Oracle® Tuxedo

Using Security in ATMI Applications

12*c* Release 2 (12.2.2)

April 2016

ORACLE®

Contents

3. Programming Security

4. Implementing Single Point Security Administration

CHAPTER 1

Introducing ATMI Security

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

- What Security Means
- Security Plug-ins
- ATMI Security Capabilities
- Operating System (OS) Security
- Authentication
- Auditing
- Link-Level Encryption
- SSL Encryption
- Public Key Security
- Message-based Digital Signature
- Message-based Encryption
- Public Key Implementation
- Default Authentication and Authorization
- Security Interoperability

- Security Compatibility
- Denial-of-Service (DoS) Defense
- Password Pair Protection

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*.

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.

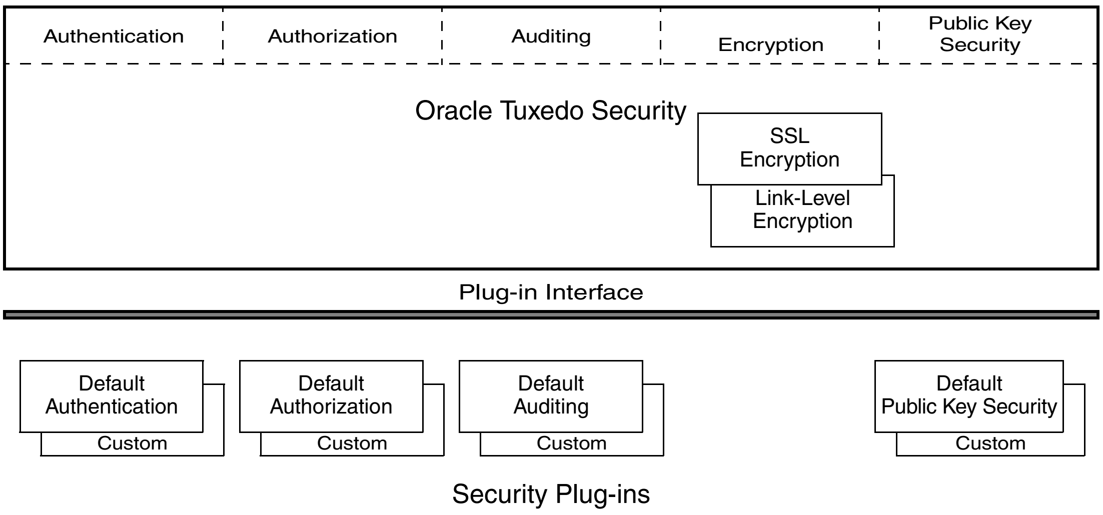
See Also

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

Security Plug-ins

As shown in Figure 1-1, 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

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. Table 1-1 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 authentication 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.

Table 1-1 ATMI Security Capabilities (Continued)

Security Capability	Description	Plug-in Interface	Default Implementation
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.
SSL encryption	Uses the industry-standard TLS 1.0 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 (OS) Security
- Authentication
- Authorization
- Auditing
- Link-Level Encryption
- SSL Encryption
- Public Key Security

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*

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.

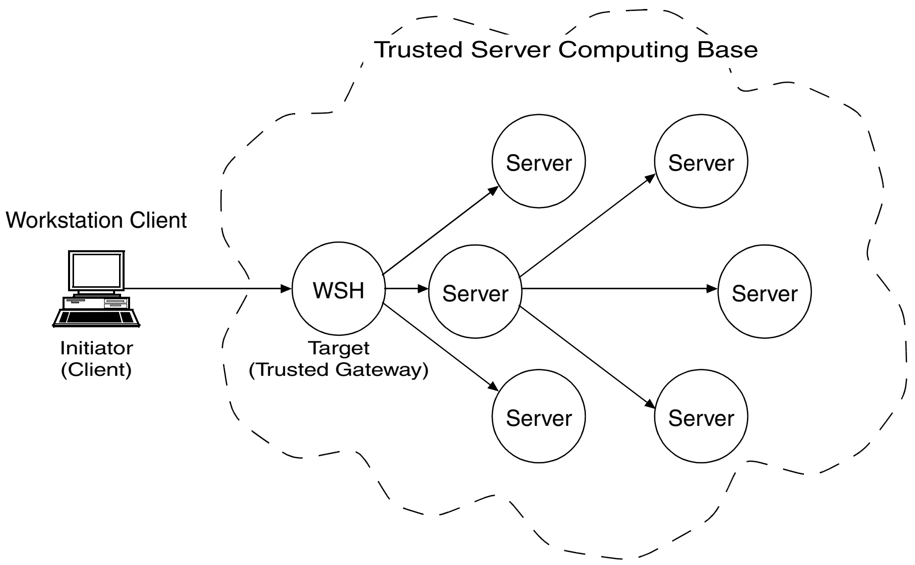
Authentication Plug-in Architecture

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.

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 Figure 1-2.

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.

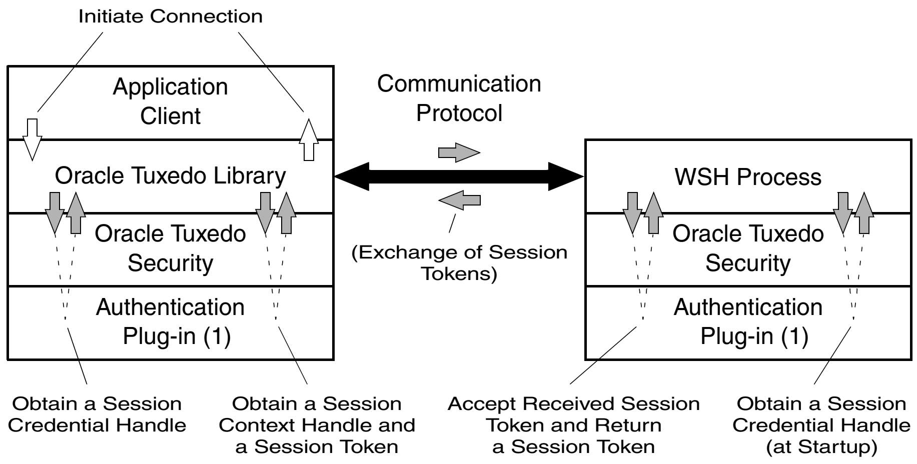
As a client request flows through a trusted gateway, the gateway attaches the client's security token to the request. The security token travels with the client's request message, and is delivered to the destination server process(es) for authorization checking and auditing purposes.

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.

Establishing a Session

Figure 1-3 shows 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 Client-WSH Authentication


Initiate Connection
Application Client
Oracle Tuxedo Library
Oracle Tuxedo Security
Authentication Plug-in (1)
Communication Protocol
(Exchange of Session Tokens)
WSH Process
Oracle Tuxedo Security
Authentication Plug-in (1)
Obtain a Session Credential Handle
Obtain a Session Context Handle and a Session Token
Accept Received Session Token and Return a Session Token
Obtain a Session Credential Handle (at Startup)

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.

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.

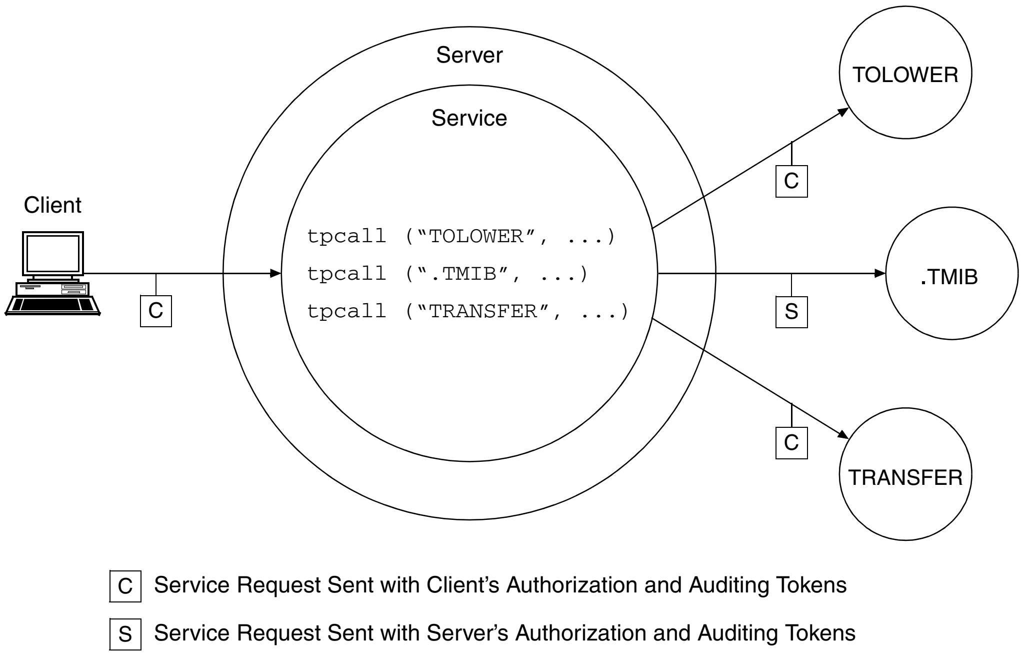
In addition, when default authentication is used, the auditing token carries the same two pieces of information: *principal name* and *application key*.

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.

Replacing Client Tokens with Server Tokens

As shown in Figure 1-4, 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


Server
Service
Client
tpcall ("TOLOWER", ...)
tpcall (".TMIB", ...)
tpcall ("TRANSFER", ...)
TOLOWER
.TMIB
TRANSFER
C
S
C
C
C Service Request Sent with Client's Authorization and Auditing Tokens
S Service Request Sent with Server's Authorization and Auditing Tokens

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*.

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

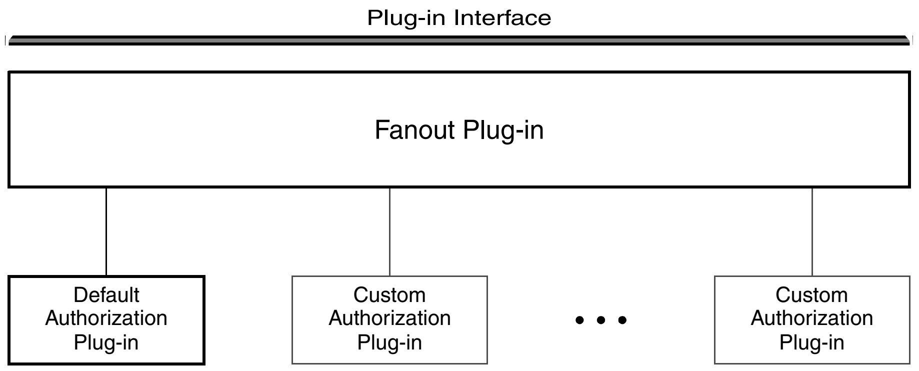
Authorization

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

A fanout is an umbrella plug-in to which individual plug-in implementations are connected. As shown in Figure 1-5, the authorization plug-in interface is implemented as a fanout.

Figure 1-5 Authorization Plug-in Architecture


Plug-in Interface
Fanout Plug-in
Default Authorization Plug-in
Custom Authorization Plug-in
• • •
Custom Authorization Plug-in

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 Table 1-2. 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*

Table 1-2 Authorization Composite Responses (Continued)

If Plug-ins Return . . .	The Composite Response Is . . .
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.

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

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.

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.

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

Auditing

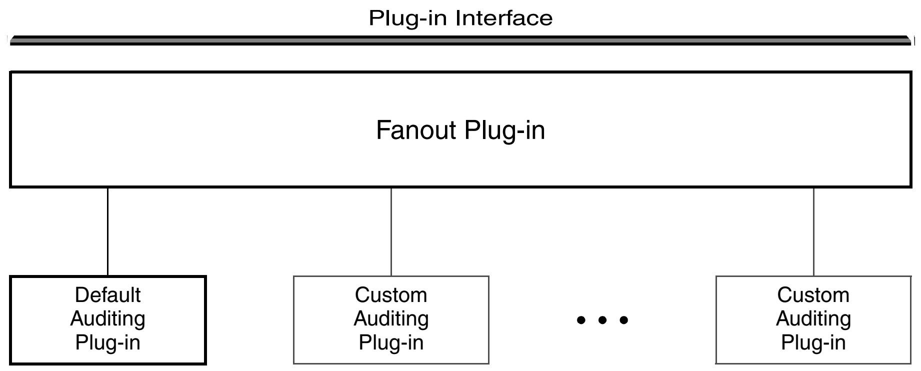
Auditing provides a means to collect, store, and distribute information about operating requests and their outcomes. Audit-trail records may be used to determine which principals performed, or attempted to perform, actions that violated the security levels of an ATMI application. They may also be used 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

A fanout is an umbrella plug-in to which individual plug-in implementations are connected. As shown in Figure 1-6, 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.

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

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.

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.

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. For more detail about the registry, see , "".

Now Oracle Tuxedo supports Oracle Platform Security Services (OPSS) plug-in. See Integrating Audit with Oracle Platform Security Services (OPSS) for more information.

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

How LLE Works

LLE 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.

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

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.

Finding a Common Key Size

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

Table 1-3 shows which key size, if any, is agreed upon by two processes when all possible combinations of `min-max` values are negotiated. The header row holds the `min-max` values for one process; the far left column holds the `min-max` values for the other.

Table 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

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

Table 1-4 shows 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)	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

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

	(0, 0)	(0, 56)	(0, 128)	(56, 56)	(56, 128)	(128, 128)
(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), (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.

Interoperating with Pre-Release 6.5 Oracle Tuxedo Software

Table 1-5 shows 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	40	40	ERROR	ERROR	ERROR
(0, 128)	0	40	128	ERROR	128	128
(40, 40)	ERROR	40	40	ERROR	ERROR	ERROR
(40, 128)	ERROR	40	128	ERROR	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.

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*

SSL Encryption

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

Note: The actual network protocol used is TLS 1.0, which is the successor to the SSL protocol, but this document will follow common usage and refer to this protocol as SSL Encryption.

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

SSL can be used on the following types of ATMI application links:

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

How the SSL Protocol Works

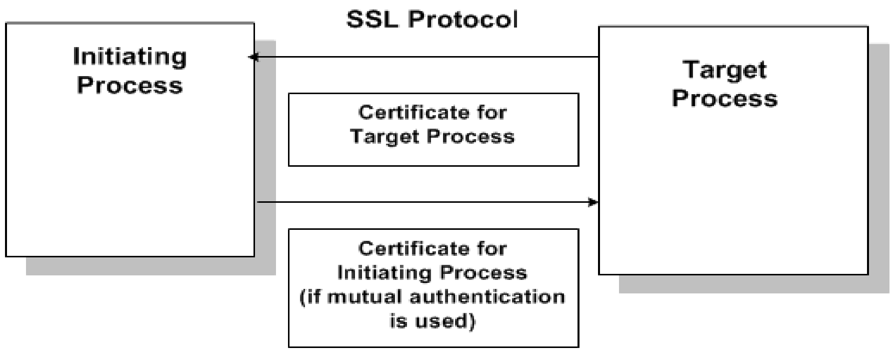
The SSL protocol works in the following manner:

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 SSL connection.

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

Figure 1-7 illustrates how the SSL protocol works.

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


SSL Protocol
Initiating Process
Target Process
Certificate for Target Process
Certificate for Initiating Process (if mutual authentication is used)

Requirements for Using the SSL Protocol

The implementation of the SSL protocol is flexible enough to fit into most public key infrastructures. Tuxedo offers two different methods to store SSL 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 SSL protocol in a Tuxedo application.

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

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.

Notes:

- 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.

Finding a Common Key Size

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

Table 1-6 shows which key size, if any, is agreed upon by two processes when all possible combinations of *`min-max`* values are negotiated. The header row holds the *`min-max`* values for one process; the far left column holds the *`min-max`* values for the other.

Table 1-6 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 1-7 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

Backward Compatibility of SSL

In order to use SSL between two Tuxedo processes, both processes must be running Tuxedo 10.0 or later (except when using the CORBA SSL capabilities described in "Using Security in CORBA Applications." It is possible to specify both non-SSL and SSL ports for WSL and JSL processes and to specify SSL 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 SSL as individual workstation clients and Tuxedo domains are upgraded to Tuxedo 10.

Notes:

- It is not possible to use SSL 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 SSL 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.

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).

Supported Cipher Suites

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

The ATMI security environment supports the cipher suites described in Table 1-8.

Table 1-8 SSL 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

SSL Installation

SSL 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.)

The `epifregedt` command can be used if this information needs to be changed after installation time.

See Also

- Security Administration Tasks
- Administering SSL Encryption
- "Distributing ATMI Applications Across a Network" and "Creating the Configuration File for a Distributed ATMI Application" on page 10-1 in *Setting Up an Oracle Tuxedo Application*
- Using Security in CORBA Applications

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

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.

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

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.

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.

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.

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

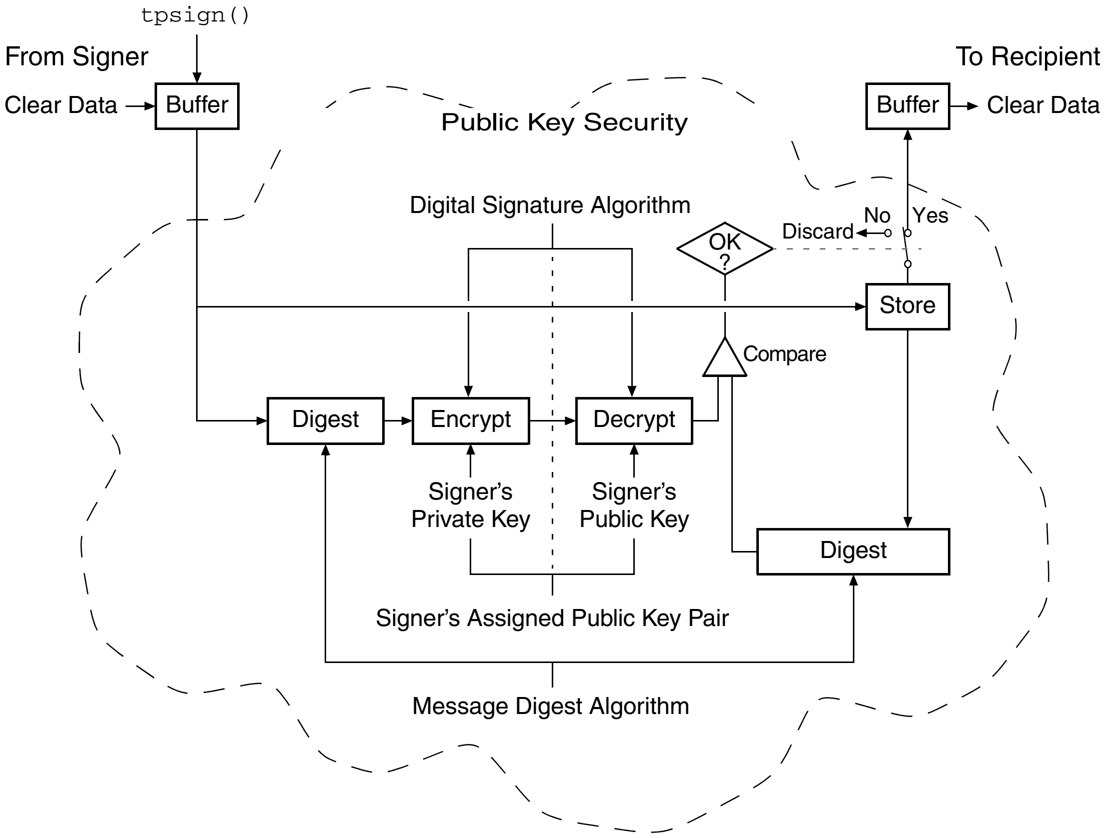
Message-based Digital Signature

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.

Figure 1-8 shows 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

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.

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 that Oracle has no plans to become a CA. By offering a public key plug-in interface, Oracle extends the opportunity to all Oracle Tuxedo customers to select a CA of their choice.

Certificate Repositories

To make a public key and its identification with a specific subscriber readily available for use in verification, the digital certificate may be published in a repository or made available by other means. 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 directly request certificates from the repository as needed.

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 will evolve with varying degrees of interoperability. In this sense, the state of PKI technology today can be viewed as similar to local and wide-area network technology in the 1980s, before there was widespread connectivity via the Internet.

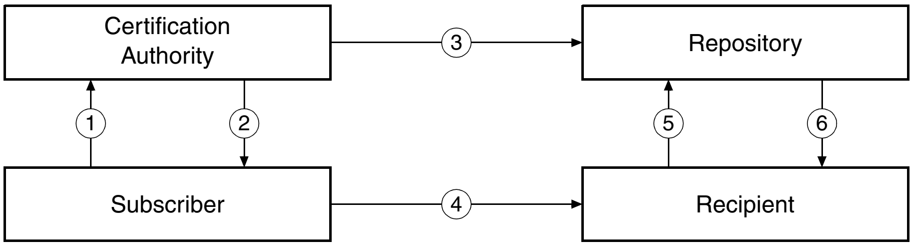
The following services are likely to be found in a PKI:

- 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

Figure 1-9 shows 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 that Oracle has no plans to become a PKI vendor. By offering a public key plug-in interface, Oracle extends the opportunity to all Oracle Tuxedo customers to use a PKI security solution with the PKI software from their vendor of choice.

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

Message-based Encryption

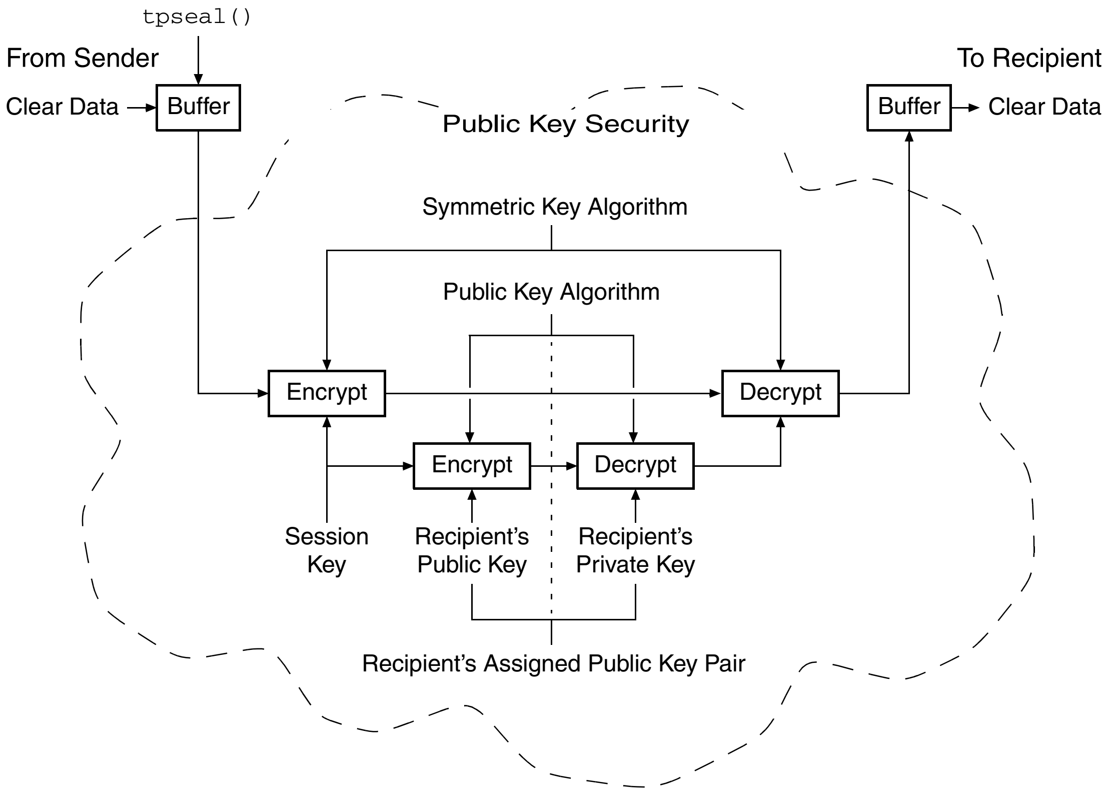
Message-based encryption keeps data private, which is essential for ATMI applications that transport data over the Internet, whether between companies or between a company and its customers. Data privacy is also critical for ATMI applications deployed over insecure internal networks.

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.

Figure 1-10 shows 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 figure does not show two other steps in this process: (1) the data is compressed immediately before the message is encrypted; and (2) the data is uncompressed immediately after the message 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

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

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*.

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.

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.

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.

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.

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.

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.

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

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 are:

- 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 Table 1-9.

Table 1-9 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 1-9 Security Levels for Default Authentication and Authorization (Continued)

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

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

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 Table 1-10.

Table 1-10 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.
`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.

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`.)

Table 1-11 shows 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 1-11 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

Table 1-11 Application Key Assignments (Continued)

At This Security Level	Messages of This Type	Are Assigned the Following Application Key
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)
	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.

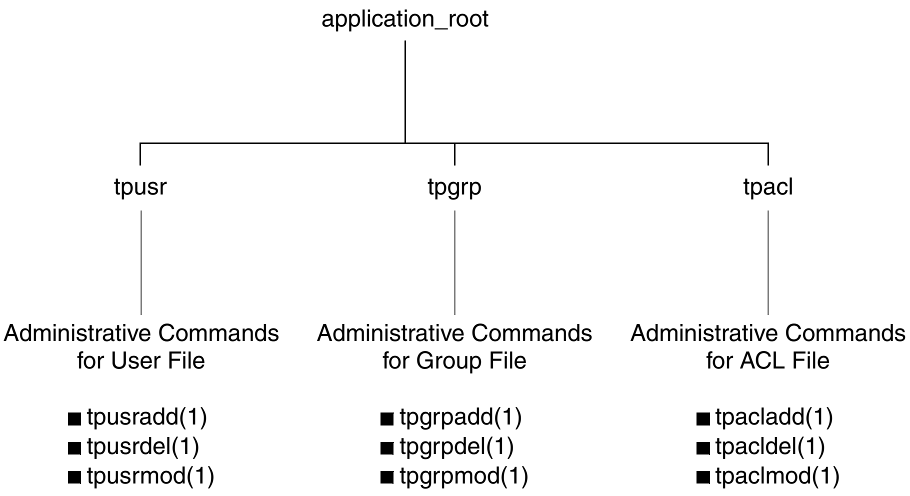
User, Group, and ACL Files

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. Figure 1-11 identifies 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 informations should 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.

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*

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. See the diagram Inter-Domain Interoperability for clarification.

- 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. See the figure Intra-Domain Interoperability for clarification.

Figure 1-12 Inter-Domain Interoperability

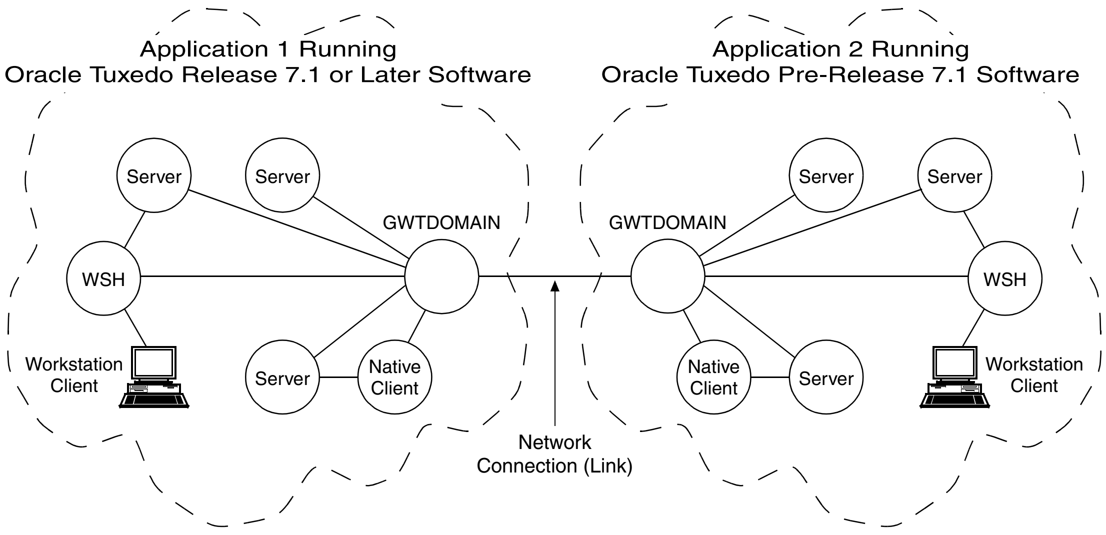
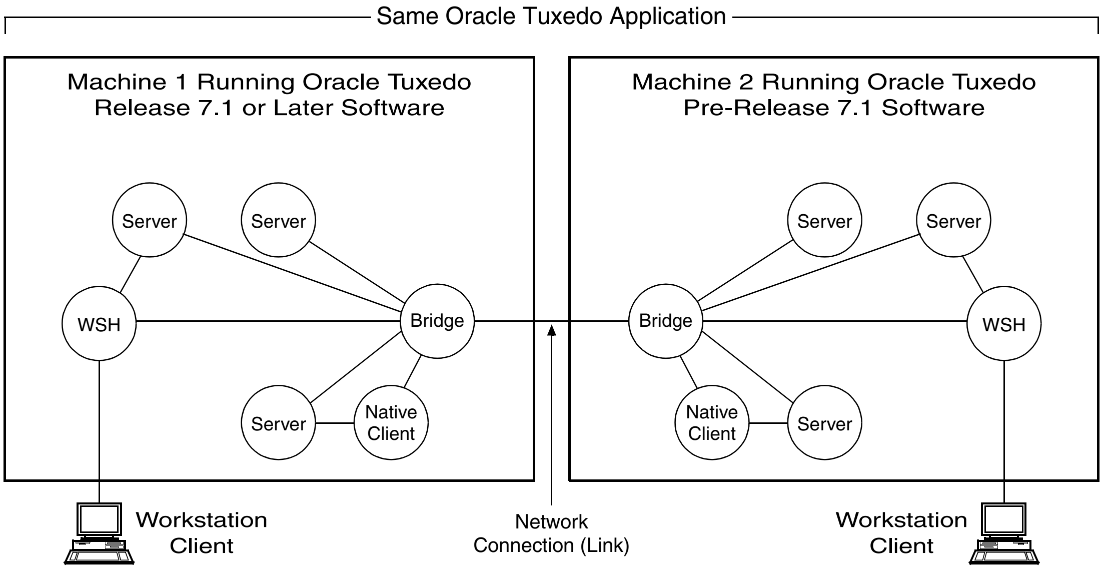


Figure 1-13 Intra-Domain Interoperability


Same Oracle Tuxedo Application
Machine 1 Running Oracle Tuxedo Release 7.1 or Later Software
Machine 2 Running Oracle Tuxedo Pre-Release 7.1 Software
Server
Server
WSH
Bridge
Server
Native Client
Bridge
Server
Server
WSH
Native Client
Server
Workstation Client
Network Connection (Link)
Workstation Client

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`.

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.

Interoperability for SSL Encryption

SSL 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 SSL encryption interoperabiliy rules is that the CORBA related SSL 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.

Interoperability for Public Key Security

The following interoperability rules for public key security shown in Table 1-12 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 1-12 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.

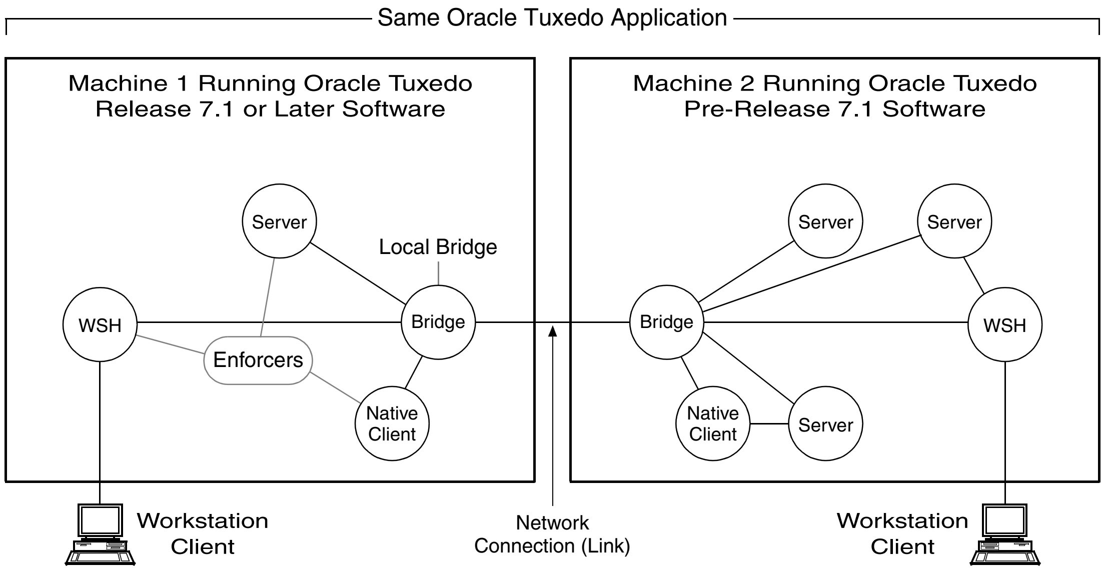
Table 1-12 Interoperability Rules for Public Key Security (Continued)

Interoperability Rule	Example	Comments
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 Figure 1-14. 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


Same Oracle Tuxedo Application
Machine 1 Running Oracle Tuxedo Release 7.1 or Later Software
Machine 2 Running Oracle Tuxedo Pre-Release 7.1 Software
Server
Local Bridge
WSH
Bridge
Enforcers
Native Client
Server
Server
Bridge
WSH
Native Client
Server
Workstation Client
Network Connection (Link)
Workstation Client

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

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

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.)

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.)

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

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.

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.

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()` 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()` 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()` message to report the failure.

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.

- 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 `tpenqueue(3c)` operation fails, and the system generates an ERROR `userlog()` 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 `tpenqueue(3c)` operation fails, and the system generates an ERROR `userlog()` 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()` operation has the side effect of destroying such a message. A transactional `tpdequeue()` operation causes transaction rollback, and all future transactional attempts to dequeue the message will continue to fail.

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.

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.

Figure 1-15 is an aid to understanding how domain gateway processes interact with local and remote ATMI applications. The table following the figure describes how release 7.1 or later domain gateway processes handle digitally signed and encrypted message buffers.

Figure 1-15 Communication Between ATMI Applications

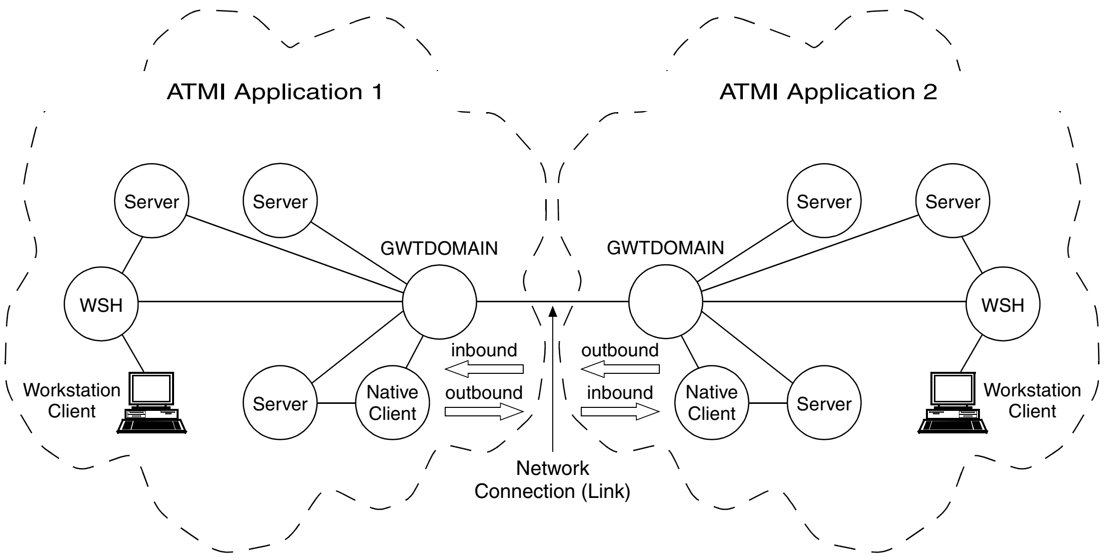


Table 1-13 Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes

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.

Table 1-13 Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes (Continued)

Message Type	Condition	Resulting Operation
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.)
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.

Table 1-13 Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes (Continued)

Message Type	Condition	Resulting Operation
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	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 the message and forwards it over the network.

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

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

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 timeusing 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)`.

Setting Up Connection Limitations/Restrictions

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

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.

Notes: 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

Listing 1-1 shows 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"
```

Listing 1-2 shows 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 connections))
- 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”
```

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."
  ```

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.

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

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

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

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 1-14 DMCONFIG File Keywords

Keyword	Option	Definition
MAC	OFF	Turn off feature. This is the default value.

Table 1-14 DMCONFIG File Keywords

	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.

Listing 1-3 shows an example DMCONFIG configuration.

Listing 1-3 DMCONFIG File Configuration

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

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 1-15 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.

Listing 1-4 and Listing 16 show examples of how to *retrieve* and *update* MAC attributes using ud32 respectively.

Listing 1-4 Sample Retrieve MAC Attribute Script

```
SRVCNM  .TMIB
TA_OPERATION    GET
TA_CLASS        T_DM_TDOMAIN
TA_DMACCESSPOINT        RDOM
TA_DMNWADDR     //host:port
```

Listing 16 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

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.

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 will only be printed 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>"
  ```

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"
```

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 funtional 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.