP Schema Configuration File Keywords

Configuration Keyword	Value Type	Description
UserObjectClass	string	The LDAP object class that stores users . The default is person.
UserBaseDN	string	The base distinguished name (DN) of the tree in the LDAP directory that contains users. The default value is ou=people, o=example.com
UserFromNameFilter	string	An LDAP search filter for finding a user given the name of the user. The default value is (&(cn=%u) (objectclass=person))
UserSearchScope	string	Specifies how deep in the LDAP directory tree to search for users. Valid values are "subtree, onelevel". The default value is subtree.

Table 5-5 (Cont.) LDAP Schema Configuration File Keywords

Configuration Keyword	Value Type	Description
`UserUIDAttrName`	string	The attribute name of an LDAP user object that specifies the UID of the user or the UID and GID of the user in a fixed format. The default value is `userid`.
`UIDAttrValueType`	string	Specifies the value type of the uid attribute for the LDAP user object. Legal values include UID, and UIDAndGID. The default value is UID. **Note:** When `SECURITY` is `ACL` or `MANDATORY_ACL`, it must be set to UIDAndGID.
`UserGroupAttrNames`	string	The attribute names of an LDAP user object that specify the groups the user belongs to. This attribute can contain three types of values: GID, group CN and group DN. One type of value for each configuration. More names are separated by comma. The default value is `usergroups`.
`GroupAttrValueType`	string	Specifies the value type of the group attributes for the LDAP user object. Legal values include "GID, group CN, and group DN". The default value is `GID`.
`GroupBaseDN`	string	The base distinguished name (DN) of the tree in the LDAP directory that contains groups. The default value is `ou=groups, o=example.com`.
`GroupFromNameFilter`	string	An LDAP search filter for finding a group given the name of the group. The default value is `(&(cn=%g)(objectclass=groupofuniquenames))`.
`StaticGroupObjectClass`	string	The name of the LDAP object class that stores static groups The default value is `groupofuniquenames`.
`GroupSearchScope`	string	Specifies how deep in the LDAP directory tree to search for groups. Valid values are "subtree, onelevel" The default value is `subtree`.

Table 5-5 (Cont.) LDAP Schema Configuration File Keywords

Configuration Keyword	Value Type	Description
GroupGIDAttrName	string	The attribute of a LDAP group object that specifies the GID of the group The default value is groupid.

5.3.2.3 Example GAUTHSVR Configuration File

The following Listing shows a GAUTHSVR configuration file for WebLogic Server example. Please refer to this example when configuring other LDAP servers.

Listing 4-3 Example WebLogic GAUTHSVR Configuration File

```
#
         # Tuxedo LDAP Authentication Server configuration file.
         #
         # created: Thu May 26 15:36:59 2002
         # end of file


# Tuxedo configuration
         UserCacheExpire = 600
         UserCacheSize = 16384
         SYSADM = sysadm
         SYSOP = sysop

        # LDAP server configuration
         Host = server.bea.com
         Port = 7001
         Principal = cn=Admin
         Credential= weblogic

         UserObjectClass = person
         UserBaseDN = ou=people,ou=myrealm,dc=examples
         UserFromNameFilter = (&(uid=%u)(objectclass=person))
         UserUIDAttrName = description
         UserGroupAttrNames=wlsMemberOf
         RetrieveUIDAndGID = true
         UIDAttrValueType = UIDAndGID
```

> **Note:**
>
> Ensure that the UID =* and GID = * in the LDAP description are the same as defined in SECURITY IS ACL.

> **WARNING:**
>
> It is recommended that the system administrator secures this file with the correct access permissions, as the PASSWORD for the LDAP administrator is in clear text.

5.3.3 Example UBBCONFIG Using GAUTHSVR

The following listing describes an example `UBBCONFIG` file with `SECURITY` set to `ACL` and `GAUTHSVR` defined.

Listing Example UBBCONFIG File Using GAUTHSVR

```
# UBBCONFIG
          *SERVER
           GAUTHSVR SVRGRP="SYSGRP" SVRID=100
                         CLOPT="-A -- -f ${APPDIR}/tpgauth"
                         ENVFILE="${APPDIR}/tpgauth.env"
```

> **See Also:**
>
> - GAUTHSVR(5) and LAUTHSVR(5) in the *Oracle Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

5.3.4 Using tpmigldif to Migrate User Information

You can use the `tpmigldif` command utility to migrate Tuxedo user and group information to LDAP servers in LDAP Interchange Format (LDIF). In order to use `tpmigldif`, you must create a migration template.

- Using tpmigldif Command Line Options
- tpusr and tpgrp File Format
- Creating a Migration Template

5.3.4.1 Using tpmigldif Command Line Options

The following table lists the command line options for the `tpmigldif` utility. The order of the command line options does not matter.

Table 5-6 tpmigldif Command Line Options

Command Line Option	Option Argument	Default Value	Usage
`-t`	user|group	user	Specifies migration type.
`-f`	template filename	tpusr-template (when type is user), or tpgrp-template(when type is group)	Specifies the template file name.
`-o`	o (output filename)	console/stdout	Specifies the output file name.
`-u`	full path name	tpusr	The full directory path for thetpusr file.
`-g`	full path name	tpgrp	The full directory path for thetpgrp file.

5.3.4.2 tpusr and tpgrp File Format

The following listing shows a `tpusr` file with five fields separated by a colon:

```
name:password(encrypted) :user id:group id:client name::
```

Listing Example `tpusr` File

```
user1:EI4xxxjrCc:16668:601:TPCLTNM,client::
         user2:EI4xxxjrCc:16669:602:TPCLTNM,client::
```

The listing shows a `tpgrp` file with three fields separated by a colon:

```
name::group id:
```

Listing Example `tpgrp` File

```
group1::601:
         group2::602:
```

- Assigning New Passwords for the tpusr File (Optional)

5.3.4.2.1 Assigning New Passwords for the tpusr File (Optional)

Before migrating the user and group information, the administrator could assign new passwords for each user so the generated LDIF output contains correct password for each user. This step is required because the passwords in the tpusr file are encrypted with one-way encryption; therefore, it is impossible to retrieve the original password from the file.

Using a text-editor, there are two methods to modify `tpusr` file passwords:

- Modify the `tpusr` file password field to change the user password for each user in the file. The password field is the *second field* in the `tpusr` file. Each user is entered on a separate line in the `tpusr` file. See listing Listing 4-5, for original `tpusr` file example.

  ```
  user1:pwd1:16668:601:TPCLTNM,client::
           user2:pwd2:16669:602:TPCLTNM,client:
  ```

- Add a new password to the last `tpusr` file field

  ```
  user1:EI4xxxjrCc:16668:601:TPCLTNM,client::pwd1:
           user2:EI4xxxjrCc:16669:602:TPCLTNM,client::pwd2:
  ```

5.3.4.3 Creating a Migration Template

The migration template is a text file used by the `tpmigldif` command utility to translate the `tpusr` or `tpgrp` file into an LDIF output file.

The following listing shows a `tpusr-template` migration file example. `<%n>` refers to a `tpusr` file field, where n starts at 1.

> **Note:**
>
> Use `<%gn>` for group field in `tpgrp` file for given user.

Listing tpusr-template

```
dn: CN=<%1>,CN=Users,DC=tuxdev,DC=bea,dc=com
         objectclass: top
         objectclass: person
         objectclass: organizationalPerson
         objectclass: user
         cn: <%1>
         description: Tuxedo User, TUXEDO_UID=<%3> TUXEDO_GID=<%4>
         password: <%7>
```

The following listing shows the LDIF output from the `tpusr`-template.

Listing LDIF Output

```
dn: CN=user1,CN=Users,DC=tuxdev,DC=bea,dc=com
         objectclass: top
         objectclass: person
         objectclass: organizationalPerson
         objectclass: user
         cn: user1
         description: Tuxedo User, TUXEDO_UID=16668 TUXEDO_GID=601
         password: pwd1

         dn: CN=user2,CN=Users,DC=tuxdev,DC=bea,dc=com
         objectclass: top
         objectclass: person
         objectclass: organizationalPerson
         objectclass: user
         cn: user2
         description: Tuxedo User, TUXEDO_UID=16669 TUXEDO_GID=602
         password: pwd2
```

5.3.5 Supported LDAP Server Template Example

Tuxedo provides an example template for supported LDAP servers. The following table lists the files: 1.

Table 5-7 Supported LDAP Server Template Example

LDAP Server	GAUTHSVR Configuration	User Migration Template	Group Migration Template
WebLogic Server	tpgauth	tpusr-template	tpgrp-template
Active Directory 2	tpgauth-ad	tpusr-template-ad	tpgrp-template-ad
IPlanet	tpgauth-iplanet	tpusr-template-iplanet	tpgrp-template-iplanet
z/OS LDAP, with RACF backend 3	tpgauth-racf	tpusr-template-racf	tpgrp-template-racf

1

All files are available under `$TUXDIR/udataobj`.

2

For Active Directory user's password cannot be added on creation. For help on how to change or reset it, please refer to Microsoft support document, http://support.microsoft.com/kb/269190, http://support.microsoft.com/kb/263991, etc;

3

Two things require to be completed for activating *z/OS RACF* account after migration:

1. reset the password by z/OS administrator
2. logon with the account to change its password

5.4 Setting up OAUTHSVR as the Authentication Server

`OAUTHSVR` is a Tuxedo provided server that offers the authentication and authorization service while the user security information is located in Oracle Access Manager (OAM) Server. To enable the single security administration feature, you must configure `OAUTHSVR` as the authentication server. At runtime, the `OAUTHSVR` authenticates and authorizes the user using OAM Server

To define `OAUTHSVR` as the authentication server, you must define the following parameters in the UBBCONFIG file:

- `SECURITY` must be set to `USER_AUTH, ACL`, or `MANDATORY_ACL` in the `RESOURCES` section.
- A `TMJAVASVR` with `<server-classname="OAUTHSVR"/>` must be specified in the `SERVERS` section.

- Setting Up the OAUTHSVR Configuration File
- /T DOMAIN Support
- Oracle SALT Support
- WTC Support
- Oracle JCA Support

5.4.1 Setting Up the OAUTHSVR Configuration File

`OAUTHSVR` supports an input configuration file that contains information such as OAM access client configuration file and the resource type mapping between Tuxedo and OAM. This configuration file is a plain text file and can be edited using any text editor and must be protected by the system using file permissions. By default the configuration file, named `tpoam.auth`, is located in `$TUXDIR/udataobj` directory. You can overwrite this file in the command line for `OAUTHSVR`. The `OAUTHSVR` configuration file contains keyword and value pairs as defined in the following table.

- Syntax Requirements for OAUTHSVR Configuration File
- OAUTHSVR Configuration File Keywords
- OAM Access Client Configuration (OAM_CONFIG_DIR)
- Examples

5.4.1.1 Syntax Requirements for OAUTHSVR Configuration File

- Although the default values for the `OAUTHSVR` configuration file are usually sufficient, a system administrator may choose to configure it with different names. Therefore, you should be aware of the following requirements for the `OAUTHSVR` configuration file:
- The `OAUTHSVR` configuration file is a plain text file.
- Keyword order does not matter; however, there must be at least one space character between the keyword and its value.
- Comments begin with the pound symbol (#). Text after the # is ignored.

> **Note:**
>
> Before an administrator can set up and use the Tuxedo OAM-based security authentication and authorization server, the administrator must register a OAM access client or use already installed WebGate. For how to register and configure OAM access client, please refer Oracle OAM documents.

5.4.1.2 OAUTHSVR Configuration File Keywords

The following table lists the `OAUTHSVR` configuration file keywords.

Table 5-8 OAUTHSVR Configuration File Keywords

Keyword	Value Type	Usage
OAM_CONFIG_DIR	string	The directory location where OAM access client configuration file will be searched. The access client configuration can be obtained by registering an access client as an OAM 11g Agent with the OAM 11g server or copied from already installed WebGate.

Table 5-8 (Cont.) OAUTHSVR Configuration File Keywords

Keyword	Value Type	Usage
RESTYPE_MAPPING	string	The resource type mapping between Tuxedo and OAM. Format is"`RESTYPE_MAPPING $TUX_RESTYPE $OAM_RESTYPE`", multiple resource types can be defined, such as `RESTYPE_MAPPING SERVICE TUXEDO_SERVICE RESTYPE_MAPPING QUEUE TUXEDO_QUEUE RESTYPE_MAPPING EVENT TUXEDO_EVENT`. If the resource type name defined in OAM is same as Tuxedo resource type, no mapping is needed.
TUXEDO_DEF_RESTYPE	string	The Tuxedo default resource type defined in OAM. The default is "`TUXEDO_SERVICE`".
TUXEDO_DEF_RESOURCE	string	The Tuxedo default resource defined in OAM. OAM always requires a resource to figure out the authentication level and policies to authenticate a user. Administrator need define a default resource. The default is "`tuxres`".

5.4.1.3 OAM Access Client Configuration (OAM_CONFIG_DIR)

OAM Access Client configuration information is required by OAM.

For more information, see OAM documents https://docs.oracle.com/cd/E52734_01/oam/AIDEV/as_api.htm#AIDEV151

- Limitations

5.4.1.3.1 Limitations

`OAUTHSVR` does not support OAM 10g agent; 11g or above WebGate agent is required. An example 11gR1PS2 OAM configuration directory is shown in the following listing .

Listing Example 11gR1PS2 OAM Configuration Directory

```
OAM_CONFIG_DIR/
        |------config/
        |------ cwallet.sso (Get from WebGate)
        |------ jps-config.xml (Get from OAM SDK)
        |------ ObAccessClient.xml (Get from WebGate)
```

Another example configuration directory for OAM12cR2 is shown in the following listing (the communication transportation security mode between the Agent and OAM server is Simple or Cert).

Listing Example 11gR1PS3 OAM Configuration Directory

```
OAM_CONFIG_DIR/
        |------config/
        |------ cwallet.sso (Get from WebGate)
        |------ jps-config.xml (Get from OAM SDK)
        |------ ObAccessClient.xml (Get from WebGate)
        |------ oamclient-keystore.jks (Get from WebGate)
        |------ oamclient-truststore.jks (Get from WebGate)
        |------ password.xml (Get from WebGate)
```

For more information, see OAM documents http://docs.oracle.com/cd/E21764_01/install.1111/e12002/webgate.htm#INOIM75755.

> **Note:**
>
> In OAM server host, under directory `<OAM_DOMAIN_HOME>/output/<WebGate_ID>` you can find cwallet.sso and ObAccessClient.xml file.

5.4.1.4 Examples

5.4.1.4.1 1. Example OAUTHSVR Configuration File

The following listing describes an example of a OAUTHSVR configuration file.

Listing Example OAUTHSVR Configuration File

```
# Tuxedo OAM Authentication Server configuration file.

        OAM_CONFIG_DIR /usr/tuxedo/accessclient
        #RESTYPE_MAPPING SERVICE TUXEDO_SERVICE
        #RESTYPE_MAPPING QUEUE TUXEDO_QUEUE
        #RESTYPE_MAPPING EVENT TUXEDO_EVENT
        TUXEDO_DEF_RESTYPE TUXEDO_SERVICE
        TUXEDO_DEF_RESOURCE tuxres
        # end of file
```

5.4.1.4.2 2. Example UBBCONFIG Using OAUTHSVR

The following listing describes an example `UBBCONFIG` file with `SECURITY` set to `ACL` and `OAUTHSVR` defined.

Listing Example UBBCONFIG File Using OAUTHSVR

```
*RESOURCES

         IPCKEY 51002
         MASTER site1
         MAXACCESSERS 50
         MAXSERVERS 20
         MAXSERVICES 20
         MODEL SHM
         LDBAL N
         BLOCKTIME 10
         SECURITY ACL
         AUTHSVC "..AUTHSVC"
         OPTIONS EXT_AA

         *MACHINES
          DEFAULT:
          APPDIR="/home/tuxedo/application"
          TUXCONFIG="/home/tuxedo/application/TUXCONFIG"
          TUXDIR="/home/tuxedo/tuxedo12"
          Server1 LMID=site1
          MAXWSCLIENTS=20

         *GROUPS
          GROUP1 LMID=site1 GRPNO=1
          GROUP2 LMID=site1 GRPNO=2
          GROUP3 LMID=site1 GRPNO=3
          GROUP4 LMID=site1 GRPNO=4

        *SERVERS
         DEFAULT:
         CLOPT="-A" RESTART=N MAXGEN=5
         TMJAVASVR SRVGRP=GROUP1 SRVID=2 CLOPT="-A -- -c tjsoam.xml"

         DMADM SRVGRP=GROUP2 SRVID=20
         GWADM SRVGRP=GROUP3 SRVID=30
         GWTDOMAIN SRVGRP=GROUP3 SRVID=31
         Simpserv SRVGRP=GROUP4 SRVID=40

        *SERVICES
         TOUPPER
```

5.4.1.4.3 3. Example tjsoam.xml Java Server Configuration File

The following listing describes an example Java Server configuration file using `OAUTHSVR`.

Listing Example Java Server Configuration File Using OAUTHSVR

```
<?xml version="1.0" encoding="UTF-8"?>
         <TJSconfig version="2.0">
         <!-- do not forget modify $TUXDIR $OAMASDK_DIR to absolute path -->
         <java-config>
         <jvm-options>-Doracle.security.jps.config=/home/tuxedo/application/
oamclient/config/jps-config.xml</jvm-options>
```

```
        <jvm-options>-Doracle.tuxedo.oam.config=tpoam.auth</jvm-options>
        </java-config>

        <classpath-config>
        <classpath>/home/tuxedo/udataobj/tuxj/oam/
com.oracle.tuxedo.tjoam.jar</classpath>
        <classpath>/home/tuxedo/oamsdk/*</classpath>
        </classpath-config>

        <tux-server-config>
        <server-class name="OAUTHSVR"/>
        </tux-server-config>

        </TJSconfig>
```

5.4.2 /T DOMAIN Support

ACL_POLICY and CREDENTIAL_POLICY impact credential propagation.

When local domain receives request from remote domain, if ACL_POLICY is set to LOCAL, the local domain removes the OAM session token of any service request received from the remote domain if session token exists. If ACL_POLICY is set to GLOBAL the local domain does not remove the OAM session token received with a remote service request.

When a Tuxedo domain sends request to a remote /T domain, if CREDENTIAL_POLICY is set to LOCAL, then the local domain removes the session token from a local service request destined for the remote domain access point. If CREDENTIAL_POLICY is set to GLOBAL, the local domain does not remove the session token from a local service request destined for this remote domain access point.

From above description we can see to pass OAM session token between Tuxedo /T domains, ACL_POLICY and CREDENTIAL_POLICY should both configured to GLOBAL, and same OAM access client configuration (OAM_CONFIG_DIR parameter in OAUTHSVR configuration file) must be used to ensure that the OAM session token is valid in both domains.

To authenticate or authorize user requests, username/password pair or valid session token issued by OAM server must exist. If both username/password pair and valid session token do not exist, it is not possible to impersonate the desired principle; authentication or authorization with OAM server cannot be done.

When domain gateway receives a request, if ACL_POLICY is set to LOCAL, or the request doesn't contain OAM session token (for example, remote domain doesn't use OAM, or CREDENTIAL_POLICY is set to LOCAL, or Tuxedo version of remote domain is not 12.2.2.0.0 or later, or remote domain can't pass OAM session token like WTC), to impersonate the desired principle, the local domain gateway replaces the credential of any service request received from the remote domain with the principle name specified in the LOCAL_PRINCIPAL_NAME parameter (if not specified, the principle name defaults to the ACCESSPOINTID string for the remote domain access point) for this remote domain access point, the password will use "Remote Domain Password", that is the SECURITY parameter in the DM_LOCAL section of the DMCONFIG file must set to DM_PW. User LOCAL_PRINCIPAL_NAME (or ACCESSPOINTID) with same password as "Remote Domain Password" must be defined in OAM. If you do not meet these prerequisites and SECURITY in UBBCONFIG is set to ACL or MANDATORY_ACL, authorization fails.

5.4.3 Oracle SALT Support

OAM integration only support SALT inbound request, for HTTP Basic Authentication GWWS will extract username and password and calls Tuxedo `AUTHSVC` to authenticate the user, `OAUTHSVR` will communicate with OAM to authenticate, if it is successful then GWWS will retrieve OAM session token, the session token is passed in following service call, `OAUTHSVR` uses the session token to authorize.

For WSSE situation, GWWS will use user credential received and authenticate with Tuxedo, before it calls Tuxedo service it will check if auth level is `TPAPPAUTH` and insert the session token into context and call Tuxedo service.

If it is either X509 authentication or SAML SSO is used then it depends on whether Basic Authentication is attached to the request. If Basic Authentication is not attached to the request, Tuxedo cannot retrieve username and password, authorization will fail.

If user is already authenticated with WebGate and the OAM session token is exist in HTTP header, GWWS will extract the token and use it to authorize.

WebGate is a agent provided for various Web Servers (Oracle HTTP server - OHS, IBM HTTP server -IHS, Apache ...) as part of the OAM product. It's installed on different HTTP server, to use OAM for authentication and authorization, HTTP server and WebGate are necessary. Often the HTTP server works as reverse proxy to backend applications, such as WLS or SALT.

> **Note:**
>
> For 11g WebGate, the OAM token cookie (OAMAuthnCookie) is not passed to downstream applications such as SALT, please specify WebGate user-defined parameter filterOAMAuthnCookie to false. For more information, see Registering and Managing OAM 11g Agents.

5.4.4 WTC Support

For WTC inbound service, client is authenticated in Tuxedo domain, the request is passed to WTC. WTC will look up the EJB name and invoke the target EJB using passed principle (`ACL_POLICY` is global) or domain name (`ACL_POLICY` is local). No authentication is required, although WLS security module will check the authorization of this principle (security identity). The target EJB will receive the identity only, it will not receive any authentication data. There is no way for WLS to authenticate the identity, the identity is only used in authorization checks. The OAM session token will not pass to WTC.

For WTC outbound service, the authentication only occurs in WLS, the authorization check should also occurs in WLS. When WTC pass the request to Tuxedo domain, user is already authenticated, WTC cannot get the OAM session token from WLS. Tuxedo local domain gateway will use the same approach as /T domain to impersonate desired principle (`LOCAL_PRINCIPAL_NAME` or `ACCESSPOINTID` and remote domain password).

5.4.5 Oracle JCA Support

Tuxedo JCA adapter can't and should not be changed for OAM as JCA architecture has its own way of importing the security principal identity, and we should not break the contract and made it not portable between different Java AS. JCA architecture specification has its own way and that is supported by all JCA 1.6 compliant JCA

See Also:

- tpmigldif(1) in the *Oracle Tuxedo Command Reference*

6

Integrating Audit with Oracle Platform Security Services (OPSS)

> **Note:**
>
> Before setting up single point security, ensure that you are familiar with the Tuxedo security architecture and requirements. You may also want to coordinate this effort with your OPSS Administrator.

6.1 Overview

Oracle Tuxedo can integrate with Oracle Platform Security Services (OPSS) audit component. This integration assures you taking all advantages that OPSS has for audit analysis and reporting.

This feature provides you:

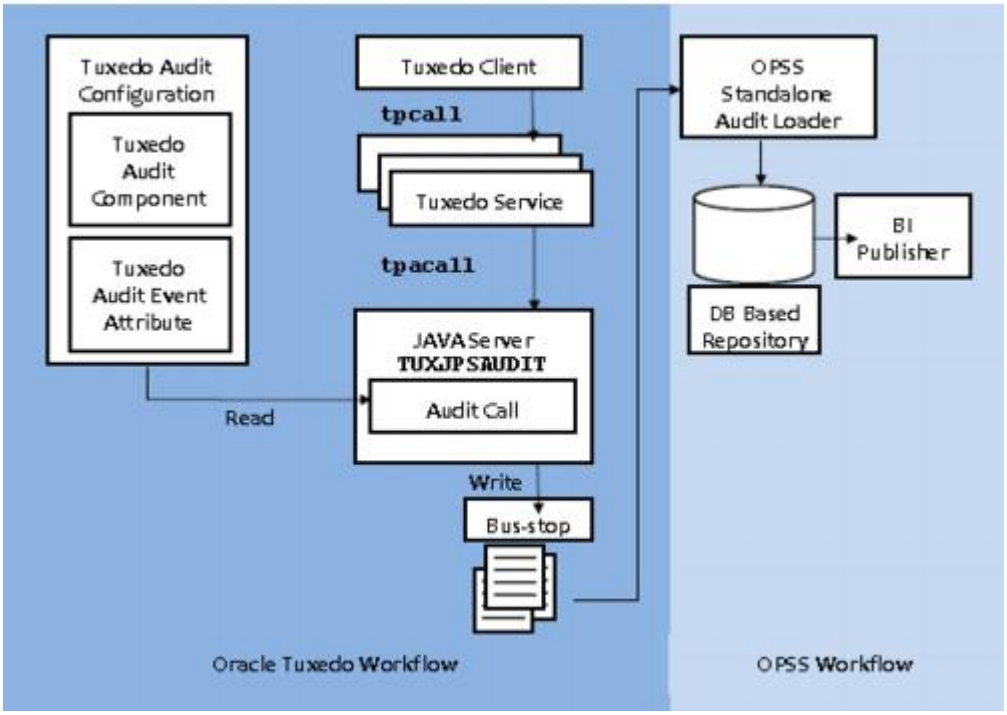
- Rich Data for Business Intelligence Analysis
  Integrated with OPSS audit component, Oracle Tuxedo can generate rich data centrally stored in an audit sotre. You can then continue to use diverse BI tools (such as OPSS) to view and analyze the data using Oracle Business Intelligence Publisher and the like.
- Customized Audit Strategies/Policies
  This feature enables you to generate data for specific events by defining these events on a static XML file (`component_events.xml`), which makes it very convenient for you to change audit strategies/policies without affecting the application.
- Easy Approach to Use
  OPSS provides an abstraction layer in the form of standards-based application programming interfaces (APIs) that insulate you from security and identity management implementation details. With the integration with OPSS audit component, Oracle Tuxedo can directly use these OPSS APIs for auditing; you do not need to write a single line of audit-related code, or change any of your existing code.

6.2 Components and Deployment

6.2.1 Audit Flow

This figure illustrates you an Oracle Tuxedo event flow with OPSS audit framework when an event (such as `tpcall`) occurs.

Figure 6-1 Oracle Tuxedo Audit Flow with OPSS



Oracle Tuxedo Part
This part requires you to configure Oracle Tuxedo as Configurations instructs. On this figure, an Oracle Tuxedo client invokes an Oracle Tuxedo service. This service then sends a request (also known as "an event") to Oracle Tuxedo Java Server, which has already configured OPSS audit module (`.TUXJPSAUDIT`). This module then invokes OPSS audit APIs, which check if this event should be audited. If it should be, this module audits it to a local file in an intermediate location (known as the "bus-stop"), creating the audit event structure and collecting event information such as status, initiator, resource, and ECID.

OPSS Part
This part requires you to configure OPSS standalone Audit Loader. Once OPSS APIs audit an event to the bus-stop, OPSS Audit Loader collects the audit records throughout all components running in the instance. If a database is configured for an audit store, the OPSS Audit Loader pulls the event from bus-stop and moves its data to audit store.

6.3 Configurations

Do the following to configure Oracle Tuxedo for this feature:

- Register OPSS Audit Plug-In to Oracle Tuxedo Registry
- Configure Oracle Tuxedo Auditing Framework

- Configure Oracle Tuxedo OPSS Audit Module
- Configure OPSS Configuration Files
- Configure OPSS Audit Bus-Stop

6.3.1 Register OPSS Audit Plug-In to Oracle Tuxedo Registry

Oracle Tuxedo registry is a disk-based repository for storing information related to plug-in modules. Initially, this registry holds registration information about the default security plug-ins; now that you want to use this feature, you must configure this registry for the OPSS plug-in before installing it.

- Register OPSS Audit Plug-In to Oracle Tuxedo Registry

  If you do not want to use this feature any more, you can

- Unregister OPSS Audit Plug-In from Oracle Tuxedo Registry
- Register OPSS Audit Plug-In to Oracle Tuxedo Registry
- Unregister OPSS Audit Plug-In from Oracle Tuxedo Registry

6.3.1.1 Register OPSS Audit Plug-In to Oracle Tuxedo Registry

You must use `epifreg` tool to register OPSS plug-ins for Oracle Tuxedo registry. Registering OPSS plug-in will replace the default Oracle Tuxedo audit implement from ULOG to OPSS audit.

The following listing describes an example.

Listing Example for Registering OPSS Audit Plug-In to Oracle Tuxedo Registry

```
epifregedt -s -k "SYSTEM/impl/bea/native/audfan" -a InterceptionSeq=bea/
native/audopss


epifreg -r -p bea/native/audopss -i engine/security/auditing -v 1.0 -f
libtux.so -e _ep_dl_audopss
```

Use the following shell script tools located at `$TUXDIR/bin` for registering OPSS audit plug-in to Oracle Tuxedo registry:

- `opssreg.bat`
  Command of registering OPSS Audit Plug-In for Oracle Tuxedo on windows platforms.
- `opssreg.sh`
  Command of registering OPSS Audit Plug-In for Oracle Tuxedo on UNIX platforms.

6.3.1.2 Unregister OPSS Audit Plug-In from Oracle Tuxedo Registry

You should use `epifunreg`tool to unregister OPSS plug-in from Oracle Tuxedo registry. Unregistering OPSS plug-in will restore the default Oracle Tuxedo audit implement back to ULOG.

The following listing describes an example.

Listing Example for Unregistering OPSS Audit Plug-In from Oracle Tuxedo Registry

```
epifunreg -p bea/native/audopss
```

Use the following shell script tools located at `.$TUXDIR/bin` for unregistering OPSS audit plug-in from Oracle Tuxedo registry:

- `opssunreg.bat`
  Command of unregistering OPSS Plug-In for Oracle Tuxedo on windows platforms.
- `opssunreg.sh`
  Command of unregistering OPSS Plug-In for Oracle Tuxedo on UNIX platforms.

6.3.2 Configure Oracle Tuxedo Auditing Framework

This feature requires you to create Oracle Tuxedo Auditing framework.

- Add OPSS Audit to Oracle Tuxedo Plug-In Framework
  Follow Oracle Tuxedo Auditing configuration rules to add OPSS audit to Tuxedo Plug-in framework. See Auditing for more information.
- Configure Security Options in `UBBCONFIG`
  In `UBBCONFIG RESOURCES` section, set `SECURITY` option to `ACL` or `MANDATORY_ACL`.

6.3.3 Configure Oracle Tuxedo OPSS Audit Module

This feature requires to configure Oracle Tuxedo OPSS Audit Module.

- Configure Oracle Tuxedo Java Server (TMJAVASVR)
- Configure Oracle Tuxedo OPSS Audit Module
- Configure Oracle Tuxedo Java Server (TMJAVASVR)
- Configure Oracle Tuxedo OPSS Audit Module

6.3.3.1 Configure Oracle Tuxedo Java Server (TMJAVASVR)

Oracle Tuxedo OPSS Audit Module runs in Oracle Tuxedo Java server (`TMJAVASVR`), so `TMJAVASVR` must be configured in your `UBBCONFIG`. `TMJAVASVR` handles the entire audit request, advertising audit module `.TUXJPSAUDIT`, which acts as a bridge between Oracle Tuxedo system with OPSS audit and Oracle Tuxedo application services. The following listing describes an example for configuring `TMJAVASVR` in `UBBCONFIG SERVERS` section.

`TMJAVASVR` can

- Read configuration file `tpopss_audit.xml`.
- Advertise audit module `.TUXJPSAUDIT`, which is implemented with Java code according to `tpopss_audit.xml`.
- Launch JVM.
- Forward an audit request to .TUXJPSAUDIT.
- Get and execute the results from this .TUXJPSAUDIT.

Listing TMJAVASVR Configuration Example

```
*SERVERS
          TMJAVASVR
          SRVGRP=TJSVRGRP SRVID=3
          CLOPT="-- -c/home/oracle/app/javaserver/tpopss_audit.xml"
          MINDISPATCHTHREADS=2 MAXDISPATCHTHREADS=3
```

6.3.3.2 Configure Oracle Tuxedo OPSS Audit Module

Now that you have configured `TMJAVASVR`, you can configure Oracle Tuxedo OPSS Audit Module in Oracle Tuxedo Java Server Configuration File called `tpospss_audit.xml`, which you can find in `${TUXDIR}/udataobj/tuxj/opss`.

Two packages that Oracle Tuxedo Java Server uses for this feature are `com.oracle.tuxedo.tjopss_12.2.2.0.jar` (Oracle Tuxedo ships it and it is located in `${TUXDIR}/udataobj/tuxj/opss`) and `opss-manifest.jar` (OPSS ships it and it is located in the path where `-Dcommon.components.home` specifies. For example, if you specify `-Dcommon.components.home=/testarea/tuxuser/opss_standalone/`, this `opss-manifest.jar` is located in `/testarea/tuxuser/opss_standalone/modules/oracle.jps_12.1.2/opss-manifest.jar`).

The listing for an example, where the following attributes are specified.

- java-config
- classpath-config

6.3.3.2.1 java-config

Declare the following `jvm-options` attributes, and ensure that every path you set is an absolute path.

- Required `jvm-options` are
  - `-Doracle.security.jps.config`
    This declares the absolute path of OPSS configuration file `jps-config.xml`.
  - `-Djava.security.policy`
    This declares the absolute path of `java.policy`.
  - `-Doracle.tuxedo.opss.event.config.dir`
    This declares the absolute path of `component_events.xml`.
  - `-Doracle.tuxedo.audit.type`
    This declares the component type of `TMJAVASVR .TUXJPSAUDIT`, determining which component table stores the record to the bus-stop.
  - `-Doracle.tuxedo.audit.category`
    This declares the component event category of `TMJAVASVR .TUXJPSAUDIT`.
  - `-Dcommon.components.home`
    This declares the absolute path of OPSS component home directory.
- Optional `jvm-options` are
  - `-Djps.auth.debug`
  - `-Djps.auth.debug.verbose`

If these two jvm-options are set to true, JPS debug/trace functions are open.

6.3.3.2.2 classpath-config

Declare the following `classpath` attributes, and make sure every path you set is an absolute path.

- `com.oracle.tuxedo.tjopss_12.2.2.0.jar`
  Oracle Tuxedo ships `com.oracle.tuxedo.tjopss_12.2.2.0.jar` to integrate OPSS Audit module. This library is located at `$TUXDIR/udataobj/tuxj/opss`.

  You should declare this path in `<classpath-config> classpath` attribute in `tpopss_audit.xml`.

- `opss-manifest.jar`
  Oracle OPSS module ships `opss-manifest.jar`.

  You should declare this path in `<classpath-config> classpath` attribute in `tpopss_audit.xml`.

Listing Example for tpopss_audit.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<TJSconfig version="2.0">
<!--
do not forget modify $TUXDIR $COMMON_COMPONENTS_HOME $APPDIR to absolute path
-->
<java-config>
<jvm-options>-Dtuxedo.tjatmi.strictly_check=yes</jvm-options>
<jvm-options>-Doracle.security.jps.config=${TUXDIR}/udataobj/tuxj/opss/jps
-config.xml</jvm-options>
<jvm-options>-Djava.security.policy=${TUXDIR}/udataobj/tuxj/opss/java.poli
cy</jvm-options>
<jvm-options>-Doracle.tuxedo.opss.event.config.dir=${TUXDIR}/udataobj/tuxj
/opss/</jvm-options>
<jvm-options>-Doracle.tuxedo.audit.type=tuxedo_opss_template</jvm-options>
<jvm-options>-Doracle.tuxedo.audit.category=TUXEDOOPSSAUDIT</jvm-options>
<jvm-options>-Dcommon.components.home=${COMMON_COMPONENTS_HOME}</jvm-options>

<jvm-options>-Djps.auth.debug=true</jvm-options>
<jvm-options>-Djps.auth.debug.verbose=true</jvm-options>
</java-config>

<classpath-config>
<classpath>${TUXDIR}/udataobj/tuxj/opss/com.oracle.tuxedo.tjopss_13.1.1.0.
jar</classpath>
<classpath>${COMMON_COMPONENTS_HOME}/modules/oracle.jps_12.1.2/opss-
manifest.jar</classpath>
</classpath-config>

<tux-server-config>
<server-class name="TuxAuditServer"/>
</tux-server-config>

</TJSconfig>
```

6.3.4 Configure OPSS Configuration Files

This feature requires you to configure the following OPSS configuration files. All of them are located at `$TUXDIR/udataobj/tuxj/opss`.

- jps-config.xml
- java.policy
- component_events.xml (static) and audit-store.xml (dynamic)
- system-jazn-data.xml

- jps-config.xml
- java.policy
- component_events.xml (static) and audit-store.xml (dynamic)
- system-jazn-data.xml

6.3.4.1 jps-config.xml

Oracle Tuxedo integrates with the Oracle Fusion Middleware Audit Framework through `jps-config.xml` runtime configuration file, which contains the initial filter settings for using OPSS Audit Plug-In. You should declare its absolute path in `tpopss_audit.xml` configuration file (jvm-options: `-Doracle.security.jps.config`). See Configure Oracle Tuxedo OPSS Audit Module for more information.

See the following listing describes for an example, where `jps-config.xml` declares `serviceInstance audit`, whose provider is `audit.provider` and location is `./audit-store.xml`.

Listing jps-config.xml Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jpsConfig xmlns="http://xmlns.oracle.com/oracleas/schema/11/jps-
config-11_1.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" schema-major-
version="11" schema-minor-version="1"
xsi:schemaLocation="http://xmlns.oracle.com/oracleas/schema/11/jps-
config-11_1.xsd jps-config-11_1.xsd">
<property name="oracle.security.jps.jaas.mode" value="off"/>
<property name="oracle.security.jps.enterprise.user.class"
value="weblogic.security.principal.WLSUserImpl"/>
<property name="oracle.security.jps.enterprise.role.class"
value="weblogic.security.principal.WLSGroupImpl"/>
<propertySets>
<propertySet name="saml.trusted.issuers.1">
<property name="name" value="www.oracle.com"/>
</propertySet>
<propertySet name="trust.provider.embedded">
<property name="trust.provider.className"
value="oracle.security.jps.internal.trust.provider.embedded.EmbeddedProviderIm
pl"/>
<property name="trust.clockSkew" value="60"/>
<property name="trust.token.validityPeriod" value="1800"/>
<property name="trust.token.includeCertificate" value="false"/>
```

```
</propertySet>
</propertySets>
<serviceProviders>
<serviceProvider type="POLICY_STORE" name="policystore.xml.provider"
class="oracle.security.jps.internal.policystore.xml.XmlPolicyStoreProvider">
<description>XML-based PolicyStore Provider</description>
</serviceProvider>
<serviceProvider type="AUDIT" name="audit.provider"
class="oracle.security.jps.internal.audit.AuditProvider">
<description>Audit Service</description>
</serviceProvider>
</serviceProviders>
<serviceInstances>
<serviceInstance name="policystore.xml"
provider="policystore.xml.provider" location="./system-jazn-data.xml">
<description>File Based Policy Store Service
Instance</description>
</serviceInstance>
<serviceInstance name="audit" provider="audit.provider" location="./audit-
store.xml">
<description>Audit Service</description>
<property name="audit.filterPreset" value="None"/>
<property name="audit.maxDirSize" value="0"/>
<property name="audit.maxFileSize" value="104857600"/>
<property name="audit.timezone" value="utc"/>
<property name="audit.loader.interval" value="15"/>
<property name="audit.loader.repositoryType" value="File"/>
<property name="auditstore.type" value="file"/>
<property name="audit.db.principal.map"
value="AuditDbPrincipalMap"/>
<property name="audit.db.principal.key"
value="AuditDbPrincipalKey"/>
</serviceInstance>
</serviceInstances>
<jpsContexts default="default">
<jpsContext name="default">
<serviceInstanceRef ref="policystore.xml"/>
<serviceInstanceRef ref="audit"/>
</jpsContext>
</jpsContexts>
</jpsConfig>
```

6.3.4.2 java.policy

`java.policy` is the system policy file to grant system-wide code permissions; this policy is represented by a Policy object for Java programming language application environment (specifying which permissions are available for code from various sources, and executing as various principals). For this feature in particular, you should use this file to grant audit store access permissions to all domains invoke OPSS Audit APIs. You must declare its absolute path in `tpopss_audit.xml` configuration file (jvm-options: `-D Djava.security.policy`). See Configure Oracle Tuxedo OPSS Audit Module for more information. See the following listing for an example, where the following two grants are specifically added for this feature.

- This is to grant permissions to `file:${common.components.home}/modules/oracle.jps_12.1.2/*`, where `opss-manifest.jar` is located.

```
grant codeBase "file:${common.components.home}/modules/
oracle.jps_12.1.2/*" {
         permission java.security.AllPermission;
         };
```

- This is to grant permissions to `file:${oracle.deployed.app.dir}/*`, where required configuration files (`tpopss_audit.xml`, `jps-config.xml`, `component_events.xml`, `audit-store.xml`, `java.policy`, and `system-jazn-data.xml`) are located.

```
grant codeBase "file:${oracle.deployed.app.dir}/*" {
         permission java.security.AllPermission;
};
```

```
// Standard extensions get all permissions by default

         grant codeBase "file:${{java.ext.dirs}}/*" {
         permission java.security.AllPermission;
         };
         // default permissions granted to all domains
         grant {
         // Allows any thread to stop itself using the
java.lang.Thread.stop()
        // method that takes no argument.
        // Note that this permission is granted by default only to remain
        // backwards compatible.
        // It is strongly recommended that you either remove this permission
       // from this policy file or further restrict it to code sources
       // that you specify, because Thread.stop() is potentially unsafe.
      // See "http://java.sun.com/notes" for more information.
         permission java.lang.RuntimePermission "stopThread";

     // allows anyone to listen on un-privileged ports permission
         java.net.SocketPermission "localhost:1024-", "listen";

    //  "standard" properies that can be read by anyone

         permission java.util.PropertyPermission "java.version", "read";
         permission java.util.PropertyPermission "java.vendor", "read";
         permission java.util.PropertyPermission "java.vendor.url", "read";
         permission java.util.PropertyPermission "java.class.version", "read";
         permission java.util.PropertyPermission "os.name", "read";
         permission java.util.PropertyPermission "os.version", "read";
         permission java.util.PropertyPermission "os.arch", "read";
         permission java.util.PropertyPermission "file.separator", "read";
         permission java.util.PropertyPermission "path.separator", "read";
         permission java.util.PropertyPermission "line.separator", "read";

         permission java.util.PropertyPermission
"java.specification.version", "read";
         permission java.util.PropertyPermission "java.specification.vendor",
"read";
         permission java.util.PropertyPermission "java.specification.name",
```

```
"read";
         permission java.util.PropertyPermission
"java.vm.specification.version", "read";
         permission java.util.PropertyPermission
"java.vm.specification.vendor", "read";
         permission java.util.PropertyPermission
"java.vm.specification.name", "read";
         permission java.util.PropertyPermission "java.vm.version", "read";
         permission java.util.PropertyPermission "java.vm.vendor", "read";
         permission java.util.PropertyPermission "java.vm.name", "read";
         };

         grant codeBase
          "file:${common.components.home}/modules/oracle.jps_12.1.2/*"
           permission java.security.AllPermission;
           {;

         grant codeBase "file:${common.components.home}/modules/
oracle.iau_12.1.2/*" {
         permission java.security.AllPermission;
         };

         grant codeBase "file:${classpath}/*" {
         permission java.security.AllPermission;
         };

         grant codeBase "file:${oracle.deployed.app.dir}/*" {
         permission java.security.AllPermission;
         };

         grant codeBase "file:${TUXDIR}/udataobj/tuxj/opss/*" {
         permissionjava.security.AllPermission;
         };

         grant codeBase "file:${TUXDIR}/udataobj/tuxj/*" {
         permission java.security.AllPermission; };
```

6.3.4.3 component_events.xml (static) and audit-store.xml (dynamic)

`component_events.xml` is a static file that defines all the audit events that are generated by the OPSS Audit Plug-In; `audit-store.xml` is the dynamic file that defines all the audit events that are mapped from the static file `component_events.xml`.

After `tmboot` for Oracle Tuxedo audit module `.TUXJPSAUDIT`, audit policy for a specific component is stored in `audit-store.xml`. `.TUXJPSAUDIT` automatically registers the event component, and maps it from `component_events.xml` to `audit-store.xml`; after automatically un-registering the event component, `.TUXJPSAUDIT` drops it from `audit-store.xml`.

> **Note:**
>
> As Oracle Tuxedo depends on OPSS stand-alone component, `audit-store.xml` is actually the file that is mainly used for this feature. Nevertheless, you are still required to use the static file `component_events.xml` to adjust your audit policy and specify `component_events.xml` in your `CLASSPATH`. See Change Audit Policy for more information.

You must declare the absolute path for `component_events.xml` in `tpopss_audit.xml` configuration file (jvm-options: `-Doracle.tuxedo.opss.event.config.dir`). See Configure Oracle Tuxedo OPSS Audit Module for more information.

In `component_events.xml` configuration file, you must set:

- `componentType`
  Audit-Aware Components, referring to components that are integrated with the Oracle Fusion Middleware Audit Framework so that audit policies can be configured and events can be audited for these components. You should also set `componentType` in `tpopss_audit.xml` (jvm-options: `-Doracle.tuxedo.audit.type`).
- `category`
  An audit event category contains related events in a functional area. Attributes are categorized into base. You should also set `category` in `tpopss_audit.xml` (jvm-options: `-Doracle.tuxedo.audit.category`).

  See the following listing for an example, where

- `componentType` is set to `tuxedo_opss_template` (the same as "`<jvm-options>-Doracle.tuxedo.audit.type=tuxedo_opss_template</jvm-options>`").
- `category` is set to `TUXEDOOPSSAUDIT` (the same as "`<jvm-options>-Doracle.tuxedo.audit.category=TUXEDOOPSSAUDIT</jvm-options>`").

Listing component_events.xml Example

```
<?xml version="1.0" encoding="UTF-8"?><AuditConfig
         xmlns="http://xmlns.oracle.com/ias/audit/audit-2.0.xsd">
         <AuditComponent minor="0" major="1"
componentType="tuxedo_opss_template">
         <Attributes version="1.0" ns="tuxedo_opss_template">
         <Attribute order="1" displayName="visitor ID" required="true"
searchable="true" maxLength="255" type="string" name="visitorid">
         <HelpText>Visitor ID</HelpText>
         </Attribute>
         <Attribute order="2" displayName="Start Time" required="true"
searchable="true" maxLength="2048" type="dateTime"name="starttime">
<HelpText>The time a visitor enters.</HelpText> </Attribute> <Attribute
order="3"
         displayName="End Time" required="true" searchable="true"
         maxLength="2048" type="dateTime" name="endtime">
         <HelpText>the time a visitor exists.</HelpText>
         </Attribute><Attribute order="4" displayName="Service
         Charge" required="true" searchable="true" maxLength="2048"
         type="float" name="fee"> <HelpText>the dollar amount a
         visitor pays.</HelpText> </Attribute><Attribute
         order="5" displayName="Service Item" required="true"
```

```
            searchable="true" maxLength="2048" type="float" name="item">
            <HelpText>the name of an item a visitor
            borrows.</HelpText>
            </Attribute></Attributes><Events>
            <Category displayName="TUXEDOOPSSAUDIT" name="TUXEDOOPSSAUDIT">
            <Attributes>
            <Attribute version="1.1" ns="common" name="EventType">
            <HelpText>The type of the audit event. Use wlst listAuditEvents to
list out all the events.</HelpText>
            </Attribute>
            <Attribute version="1.0" ns="tuxedo_opss_template" name="visitorid">
            <HelpText>Visitor ID</HelpText>
            </Attribute>
            <Attribute version="1.0" ns="tuxedo_opss_template"
            name="starttime"> <HelpText> The time a visitor centers.</HelpText>
            </Attribute>
            <Attribute version="1.0" ns="tuxedo_opss_template" name="endtime">
            <HelpText>the time a visitor exists.</HelpText>
            </Attribute>
            </Attributes>
            <HelpText>TUXEDOOPSSAUDIT category</HelpText> <Event
displayName="SERVICECALL" name="SERVICECALL">
            <HelpText>A service call enters the facility.</HelpText>
            </Event>
            <Event displayName="ENQUEUE" name="ENQUEUE">
            <HelpText>A enqueue enters the facility.</HelpText>
            </Event> <Event displayName="DEQUEUE" name="DEQUEUE">
            <HelpText>A dequeue enters the facility.</HelpText>
            </Event> <Event displayName="POST" name="POST">
            <HelpText>A post call enters the facility.</HelpText>
            </Event> <Event displayName="CONNECT" name="CONNECT">
            <HelpText>A connect enters the facility.</HelpText>
            </Event>
            <Event displayName="IMPERSONATE" name="IMPERSONATE">
            <HelpText>A impersonate enters the facility.</HelpText>
            </Event>
            <Event displayName="LOGON" name="LOGON">
            <HelpText>A logon enters the facility.</HelpText>
            </Event>
            <Event displayName="LOGOFF" name="LOGOFF">
            <HelpText>A logoff enters the facility.</HelpText>
            </Event>
            <Event displayName="DECRYPT" name="DECRYPT">
            <HelpText>A decrypt enters the facility.</HelpText>
            </Event>
            <Event displayName="SERVICESIGNATURE" name="SERVICESIGNATURE">
            <HelpText>A service signature enters the facility.</HelpText>
            </Event>
            <Event displayName="SERVICEENCRYPTION" name="SERVICEENCRYPTION">
            <HelpText>A service encryption enters the facility.</HelpText>
            </Event>
            <Event displayName="QUEUESIGNATURE" name="QUEUESIGNATURE">
            <HelpText>A queue signature enters the facility.</HelpText>
            </Event>
            <Event displayName="EVENTSIGNATURE" name="EVENTSIGNATURE">
            <HelpText>A event signature enters the facility.</HelpText>
```

```
        </Event>
        <Event displayName="EVENTENCRYPTION" name="EVENTENCRYPTION">
        <HelpText>A event encryption enters the facility.</HelpText>
        </Event>
        <Event displayName="SIGNATURE" name="SIGNATURE">
        <HelpText>A signature enters the facility.</HelpText>
        </Event>
        <Event displayName="QUEUEENCRYPTION" name="QUEUEENCRYPTION">
        <HelpText>A queue encryption enters the facility.</HelpText>
        </Event>
        <Event displayName="UNKNOWN" name="UNKNOWN"> <HelpText>A unknown
enters the facility.</HelpText>
        </Event>
        </Category> </Events> <FilterPresetDefinitions>
        <FilterPresetDefinition helpText="" displayName="Low"
        name="Low"> <FilterCategory enabled="partial"

name="TUXEDOOPSSAUDIT">SERVICECALL,ENQUEUE,DEQUEUE,POST,CONNECT,IMPERSONATE,LO
GON,LOGOFF,DECRYPT,SERVICESIGNATURE,SERVICEENCRYPTION,QUEUESIGNATURE,QUEUEENCR
YPTION,EVENTSIGNATURE,EVENTENCRYPTION,SIGNATURE,UNKNOWN</FilterCategory>
        </FilterPresetDefinition> <FilterPresetDefinition
        helpText="" displayName="Medium" name="Medium">
        <FilterCategory enabled="partial"

name="TUXEDOOPSSAUDIT">SERVICECALL,ENQUEUE,DEQUEUE,POST,CONNECT,IMPERSONATE,LO
GON,LOGOFF,DECRYPT,SERVICESIGNATURE,SERVICEENCRYPTION,QUEUESIGNATURE,QUEUEENCR
YPTION,EVENTSIGNATURE,EVENTENCRYPTION,SIGNATURE,UNKNOWN</FilterCategory>
        </FilterPresetDefinition> </FilterPresetDefinitions>
        <Policy filterPreset="Custom">
        <CustomFilters>
        <FilterCategory enabled="partial"

name="TUXEDOOPSSAUDIT">SERVICECALL,ENQUEUE,DEQUEUE,POST,CONNECT,IMPERSONATE,LO
GON,LOGOFF,DECRYPT,SERVICESIGNATURE,SERVICEENCRYPTION,QUEUESIGNATURE,QUEUEENCR
YPTION,EVENTSIGNATURE,EVENTENCRYPTION,SIGNATURE,UNKNOWN</FilterCategory>
        </CustomFilters>
        </Policy>
        <AttributesMapping version="1.0" tableName="IAU_CUSTOM"
ns="tuxedo_opss_template">
        <AttributeColumn datatype="string" column="IAU_STRING_001"
attribute="visitorid"/>
        <AttributeColumn datatype="dateTime" column="IAU_DATETIME_001"
attribute="starttime"/>
        <AttributeColumn datatype="dateTime" column="IAU_DATETIME_002"
attribute="endtime"/>
        <AttributeColumn datatype="float" column="IAU_FLOAT_001"
attribute="fee"/>
        <AttributeColumn datatype="float" column="IAU_FLOAT_002"
attribute="item"/>
        </AttributesMapping>
        </AuditComponent>
        </AuditConfig>
```

6.3.4.4 system-jazn-data.xml

`system-jazn-data.xml` is an OPSS configuration file. Oracle Tuxdo provides this file by default in `$TUXDIR/udataobj/tuxj/opss`, and uses it for this feature. You should keep this file as it is and should not change or remove it. See Oracle Fusion Middleware Security Guide for more information about this file.

6.3.5 Configure OPSS Audit Bus-Stop

This feature requires you to configure OPSS audit bus-stop.

OPSS audit bus-stop files are named `audit_<rotation_index>.log`. You can use underscore ("_") as a separator (current file should not have `_<rotation_index>`).

The location of audit bus-stop files is currently not configurable. Oracle Tuxedo OPSS audit bus-stop file locates at the parent directory of `jps-config.xml`. For example,

```
<parent directory of
         jps-config.xml>/logs/auditLogs/<componentType>
```

See the following listing for an example.

Listing OPSS Audit Bus-Stop File Example

```
#Fields:Date Time Initiator EventType EventStatus MessageText AuditUser
ApplicationName AuditService:TransactionId ContextFields
         DomainName ECID EventCategory FailureCode HomeInstance HostId
HostNwaddr MajorVersion MinorVersion RID RemoteIP Resource Roles
         SessionId Target TargetComponentType TenantId ThreadId TransactionId
UserSession:AuthenticationMethod UserTenantId
         tuxedo_opss_template:endtime tuxedo_opss_template:fee
tuxedo_opss_template:item tuxedo_opss_template:starttime
         tuxedo_opss_template:visitorid

         #Remark Values:ComponentType="tuxedo_opss_template"
ReleaseVersion="MAIN"
          2015-05-12 06:27:39.259 - "SERVICECALL" true "WARN: TUXEDO AUDIT:
who = U1, operation_name = SERVICECALL, operation_target = TOUPPER, status =
operation success" - - - - - "0000Kp6nelk8TsS_MDS4ye1LKPpR000002,0"
         "TUXEDOOPSSAUDIT" - - "wyhbj" "10.182.54.145" "1" "0" - - - - - - - -
- "12" - - - 2015-05-12 06:27:38.794 - - 2015-05-12 06:27:38.794 "U1"
```

6.4 Administration

- Change Audit Policy

6.4.1 Change Audit Policy

You can add/remove/change events in the static configuration file `component_events.xml` to change audit policy at any time. Your audit policy change is effective right after you restart Oracle Tuxedo (after `tmboot`, `.TUXJPSAUDIT` automatically update the audit policy in the corresponding dynamic `audit-store.xml`).

See component_events.xml (static) and audit-store.xml (dynamic) for more information.

Glossary

Index