BEA Tuxedo

Using BEA Tuxedo Security

BEA Tuxedo Release 7.1
Document Edition 7.1
May 2000

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Using BEA Tuxedo Security

Document Edition	Date	Software Version
7.1	May 2000	BEA Tuxedo Release 7.1

Contents

1. Introducing Security

2. Administering Security

3. Programming Security

CHAPTER

1 Introducing Security

- What Security Means
- Security Plug-ins
- Security Capabilities
- Default Authentication and Authorization
- Security Interoperability

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 BEA Tuxedo system provides several security capabilities, most of which can be customized for your particular needs.

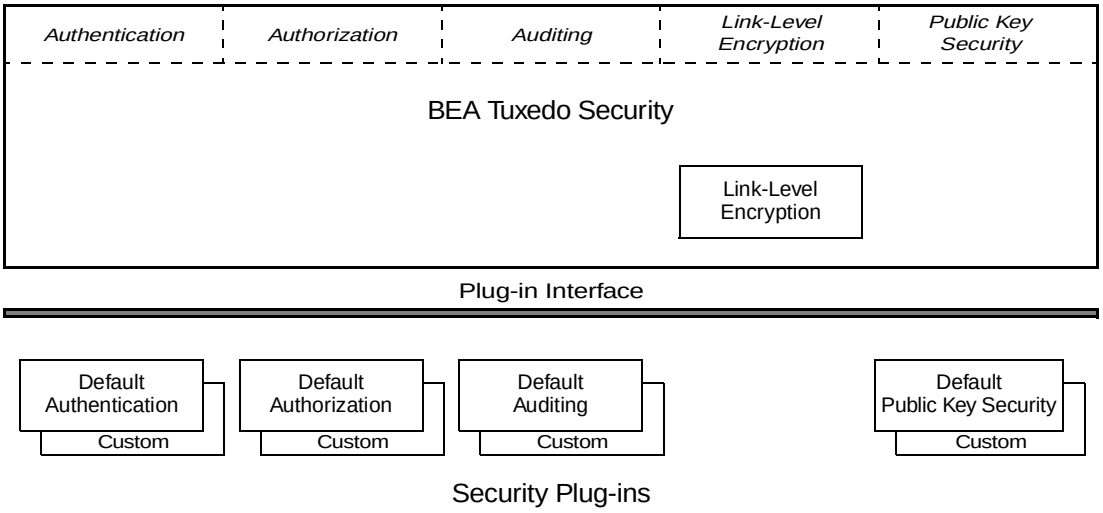
See Also

- "Security Plug-ins" on page 1-2
- "Security Capabilities" on page 1-3
- "What Administering Security Means" on page 2-1
- "What Programming Security Means" on page 3-1

Security Plug-ins

As shown in the following figure, all but one of the security capabilities available with the BEA Tuxedo system are implemented through a *plug-in interface*, which allows BEA 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 BEA Tuxedo Plug-in Security Architecture



The specifications for the security plug-in interface are not generally available, but are available to third-party security vendors who have entered into a special agreement with BEA Systems. BEA Tuxedo customers who want to customize a security capability must contact one of these vendors. For example, a BEA 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 BEA account executive.

See Also

- "Security Capabilities" on page 1-3

Security Capabilities

The BEA 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. The following table describes the security capabilities available with the BEA Tuxedo system.

Table 1-1 BEA Tuxedo 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

Table 1-1 BEA Tuxedo Security Capabilities

Security Capability	Description	Plug-in Interface	Default Implementation
Authentication	Proves the stated identity of users or system processes; safely remembers and transports identity information; and makes identity information available when needed.	Implemented as a single interface	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 BEA Tuxedo implementation of authentication has worked since it was first made available with the BEA 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 BEA Tuxedo implementation of authorization has worked since it was first made available with the BEA Tuxedo system.
Auditing	Safely collects, stores, and distributes information about operating requests and their outcomes.	Implemented as a single interface	Default auditing security is implemented by the BEA Tuxedo EventBroker and userlog (`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 a BEA Tuxedo application.	N/A	RC4 symmetric key encryption.

Table 1-1 BEA Tuxedo Security Capabilities

Security Capability	Description	Plug-in Interface	Default Implementation
Public key security	Uses public key (or asymmetric key) encryption to establish end-to-end digital signing and data privacy between BEA Tuxedo 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" on page 1-6
- "Authentication" on page 1-7
- "Authorization" on page 1-12
- "Auditing" on page 1-18
- "Link-Level Encryption" on page 1-23
- "Public Key Security" on page 1-29

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 BEA Tuxedo 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 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 application and statistics about the application are stored).

For 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" on page 2-3
- "Administering Operating System (OS) Security" on page 2-8

- "About the Configuration File" on page 2-1 and "Creating the Configuration File" on page 3-1 in *Setting Up a BEA Tuxedo Application*
- `UBBCONFIG(5)` in *BEA Tuxedo File Formats and Data Descriptions Reference*

Authentication

Authentication allows communicating processes to mutually prove identification. The BEA Tuxedo authentication plug-in interface 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 BEA Tuxedo system as easy as possible, assuming the security products have been written to the GSSAPI.

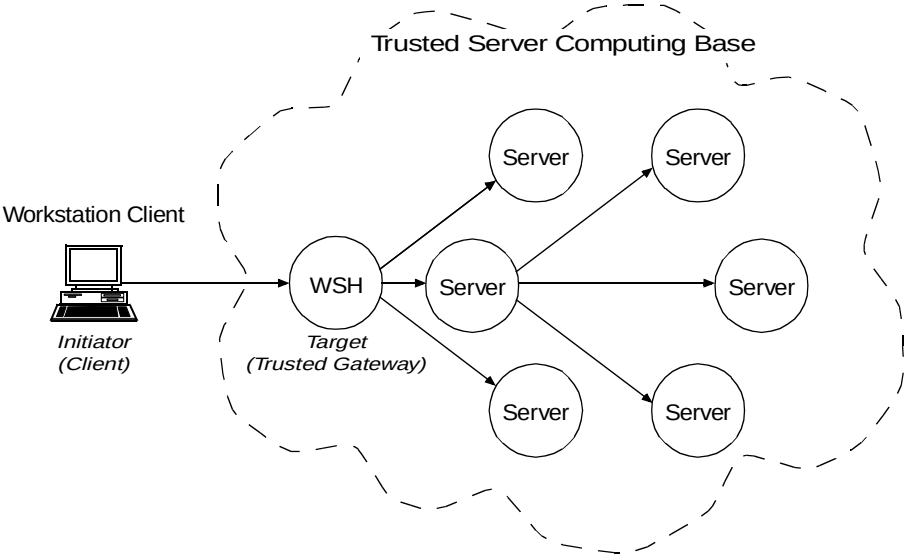
Authentication Plug-in Architecture

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 BEA 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 BEA Tuxedo system implements a *delegated trust* authentication model, as shown in the following figure.

Figure 1-2 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.

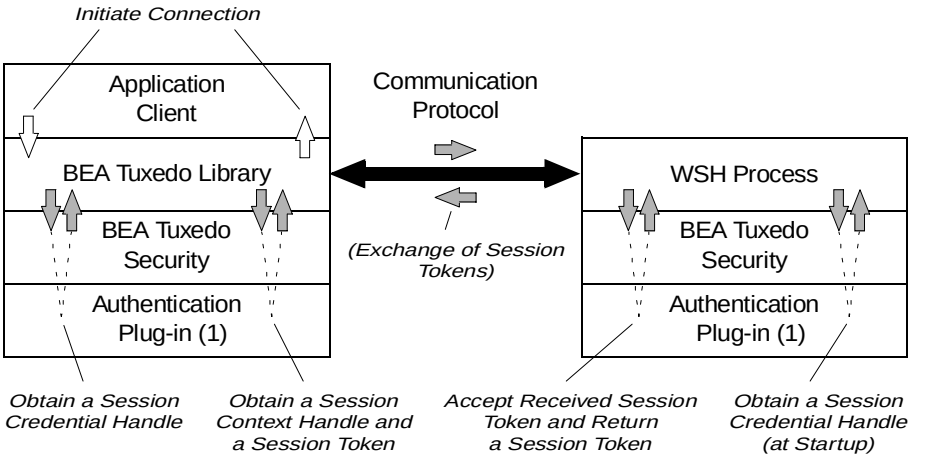
As a client request flows through a trusted gateway, the gateway attaches the client's security token to the request. The security token travels with the client's request message, and is delivered to the destination server process(es) for authorization checking and auditing purposes.

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

The following figure shows the control flow inside the BEA 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



The *initiator process* (may be thought of as a middleware client process) creates a *session context* by repeatedly calling the BEA 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 application, the BEA 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 BEA Tuxedo authentication security 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 BEA 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" on page 1-48 for more detail.

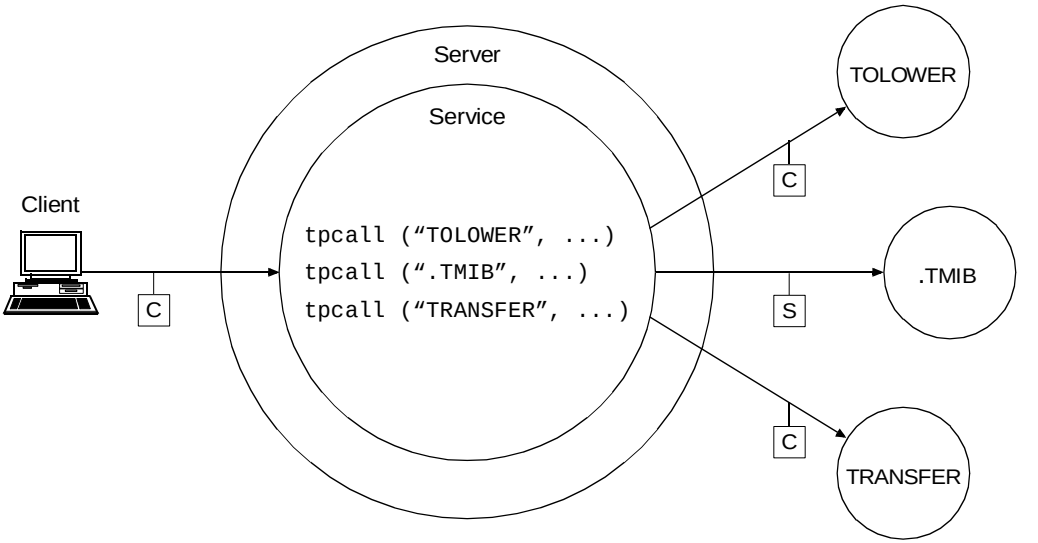
In addition, when default authentication is used, the auditing token carries the same two pieces of information: *principal name* and *application key*.

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 applications, it is useful to keep separate user identities for authorization and auditing.

Replacing Client Tokens with Server Tokens

As shown in the following diagram, 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



The feature demonstrated in the preceding diagram 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 *BEA Tuxedo File Formats and Data Descriptions Reference*.

Implementing Custom Authentication

You can provide authentication for your application by using the default plug-in or a custom plug-in. You choose a plug-in by configuring the BEA 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 BEA Tuxedo Registry” on page 2-3.

See Also

- “Default Authentication and Authorization” on page 1-44
- “Security Administration Tasks” on page 2-3
- “Administering Authentication” on page 2-9
- “Programming an Application with Security” on page 3-3
- “Writing Security Code So Client Programs Can Join the Application” on page 3-4

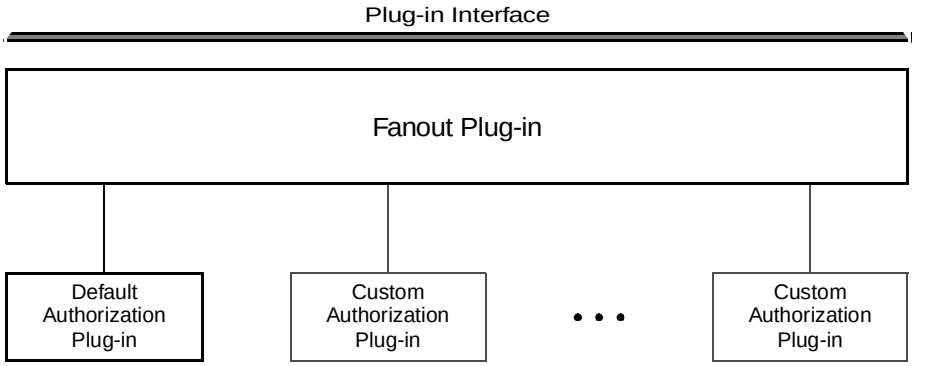
Authorization

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

Authorization Plug-in Architecture

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

Figure 1-5 Authorization Plug-in Architecture



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

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

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

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

Table 1-2 Authorization Composite Responses

If Plug-ins Return . . .	The Composite Response is . . .
All *permit* or a combination of *permit* and *abstain*	*permit*
At least one *deny*	*deny*
All *abstain*	*deny* if the `SECURITY` parameter in the application's `UBBCONFIG` file is set to `MANDATORY_ACL` *permit* if the `SECURITY` parameter is *not* set in the 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 AM and 5 PM.
- 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 AM, 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.

A BEA 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 BEA 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. A BEA Tuxedo 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 BEA 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 a BEA Tuxedo 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 BEA Tuxedo access control list (ACL), and the configured security level (optional or mandatory ACL) of the 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 BEA Tuxedo system. Specifically, the custom plug-ins may perform an additional check after an operation occurs.

When custom authorization is called by a BEA Tuxedo 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 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 BEA Tuxedo 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 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 application by using the default plug-in or adding one or more custom plug-ins. You choose a plug-in by configuring the BEA 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 BEA Tuxedo Registry" on page 2-3.

See Also

- "Default Authentication and Authorization" on page 1-44
- "Security Administration Tasks" on page 2-3
- "Administering Authorization" on page 2-34
- "Programming an Application with Security" on page 3-3

Auditing

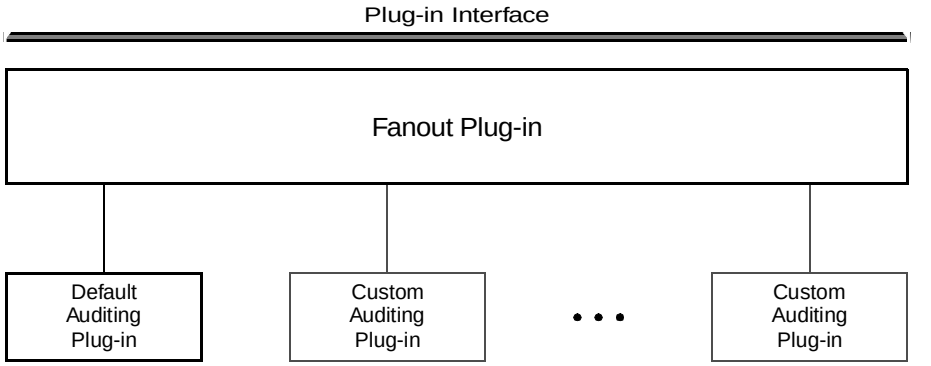
Auditing provides a means to collect, store, and distribute information about operating requests and their outcomes. Audit-trail records may be used to determine which principals performed, or attempted to perform, actions that violated BEA Tuxedo security. They may also be used to determine which operations 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 the following diagram, 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” on page 1-13.

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.

A BEA Tuxedo 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 BEA Tuxedo 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, a BEA Tuxedo 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. A BEA Tuxedo 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 BEA 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 BEA 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 a BEA Tuxedo 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 BEA Tuxedo system. Specifically, the custom plug-ins may perform an additional audit before an operation occurs.

When custom auditing is called by a BEA Tuxedo 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 BEA Tuxedo 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 application by using the default plug-in or adding one or more custom plug-ins. You choose a plug-in by configuring the BEA 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 "Setting the BEA Tuxedo Registry" on page 2-3.

Link-Level Encryption

Link-level encryption (LLE) establishes data privacy for messages moving over the network links that connect the machines in a BEA Tuxedo 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 BEA Tuxedo system encrypts data before sending it over a network link and decrypts it as it comes off the link. The system repeats this encryption/decryption process at every link through which the data passes. For this reason, LLE is referred to as a point-to-point facility.

LLE can be used on the following types of BEA Tuxedo 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

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

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

Finding a Common Key Size

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

The following table 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 Inter-process 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 BEA Tuxedo system offers some backward compatibility for LLE.

Interoperating with Release 6.5 BEA Tuxedo Software

The following table shows which key size, if any, is agreed upon by two BEA Tuxedo processes 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 BEA Tuxedo Software

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

If your current BEA Tuxedo installation is configured for (0, 56), (0, 128), (56, 56), or (56, 128), and you want to interoperate with a Release 6.5 BEA Tuxedo system 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 BEA Tuxedo Software

The following table shows which key size, if any, is agreed upon by two BEA Tuxedo processes 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 BEA 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 BEA Tuxedo installation is configured for (0, 56) or (0, 128), and you want to interoperate with a pre-Release 6.5 BEA Tuxedo system that is configured for a maximum LLE level of 40 bits, then the result of any negotiation is 40.

If your current BEA Tuxedo installation is configured for (56, 56), (56, 128), or (128, 128), then your system *cannot* interoperate with a pre-Release 6.5 BEA Tuxedo system 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 application not using LLE, and 60 seconds in an application using LLE. The 60-second interval includes the time needed to negotiate an encrypted link. This time limit can be changed when LLE is configured by changing

the value of the MAXINITTIME parameter for the Workstation Listener (WSL) server in the UBBCONFIG file, or the value of the TA_MAXINITTIME attribute in the T_WSL class of the WS_MIB(5).

LLE Installation and Licensing

As part of the BEA Tuxedo system, LLE software is delivered on the BEA Tuxedo CD-ROM. If you have a BEA Tuxedo Release 7.1 license to use LLE in the United States and Canada, you can use 56-bit or 128-bit encryption. If you have a license to use LLE on a BEA Tuxedo system outside the United States and Canada, you can use 56-bit encryption.

All BEA Tuxedo licenses are stored in the $TUXDIR/udataobj/lic.txt file on a UNIX host machine, or in the %TUXDIR%\udataobj\lic.txt file on a Windows NT host machine.

The following listing is an excerpt from a sample license file for running LLE in the United States and Canada.

```
[BEA Tuxedo]
VERSION=7.1
LICENSEE=ACME CORPORATION
SERIAL=155566678
ORDERID=
USERS=1000
EXPIRATION=2000-01-31
SIGNATURE=TXmtx+AhQdJgr3sjjznBqRB7SP9Jgr3UzAKctjz+e6RmsFSAhUAhStj
   znBQdL9n=

[LINK ENCRYPTION]
VERSION=7.1
LICENSEE=ACME CORPORATION
SERIAL=155566678
ORDERID=
USERS=1000
STRENGTH=128
EXPIRATION=2000-01-31
SIGNATURE=TXUAhSPnx2C9kMC0CFG+e6Rgr3UzmsFKRBPdJASAhU7KctjznBqFQsj
   jznBdh0h=
   .
   .
   .
```

See Also

- "Security Administration Tasks" on page 2-3
- "Administering Link-Level Encryption" on page 2-35
- "Distributing Applications Across a Network" on page 6-1 and "Creating the Configuration File for a Distributed Application" on page 7-1 in *Setting Up a BEA Tuxedo Application*

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. BEA Tuxedo public key 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 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

BEA Tuxedo public key security 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

BEA Tuxedo public key security 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 BEA Tuxedo public key security restricts the key length to 128 bits.

BEA 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.

Public Key Installation and Licensing

As part of the BEA Tuxedo system, the software for message-based digital signature and message-based encryption is delivered on the BEA Tuxedo CD-ROM, but cannot be used without a separate license. All BEA Tuxedo licenses are in the `$TUXDIR/udataobj/lic.txt` file on a UNIX host machine, or in the `%TUXDIR%\udataobj\lic.txt` file on a Windows NT host machine.

The following listing is an excerpt from a sample license file for message-based digital signature and message-based encryption.

```
[BEA Tuxedo]
VERSION=7.1
LICENSEE=ACME CORPORATION
SERIAL=155566678
ORDERID=
USERS=1000
EXPIRATION=2000-01-31
SIGNATURE=TXmtx+AhQdJgr3sjjznBqRB7SP9Jgr3UzAKctjz+e6RmsFSAhUAhStj
   znBQdL9n=
   .
   .
   .

[PK ENCRYPTION]
VERSION=7.1
```

```
LICENSEE=ACME CORPORATION
SERIAL=155566678
ORDERID=
USERS=1000
STRENGTH=128
EXPIRATION=2000-01-31
SIGNATURE=TX0CFHkaBpKpAlXGEtQqi+/jJvMo1VB9AhUAUAkizwsgYefRwQJDNTF
   0205b1ik=

[PK SIGNATURE]
VERSION=7.1
LICENSEE=ACME CORPORATION
SERIAL=155566678
ORDERID=
USERS=1000
STRENGTH=128
EXPIRATION=2000-01-31
SIGNATURE=TX0CiqA5FCAXJFXUEGvAki+gL+i09eRep9hYdshS/8a70MIJQChUAk9
   zIAhUIH4=
```

See Also

- “Message-based Digital Signature” on page 1-34
- “Message-based Encryption” on page 1-39
- “Public Key Implementation” on page 1-41
- “Security Administration Tasks” on page 2-3
- “Administering Public Key Security” on page 2-41
- “Programming an Application with Security” on page 3-3
- “Writing Security Code to Protect Data Integrity and Privacy” on page 3-15

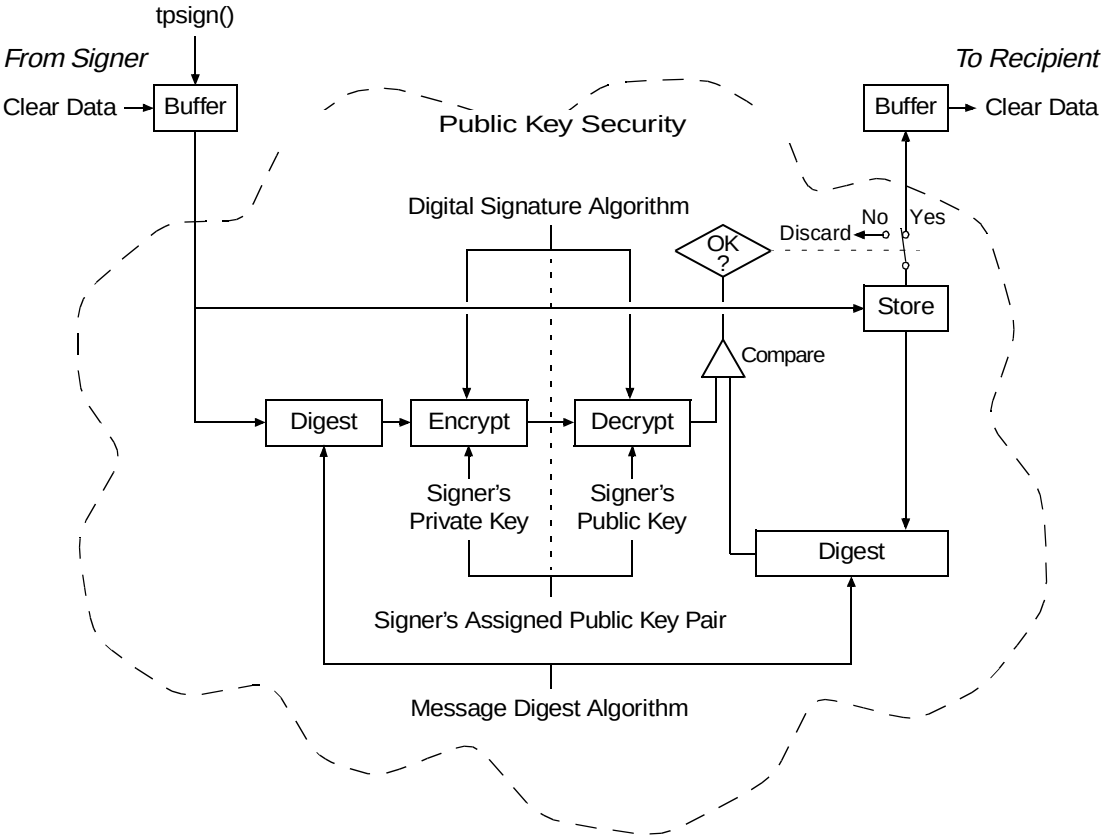
Message-based Digital Signature

Message-based digital signatures enhance BEA Tuxedo 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 most applications that transport data over the Internet, either between companies or between a company and the general public. It also is critical for applications deployed over insecure internal networks.

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

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

Figure 1-7 BEA Tuxedo 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 email 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 application complying with X.509. BEA Tuxedo public key security 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 certification authority 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 BEA Systems has no plans to become a CA. By offering a public key plug-in interface, BEA Systems extends the opportunity to all BEA 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.

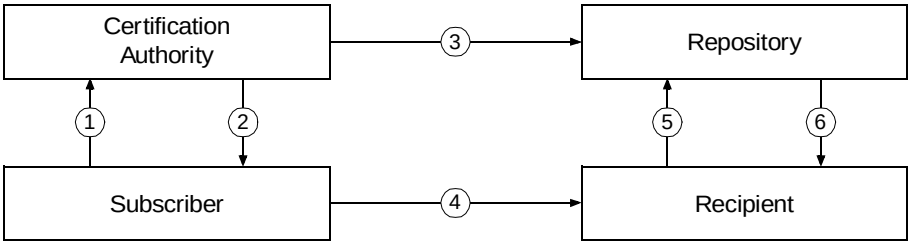
There is no single standard public key infrastructure today, though efforts are underway to define one. It is not yet clear whether a standard will be established or multiple independent PKIs will evolve with varying degrees of interoperability. In this sense, the state of PKI technology today can be 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

The following diagram shows the PKI process flow.

Figure 1-8 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 BEA Systems has no plans to become a PKI vendor. By offering a public key plug-in interface, BEA Systems extends the opportunity to all BEA Tuxedo customers to use a PKI security solution with the PKI software from their vendor of choice.

See Also

- "Public Key Implementation" on page 1-41
- "Security Administration Tasks" on page 2-3
- "Administering Public Key Security" on page 2-41
- "Programming an Application with Security" on page 3-3
- "Writing Security Code to Protect Data Integrity and Privacy" on page 3-15

Message-based Encryption

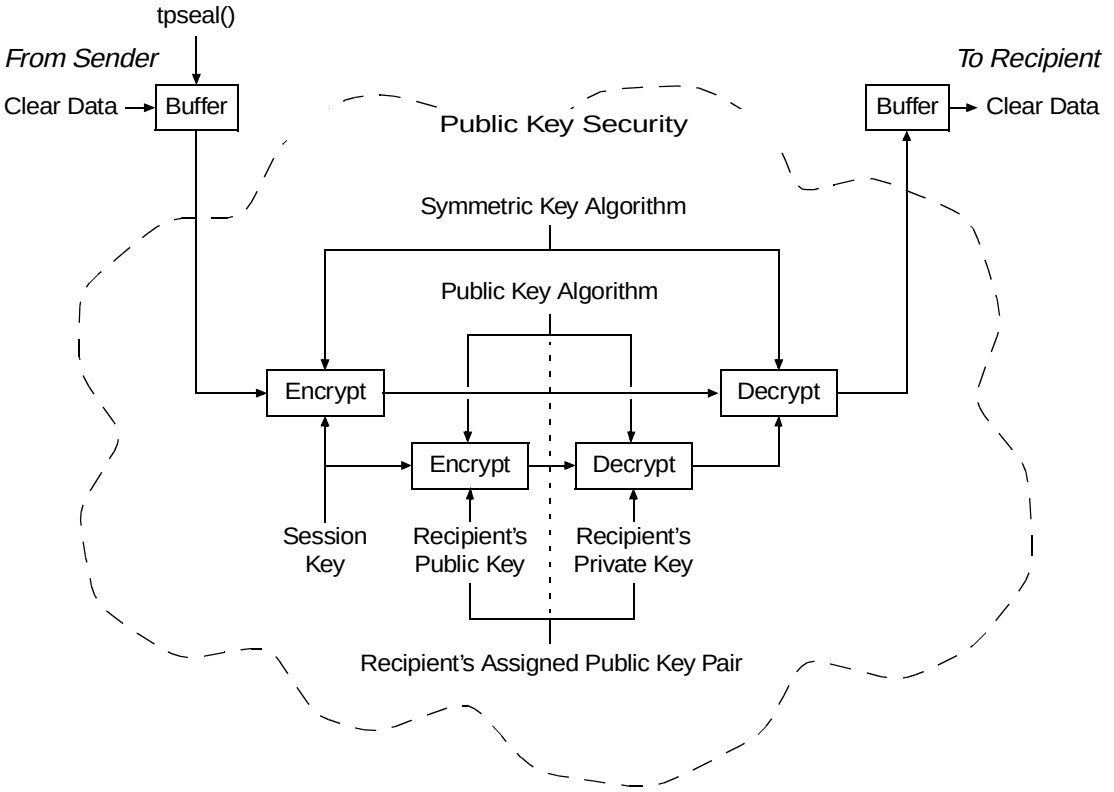
Message-based encryption keeps data private, which is essential for most applications that transport data over the Internet, whether between companies or between a company and its customers. Data privacy is also critical for applications deployed over insecure internal networks.

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 inter-server network links.

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

Figure 1-9 BEA Tuxedo 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 a BEA Tuxedo message buffer, message-based encryption is compatible with all existing BEA Tuxedo 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" on page 1-41
- "Security Administration Tasks" on page 2-3
- "Administering Public Key Security" on page 2-41
- "Programming an Application with Security" on page 3-3
- "Writing Security Code to Protect Data Integrity and Privacy" on page 3-15

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 applications.

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 BEA 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 application by using custom plug-ins. You choose a plug-in by configuring the BEA 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 BEA Tuxedo Registry" on page 2-3.

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” on page 1-29
- “Security Administration Tasks” on page 2-3
- “Administering Public Key Security” on page 2-41
- “Programming an Application with Security” on page 3-3
- “Writing Security Code to Protect Data Integrity and Privacy” on page 3-15

Default Authentication and Authorization

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

An application administrator can use the default authentication and authorization plug-ins to configure an 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 the following table.

Table 1-6 Security Levels for Default Authentication and Authorization

Security Level	Description
No authentication	Clients do not have to be verified before joining the application. When joining an 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 application, and clients must provide the password to join the application. When successfully joining an 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 user name and user-specific data, such as a password, to join the application. When successfully joining an application at this security level, a user has access to all application resources.
Optional ACL security	Clients must provide the application password, a user name, and user-specific data such as a password. For a user who successfully joins an 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 application is set to optional ACL security, all users *are* permitted to access the resource.

Table 1-6 Security Levels for Default Authentication and Authorization

Security Level	Description
Mandatory ACL security	Clients must provide the application password, a user name, and user-specific data such as a password. For a user who successfully joins an 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 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. A BEA Tuxedo 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 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 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 *user name* 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 BEA Tuxedo application administrator, and tpsysop is treated as the BEA Tuxedo application operator.

User-Client Names

When an authenticated client joins an application, it passes a user name 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 user name and client name, as well as other security-related fields in the TPINIT buffer/ TPINFDEF-REC record, are described in the following table.

Table 1-7 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 user name represents the caller.

* The binary equivalent of the UBBCONFIG file.

** Usually a user password.

Table 1-7 Security-Related Fields in TPINIT Buffer/ TPINFDEF-REC Record

TPINIT	TPINFDEF-REC	Description
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 user name, client name, and user-specific data are transferred to AUTHSVR without interpretation by the BEA 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 application, it is assigned a 32-bit application key by the BEA Tuxedo system. The client cannot reset the key other than by terminating its association and joining the 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 *BEA Tuxedo C Function Reference* for more information about TPSVCINFO.)

The following table 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 1-8 Application Key Assignments

At this security level	Messages of this type	Are assigned the following application key
Any security level	Messages from native BEA Tuxedo-provided clients that must be run by the administrator (like `tmadmin(1)`)	0x80000000 (application key of the administrator)
`NONE` or `APP_PW`	Messages from native 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 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 client other than `tpsysadm` or `tpsysop`	-1
`USER_AUTH`, `ACL`, or `MANDATORY_ACL`	Messages from native 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* clients that call `tpinit()`/ `TPINITIALIZE()` with a client name of `tpsysadm`	0x80000000 (application key of the administrator)
	Messages from *authenticated* clients that call `tpinit()`/ `TPINITIALIZE()` with a client name of `tpsysop`	0xC0000000 (application key of the operator)
	Messages from *authenticated* 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" on page 1-11.

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.

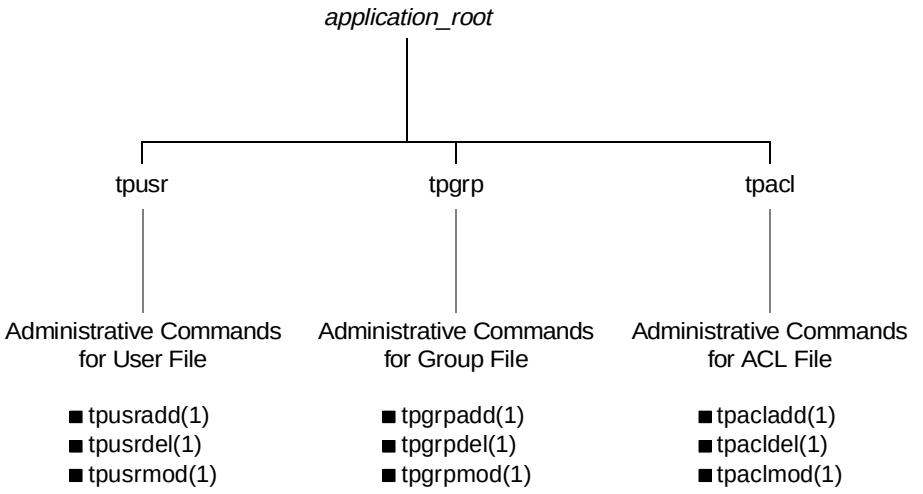
User, Group, and ACL Files

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 path name defined for the `APPDIR` variable. The following figure identifies these files and specifies the administrative commands available for controlling each list.

Figure 1-10 Default User, Group, and ACL Files



Note: For a BEA Tuxedo system 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 administer is allowed to use these commands.

An application administer 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 application.

Optional and Mandatory ACLs

The ACL and MANDATORY_ACL security levels constitute the default authorization implementation for the BEA Tuxedo system.

When the security level is ACL, if there is no entry in the tpacl file 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 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 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 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 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” on page 2-1
- “Security Administration Tasks” on page 2-3

- "Administering Authentication" on page 2-9
- "Administering Authorization" on page 2-34
- "What Programming Security Means" on page 3-1
- "Programming an Application with Security" on page 3-3
- "Writing Security Code So Client Programs Can Join the Application" on page 3-4
- "About the Configuration File" on page 2-1 and "Creating the Configuration File" on page 3-1 in *Setting Up a BEA Tuxedo Application*
- `UBBCONFIG(5)` in *BEA Tuxedo File Formats and Data Descriptions Reference*
- `AUTHSVR(5)` in *BEA Tuxedo File Formats and Data Descriptions Reference*

Security Interoperability

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

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

- Inter-domain interoperability

  Involves one BEA Tuxedo application running BEA Tuxedo Release 7.1 or later software, and another BEA Tuxedo application running BEA Tuxedo pre-Release 7.1 software. See the diagram "Inter-Domain Interoperability" on page 1-54 for clarification.

- Intra-domain interoperability

  Involves one machine in a multiple-machine BEA Tuxedo application running BEA Tuxedo Release 7.1 or later software, and another machine in the same

application running BEA Tuxedo pre-Release 7.1 software. See the diagram "Intra-Domain Interoperability" on page 1-55 for clarification.

Figure 1-11 Inter-Domain Interoperability

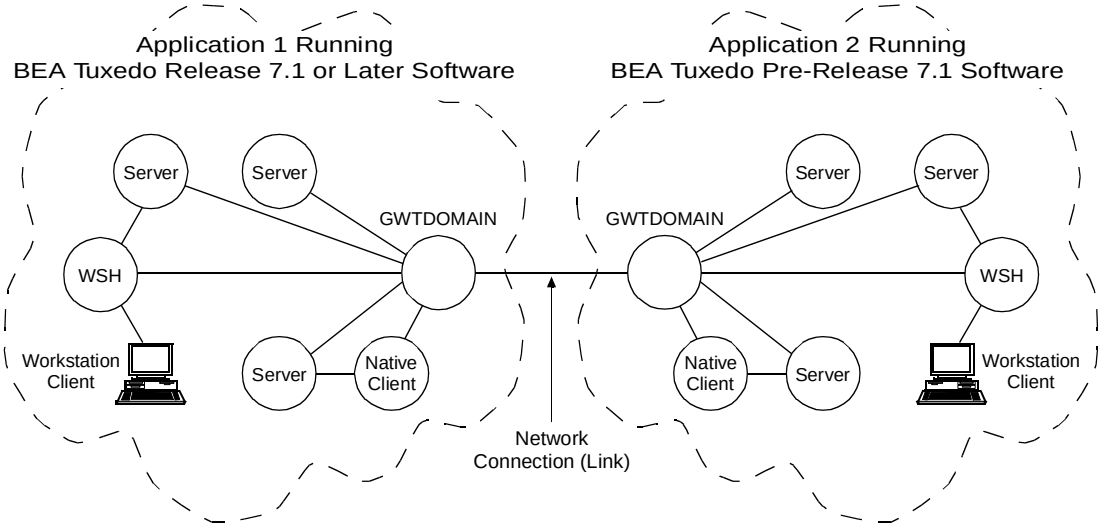
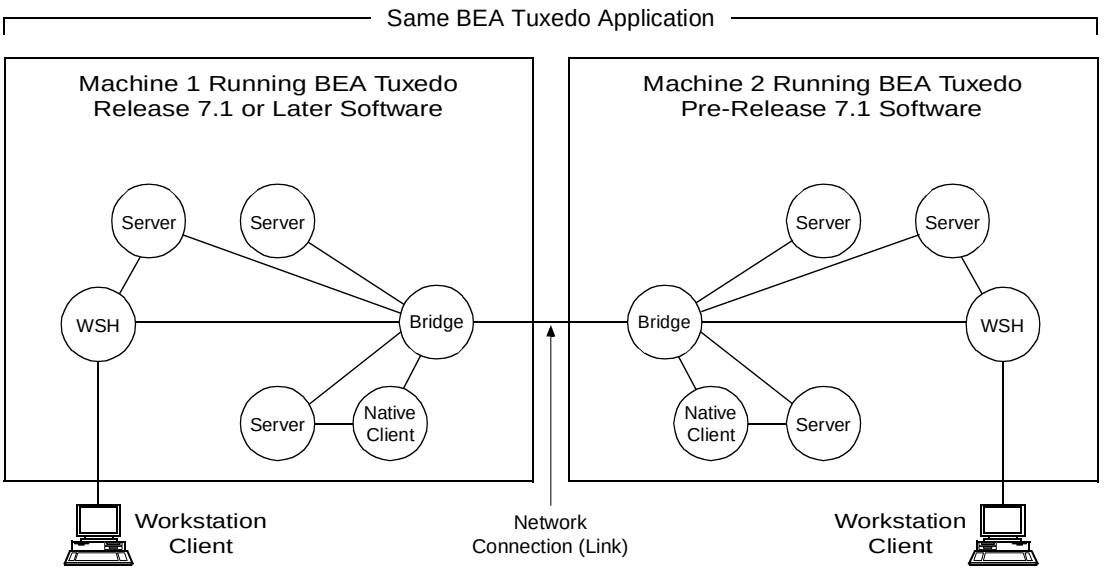


Figure 1-12 Intra-Domain Interoperability



Interoperating with Pre-Release 7.1 Software

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

By default, interoperability with a machine running BEA 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 application to interoperate with BEA Tuxedo pre-Release 7.1 software. “Mandating Interoperability Policy” on page 2-15 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 BEA 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 BEA Tuxedo pre-Release 7.1 software is described in "Backward Compatibility of LLE" on page 1-26.

Interoperability for Public Key Security

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

Table 1-9 Interoperability Rules for Public Key Security

Interoperability Rule	Example	Comments
Encrypted outgoing message buffers destined for a machine running BEA 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 a BEA 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" on page 2-47 for a description of the `ENCRYPTION_REQUIRED` configuration parameter.

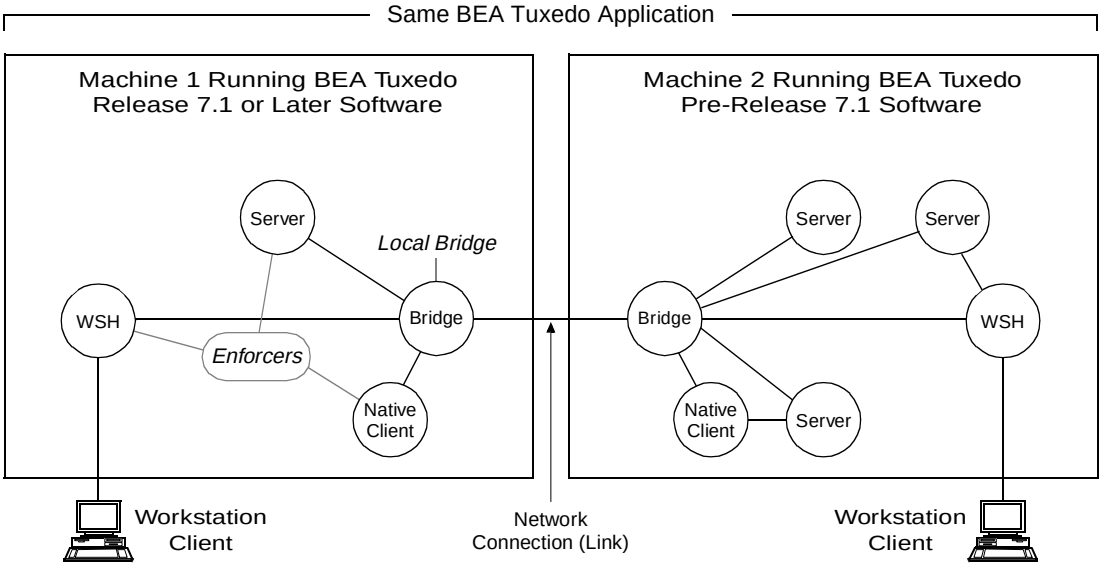
Table 1-9 Interoperability Rules for Public Key Security

Interoperability Rule	Example	Comments
For outgoing message buffers destined for the machine running BEA 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 BEA 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" on page 2-42 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 the following diagram. A bridge process operates only as a *conduit*; it does *not* decrypt message buffer content or verify digital signatures.

Figure 1-13 Enforcing Intra-Domain Interoperability Rules for Public Key Security



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

See Also

- "Security Compatibility" on page 1-59
- "Mandating Interoperability Policy" on page 2-15
- "Setting Digital Signature Policy" on page 2-42
- "Setting Encryption Policy" on page 2-47

Security Compatibility

For an application running BEA 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 user name, or *principal name*, and (2) an application key value as defined in "Application Key" on page 1-48.

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" on page 1-7 and "Authorization" on page 1-12 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 user name, or *principal name*, and (2) an application key value as defined in "Application Key" on page 1-48.

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" on page 1-7 and "Auditing" on page 1-18 for more detail.)

Compatibility Issues for Public Key Security

Public key security is compatible with all features and processes supported by BEA 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 BEA Tuxedo 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 BEA Tuxedo 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 BEA Tuxedo 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 BEA 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" on page 2-47).

  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" on page 2-42).

  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 BEA 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" on page 2-47).

  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" on page 2-42).

  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 BEA Tuxedo applications running BEA Tuxedo Release 7.1 or later software preserve digital signatures and encryption envelopes. In addition, the domain gateway processes verify digital signatures and enforce administrative system policies regarding digital signatures and encryption.

The following diagram is an aid to understanding how domain gateway processes interact with local and remote BEA Tuxedo applications. The table following the diagram describes how Release 7.1 or later domain gateway processes handle digitally signed and encrypted message buffers.

Figure 1-14 Communication Between BEA Tuxedo Applications

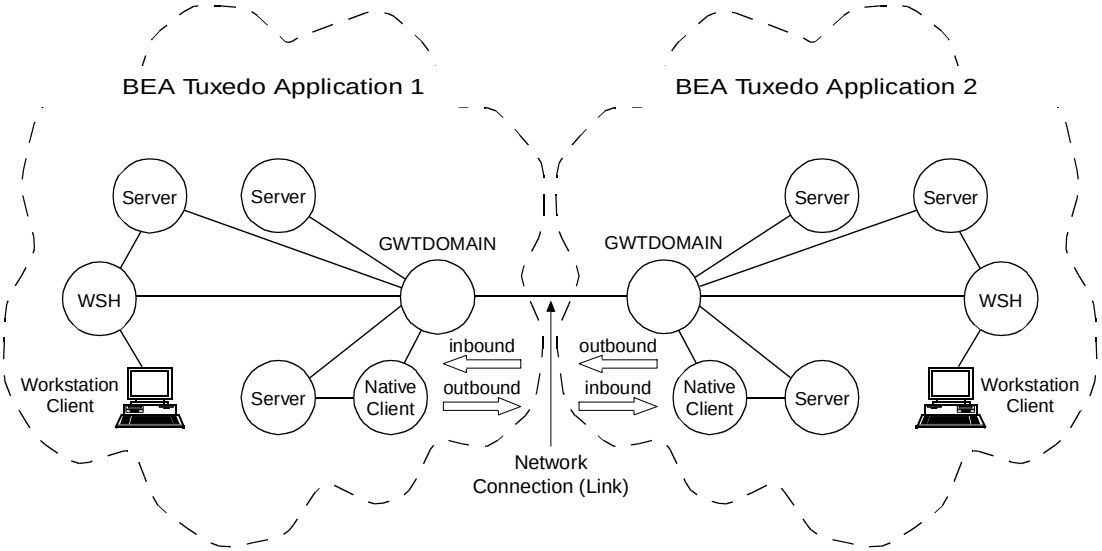


Table 1-10 Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes

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" on page 1-60 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" on page 2-47 for clarification.) If the domain gateway does *not* require encryption, the gateway process accepts and forwards the message.
Inbound message	Has digital signature but is not encrypted	The domain gateway process verifies the digital signature and forwards the message with digital signature attached.
Inbound message	Does not have digital signature and is not encrypted	If the domain gateway process is running within a domain, machine, or group *requiring* digital signatures, the gateway process rejects the message. If a service advertised by the domain gateway *requires* digital signatures, the gateway process rejects the message. (See "Setting Digital Signature Policy" on page 2-42 for clarification.) If the domain gateway does *not* require digital signatures, the gateway process accepts and forwards the message.

Table 1-10 Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes

Message Type	Condition	Resulting Operation
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" on page 1-60 for clarification.) If the encrypted message is destined for a process running BEA 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" on page 1-55 and "Interoperability for Public Key Security" on page 1-56 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" on page 2-47 for clarification.) If the domain gateway does *not* require encryption, the gateway process accepts the message and forwards it over the network.
Outbound message	Has digital signature but is not encrypted	The domain gateway process verifies the digital signature and forwards the message with digital signature attached over the network. If the message is destined for a process running BEA Tuxedo pre-Release 7.1 software *and assuming interoperability with BEA 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" on page 1-55 and "Interoperability for Public Key Security" on page 1-56 for clarification.)

Table 1-10 Operation of Release 7.1 or Later Domain Gateway (GWTDOMAIN) Processes

Message Type	Condition	Resulting Operation
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" on page 2-42 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 BEA Tuxedo 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 BEA Tuxedo application. As an example, if encryption and/or digital signatures are required at the domain level for the BEA Tuxedo application, the local domain gateway process rejects any message coming from the other vendor's gateway process.

See Also

- “Security Interoperability” on page 1-53
- “Mandating Interoperability Policy” on page 2-15
- “Setting Digital Signature Policy” on page 2-42
- “Setting Encryption Policy” on page 2-47

CHAPTER

2 Administering Security

- What Administering Security Means
- Security Administration Tasks
- Setting the BEA Tuxedo Registry
- Configuring an Application for Security
- Setting Up the Administration Environment
- Administering Default Authentication and Authorization

What Administering Security Means

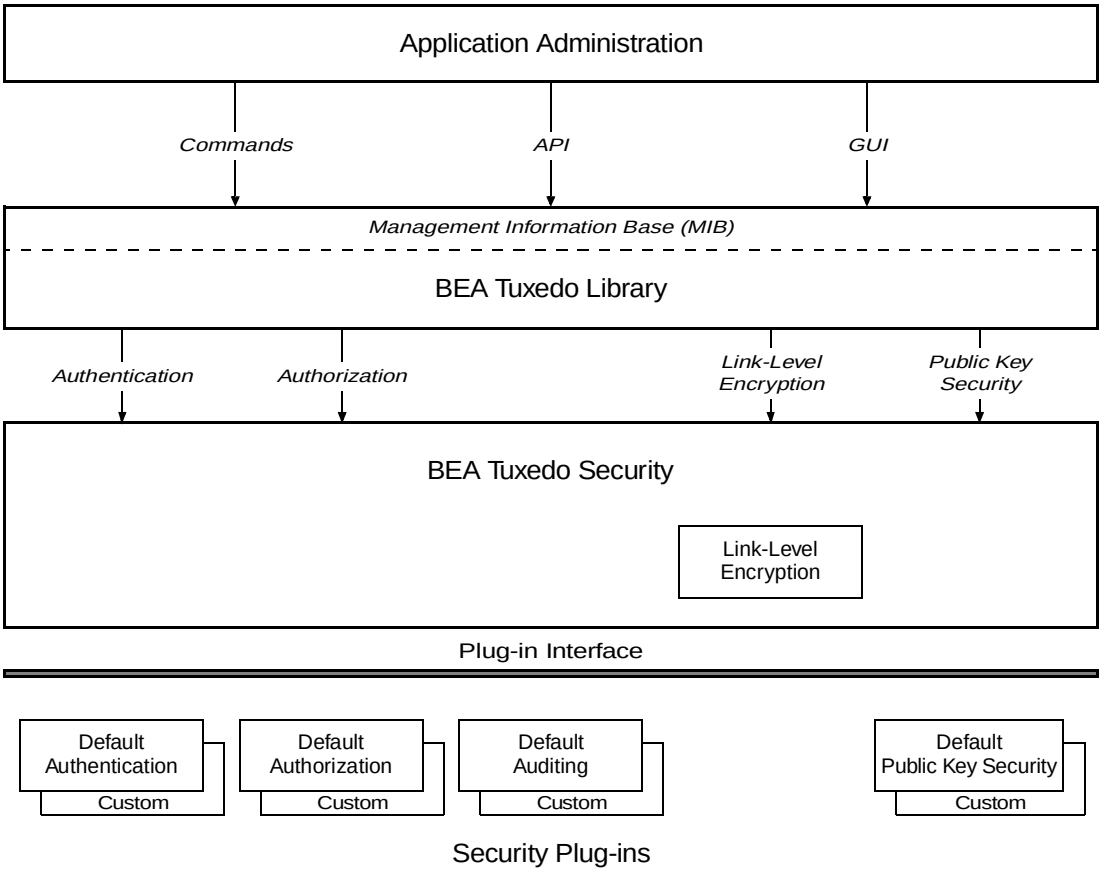
Administering security for a BEA Tuxedo application involves setting and enforcing security policies for the components of the application, including its clients, server machines, and gateway links. The application administrator sets the security policies for the application, and the BEA Tuxedo system upon which the application is built enforces those policies.

The BEA Tuxedo system offers the following security capabilities:

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

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

Figure 2-1 Administering BEA Tuxedo Security



See Also

- “Security Administration Tasks” on page 2-3
- “What Security Means” on page 1-1

- “What Programming Security Means” on page 3-1

Security Administration Tasks

Security administration consists of the following tasks:

- Setting the BEA Tuxedo registry
- Configuring an application for security
- Setting up the administration environment
- Administering operating system (OS) security
- Administering authentication
- Administering authorization
- Administering link-level encryption
- Administering public key security

See Also

- “Setting the BEA Tuxedo Registry” on page 2-3

Setting the BEA Tuxedo Registry

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

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

Purpose of the BEA Tuxedo Registry

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

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

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

Every Workstation client and server machine in a BEA Tuxedo application must use the same set of plug-in modules.

Registering Plug-ins

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

Three commands are available for administering plug-ins:

- `epifreg`—for registering a plug-in

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

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

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

See Also

- "Configuring an Application for Security" on page 2-5

Configuring an Application for Security

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

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

- Editing the configuration file (`UBBCONFIG`)
- Changing the `TM_MIB`, or
- Using the BEA Administration Console

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

Editing the Configuration File

You can edit the UBBCONFIG configuration file to set security policies for a BEA Tuxedo application. The UBBCONFIG configuration file may have any file name, as long as the content of the file conforms to the format described on the UBBCONFIG(5) reference page in *BEA Tuxedo File Formats and Data Descriptions Reference*.

For more details about UBBCONFIG and its binary equivalent, TUXCONFIG, see "About the Configuration File" on page 2-1 and "Creating the Configuration File" on page 3-1 in *Setting Up a BEA Tuxedo Application*.

Changing the TM_MIB

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

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

For more information about BEA Tuxedo MIBs, start with MIB(5) in *BEA Tuxedo File Formats and Data Descriptions Reference*. Also, see "Managing Operations Using the MIB" on page 3-10 in *Introducing the BEA Tuxedo System*.

Using the BEA Administration Console

You can also use the BEA Administration Console to change security policies for a BEA Tuxedo application. The BEA Administration Console is a Web-based tool used to configure, monitor, and dynamically re-configure an application.

For details about the BEA Administration Console, see "Using the BEA Administration Console" on page 3-4 in *Introducing the BEA Tuxedo System*.

See Also

- "Setting Up the Administration Environment" on page 2-7

Setting Up the Administration Environment

The application administrator defines certain environment variables for a BEA Tuxedo application as part of configuring the application. The values defined for the variables are absolute path names that reference BEA Tuxedo executables and data libraries.

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

For details on setting up the administration environment, see "How to Set Your Environment" on page 1-2 in *Administering a BEA Tuxedo Application at Run Time*.

See Also

- "Administering Operating System (OS) Security" on page 2-8
- "Administering Authentication" on page 2-9
- "Administering Authorization" on page 2-34
- "Administering Link-Level Encryption" on page 2-35
- "Administering Public Key Security" on page 2-41
- "Security Administration Tasks" on page 2-3

Administering Operating System (OS) Security

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

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

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

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

Recommended Practices for OS Security

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

- Limit access to files and IPC resources to the application administrator.
- Have "trusted" client programs run only with the permissions of the administrator (using a `setuid` utility).

- For maximum security on your operating system, allow only Workstation clients to access the application; client programs should not be allowed to run on the same machines on which application servers and administrative programs run.
- Combine all of these practices with BEA Tuxedo system security so that the application can identify any client making a request.

See Also

- "Operating System (OS) Security" on page 1-6
- "Security Administration Tasks" on page 2-3

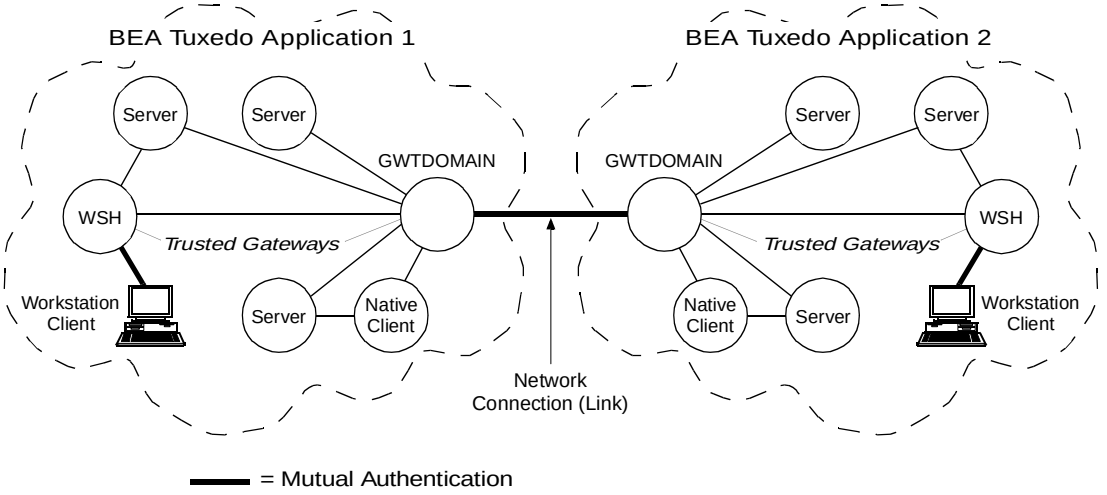
Administering Authentication

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

Except for the configuration instructions identified in this topic, the procedures for administering authentication depend upon the underlying authentication system of the application. For procedures to administer a custom authentication system, see the documentation for that system. For procedures to administer the default authentication system, see "Administering Default Authentication and Authorization" on page 2-56.

The following diagram demonstrates the use of the *delegated trust authentication model* by applications running BEA Tuxedo Release 7.1 or later software. Workstation Handlers (WSHs) and domain gateways (`GWTDOMAINs`) are known as *trusted system gateway processes* in the delegated trust authentication model, which is described in "Understanding Delegated Trust Authentication" on page 1-7.

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



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

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

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

See Also

- "Authentication" on page 1-7
- "Default Authentication and Authorization" on page 1-44

- "Administering Default Authentication and Authorization" on page 2-56
- "Security Administration Tasks" on page 2-3
- "Security Interoperability" on page 1-53
- "Security Compatibility" on page 1-59
- "What Is a Domain" on page 4-18 in *Introducing the BEA Tuxedo System*

Specifying Principal Names

As the administrator, you use the following configuration parameters to specify principal names for the Workstation Handler (WSH), domain gateway (GWTDOMAIN), and server processes running in your Release 7.1 or later BEA Tuxedo application.

Parameter Name	Description	Setting
SEC_PRINCIPAL_NAME in UBBCONFIG (TA_SEC_PRINCIPAL_NAME in TM_MIB)	During application booting, each WSH, domain gateway, and server process in the application calls the authentication plug-in to acquire security credentials for the *security principal name* specified in SEC_PRINCIPAL_NAME.*	1 - 511 characters. If not specified at any level in the configuration hierarchy, the security principal name defaults to the DOMAINID string specified in the UBBCONFIG file.
CONNECTION_PRINCIPAL_NAME for local domain access point in DMCONFIG (TA_DMCONNPRINCIPALNAME for LACCESSPOINT in DM_MIB)**	During application booting, each domain gateway process in the application calls the authentication plug-in a second time to acquire security credentials for the *connection principal name* specified in CONNECTION_PRINCIPAL_NAME.*	1 - 511 characters. If not specified, the connection principal name defaults to the DOMAINID string for the local domain access point specified in the DMCONFIG file.

* The topics that follow explain how the system processes acquire credentials and why they need them.

** The local domain access point is also known as the LDOM (pronounced "el dom") or simply *local domain*.

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

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

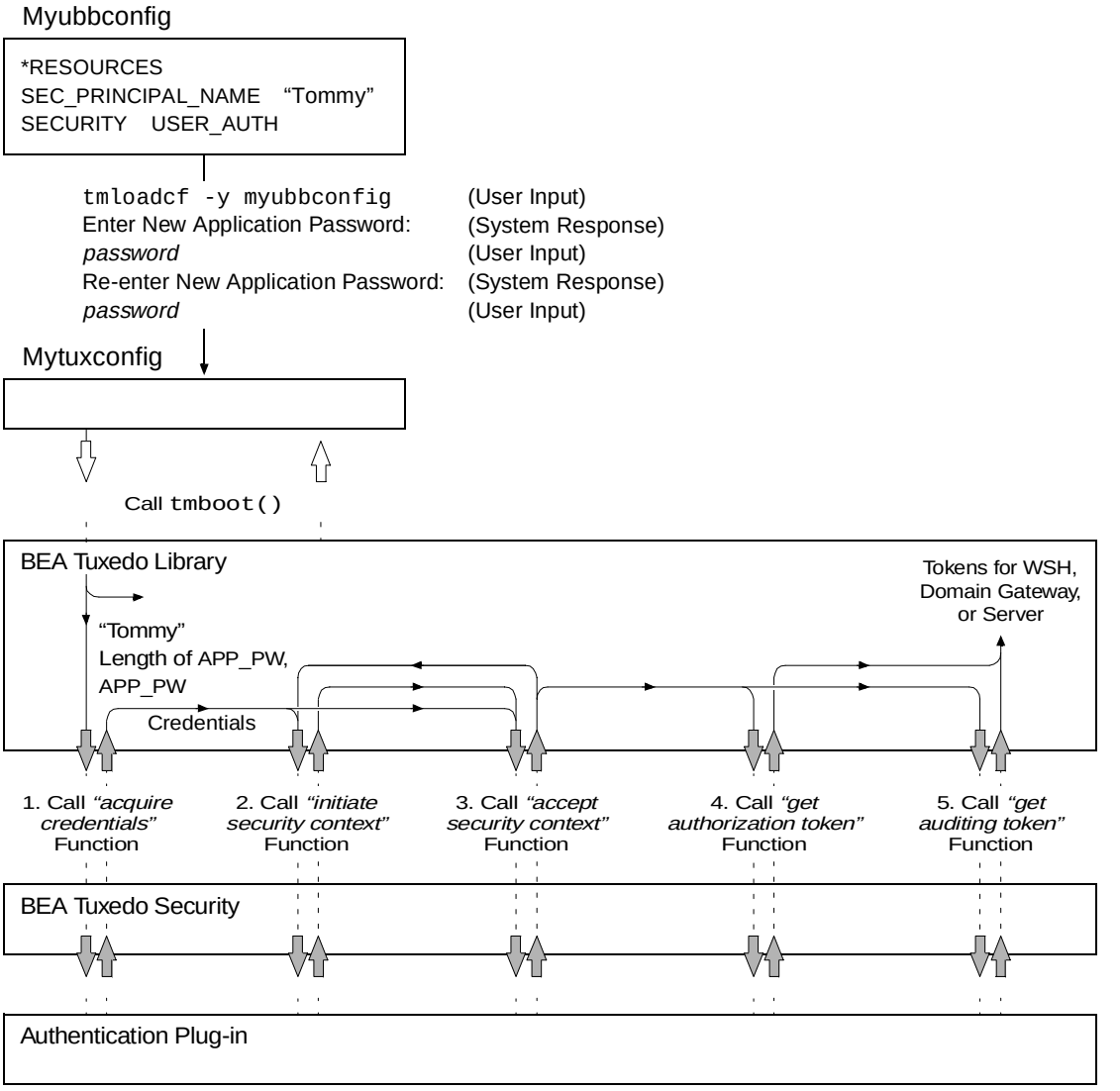
A security principal name at a particular configuration level can be overridden at a lower level. For example, suppose you configure terri as the principal name for machine mach1, and john as the principal name for server serv1 running on mach1. The processes on mach1 behave as follows:

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

How System Processes Acquire Credentials

During application booting, each WSH, domain gateway, and server process in the application includes its *security principal name* as an argument when calling the authentication plug-in to (1) acquire security credentials and (2) get authorization and auditing tokens for itself. The following diagram demonstrates the procedure.

Figure 2-3 Acquiring Credentials and Tokens During Application Booting



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

Why System Processes Need Credentials

A WSH needs credentials so that it can authenticate Workstation clients that want to join the application, and to get authorization and auditing tokens for the authenticated Workstation clients. A WSH needs its own authorization and auditing tokens when handling requests from pre-Release 7.1 clients (clients running BEA Tuxedo Release 6.5 or earlier software) so that it can call the authentication plug-in to establish identities for the older clients. This behavior is described in "Mandating Interoperability Policy" on page 2-15.

A domain gateway needs one set of credentials so that it can authenticate remote domain gateways for the purpose of establishing links between BEA Tuxedo applications, as described in "Establishing a Link Between Domains" on page 2-24. (No authorization or auditing tokens are assigned to authenticated remote domain gateways.) A domain gateway acquires these credentials for the principal name specified in the `CONNECTION_PRINCIPAL_NAME` parameter.

A domain gateway needs a second set of credentials so that it can handle requests from pre-Release 7.1 clients, which involves calling the authentication plug-in to establish identities for the older clients. This behavior is described in "Mandating Interoperability Policy" on page 2-15. It also needs these credentials to establish identities when enforcing the local access control list (ACL) policy, as described in "Setting ACL Policy" on page 2-29. A domain gateway acquires these credentials for the principal name specified in the `SEC_PRINCIPAL_NAME` parameter.

A system or application server needs its own authorization and auditing tokens when handling requests from pre-Release 7.1 clients so that it can call the authentication plug-in to establish identities for the older clients. This behavior is described in "Mandating Interoperability Policy" on page 2-15.

A server also needs its own tokens when performing a *server permission upgrade*, which occurs when the authorization and auditing tokens of the server are assigned to messages that pass through the server but originate at a client. The service upgrade capability is described in "Replacing Client Tokens with Server Tokens" on page 1-11.

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

Example UBBCONFIG Entries for Principal Names

The following example pertains to specifying security principal names in the UBBCONFIG file using the SEC_PRINCIPAL_NAME parameter. For an example of specifying connection principal names in the DMCONFIG file using the CONNECTION_PRINCIPAL_NAME parameter, see “Example DMCONFIG Entries for Establishing a Link” on page 2-27.

```
*RESOURCES
SEC_PRINCIPAL_NAME    "Tommy"
      .
      .
      .

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

See Also

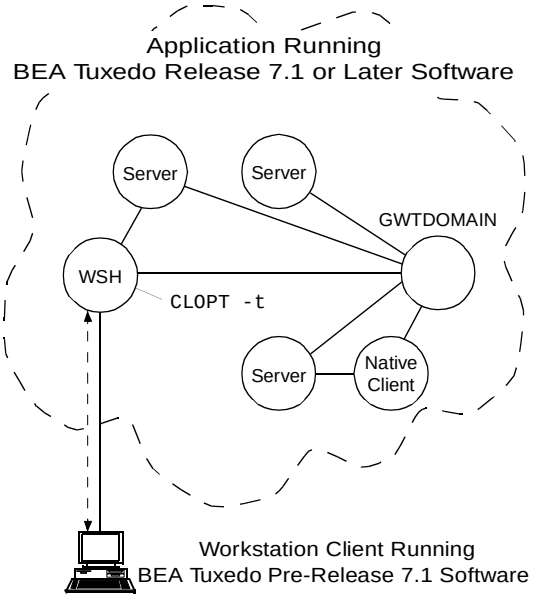
- “Mandating Interoperability Policy” on page 2-15
- “Establishing a Link Between Domains” on page 2-24
- “Setting ACL Policy” on page 2-29
- “Security Administration Tasks” on page 2-3

Mandating Interoperability Policy

As the administrator, you use the CLOPT -t option in the UBBCONFIG file to allow WSH, domain gateway (GWTDOMAIN), and server processes in your application to interoperate with machines running BEA Tuxedo pre-Release 7.1 (6.5 or earlier) software. In addition, you use the WSALLOWPRE71 environment variable to allow

Workstation clients to interoperate with machines running BEA Tuxedo pre-Release 7.1 software. The following four figures show what interoperability means for these processes.

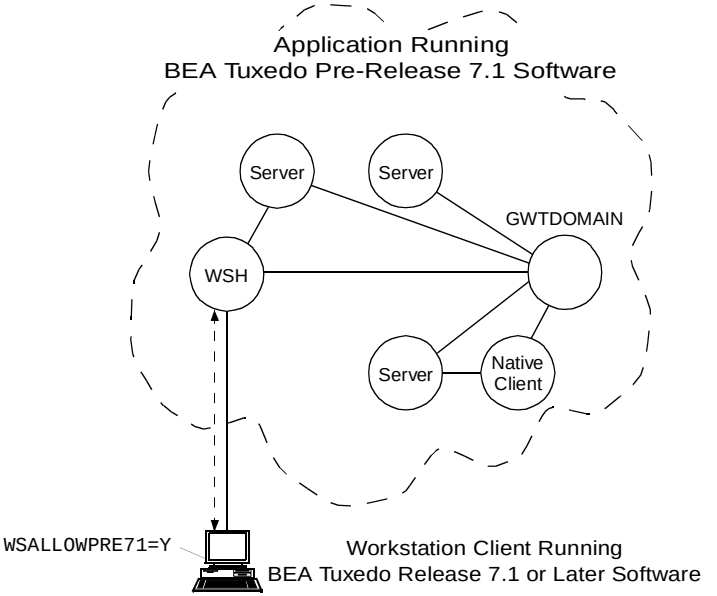
Figure 2-4 WSH Operating with Older Workstation Client



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

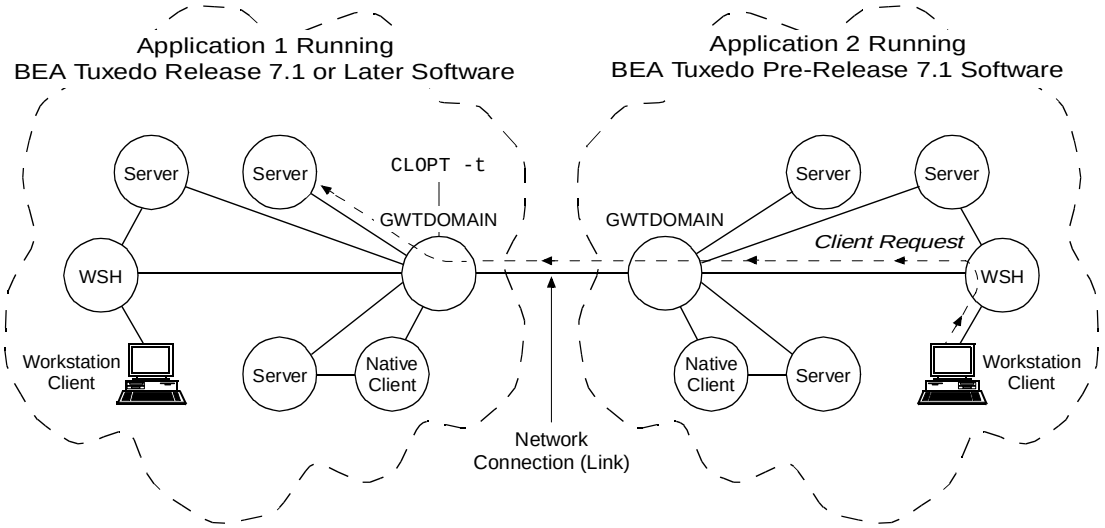
Note: The "impersonate user" function involves calling the authentication plug-in to establish an identity for the older client. See "Establishing an Identity for an Older Client" on page 2-20 for details.

Figure 2-5 Older WSH Operating with Workstation Client



In the preceding figure, the WSH authenticates with the Workstation client using an older (pre-Release 7.1) authentication protocol; the client request does *not* receive authorization and auditing tokens. If the `WSALLOWPRE71` environment variable is not set at the Workstation client or is set to `N`, no communication is possible between the older WSH and the newer Workstation client.

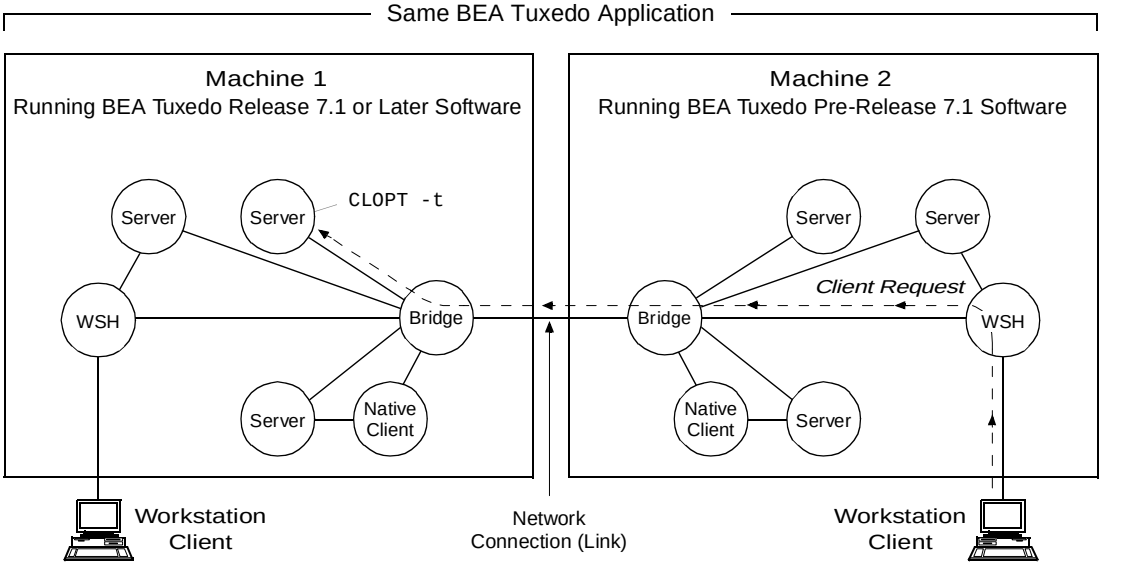
Figure 2-6 Server Interoperating with Older BEA Tuxedo Application



In the preceding figure, the local domain gateway (`GWTDOMAIN`) in application 1 authenticates with the remote domain gateway in application 2 using an older (pre-Release 7.1) authentication protocol. Upon receiving a request from a remote client, the local domain gateway calls the internal "impersonate user" function to get authorization and auditing tokens for the remote client and then attaches the tokens to the client request. For any outbound client request (client request originating in application 1 and destined for application 2), the local domain gateway strips the tokens from the request before sending the request along with the client's *application key* to the older application. (See "Application Key" on page 1-48 for a description of the application key.)

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

Figure 2-7 Server Interoperating with Older BEA Tuxedo System



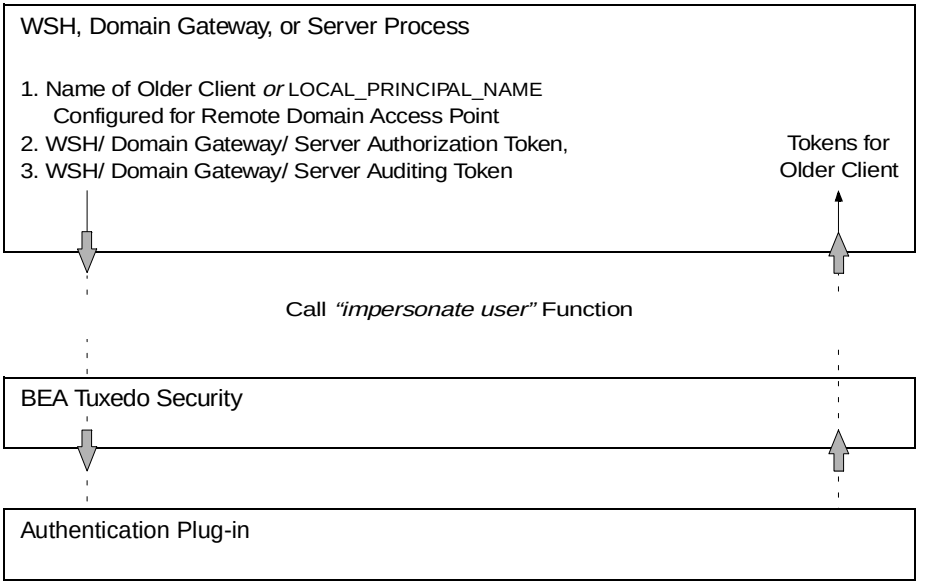
In the preceding figure, the destination server on machine 1 calls the internal "impersonate user" function to get authorization and auditing tokens for the remote client on machine 2, attaches the tokens to the client request, and then performs the request *assuming* the client passes any authorization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

Note: Also, in the preceding figure, if the WSH on machine 1 receives a client request destined for a server on machine 2, the WSH strips the tokens from the request before sending the request along with the client's application key to the older system. Similarly, if the native client on machine 1 sends a request to a server on machine 2, the native client strips the tokens from the request before sending the request along with the client's application key to the older system. See "Application Key" on page 1-48 for a description of the application key.

Establishing an Identity for an Older Client

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

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



How the WSH Establishes an Identity for an Older Client

When the CLOPT -t option is specified, the WSH establishes an identity for an older client using the usrname field of the TPINIT buffer for C, or the USRNAME field of the TPINFDEF-REC record for COBOL. (The WSH receives a TPINIT buffer/ TPINFDEF-REC record from a client when the client attempts to join the application, as described in "Joining the Application" on page 3-8.) The WSH includes the user name as the principal name when calling the "impersonate user" function.

For default authentication plug-ins, the "impersonate user" function finds the user name and its associated application key (user identifier, group identifier combination) in the local tpusr file, and then includes the user name and application key in both the

authorization and auditing tokens created for the older client. The `tpusr` file is briefly described in "Setting Up the User and Group Files" on page 2-61.

How the Domain Gateway Establishes an Identity for an Older Client

When the `CLOPT -t` option is specified, the domain gateway establishes an identity for an older client using the `LOCAL_PRINCIPAL_NAME` string configured for the remote domain access point. (The domain gateway searches the `DM_REMOTE_DOMAINS` section of the local `BDMCONFIG` file—the binary equivalent of the `DMCONFIG(5)` file—to find the `LOCAL_PRINCIPAL_NAME` string for the remote domain access point. If not specified, the identity defaults to the `DOMAINID` string for the remote domain access point.) The domain gateway uses the `LOCAL_PRINCIPAL_NAME` string as the principal name when calling the "impersonate user" function.

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

How the Server Establishes an Identity for an Older Client

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

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

Summarizing How the CLOPT -t Option Works

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

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

Process	Interoperability Allowed (CLOPT -t)	Interoperability Not Allowed
Workstation Handler (WSH)	If the WSH receives a request from a pre-Release 7.1 Workstation client to join the application, the WSH authenticates the client using a pre-Release 7.1 authentication protocol and calls the "impersonate user" function to get authorization and auditing tokens for the client *based on the user name given in the request.* When the WSH receives a service request from the authenticated Workstation client, it attaches the tokens to the client request and forwards the request to the destination server.	If the WSH receives a request from a pre-Release 7.1 Workstation client to join the application, the WSH rejects the request. No communication is possible between the newer WSH and the older Workstation client.
Domain gateway (`GWTDOMAIN`)	When the domain gateway sets up a connection to a pre-Release 7.1 remote domain gateway, it authenticates the remote domain gateway using a pre-Release 7.1 authentication protocol and then sets up the network connection. When the domain gateway receives a client request from the older domain, the domain gateway calls the "impersonate user" function to get authorization and auditing tokens for the client *based on the* `LOCAL_PRINCIPAL_NAME` (defaults to `DOMAINID`) *identity configured for the remote domain access point*, attaches the tokens to the client request, and then forwards the request to the destination server. The client has the same access permissions as the `LOCAL_PRINCIPAL_NAME` identity. For any outbound client request, the domain gateway strips the tokens from the request before sending the request along with the client's application key to the older domain.	The domain gateway does *not* set up a connection to a pre-Release 7.1 remote domain gateway. No communication is possible between the newer and older domains.

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

Process	Interoperability Allowed (CLOPT -t)	Interoperability Not Allowed
System or application server	If the server receives a request from a remote client running BEA Tuxedo pre-Release 7.1 software, the server calls the "impersonate user" function to get authorization and auditing tokens for the client *based on the client's assigned application key*, and then performs the client request assuming the client passes any authorization checks.	If the server receives a request from a remote client running BEA Tuxedo pre-Release 7.1 software, the server rejects the client request. No communication is possible between the newer server and the older client.

Example UBBCONFIG Entries for Interoperability

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

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

See Also

- "Specifying Principal Names" on page 2-11
- "Establishing a Link Between Domains" on page 2-24
- "Setting ACL Policy" on page 2-29
- "Security Administration Tasks" on page 2-3
- "Security Interoperability" on page 1-53
- "Setting Up Security in Domains" on page 2-35 and "Configuring the Connections Between Your Domains" on page 2-46 in *Using the BEA Tuxedo Domains Component*

Establishing a Link Between Domains

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

1. The *initiator* domain gateway and the *target* domain gateway exchange link-level encryption (LLE) *min-max* values to be used to set up LLE on the link between the gateways. LLE is described in "Link-Level Encryption" on page 1-23.

2. The initiator and target domain gateways authenticate one another through the exchange of security tokens *assuming* that both gateways are running BEA Tuxedo Release 7.1 or later software.

   If one or both of the domain gateways are running BEA Tuxedo pre-Release 7.1 software, the gateway processes use an older (pre-Release 7.1) authentication protocol when setting up the connection.

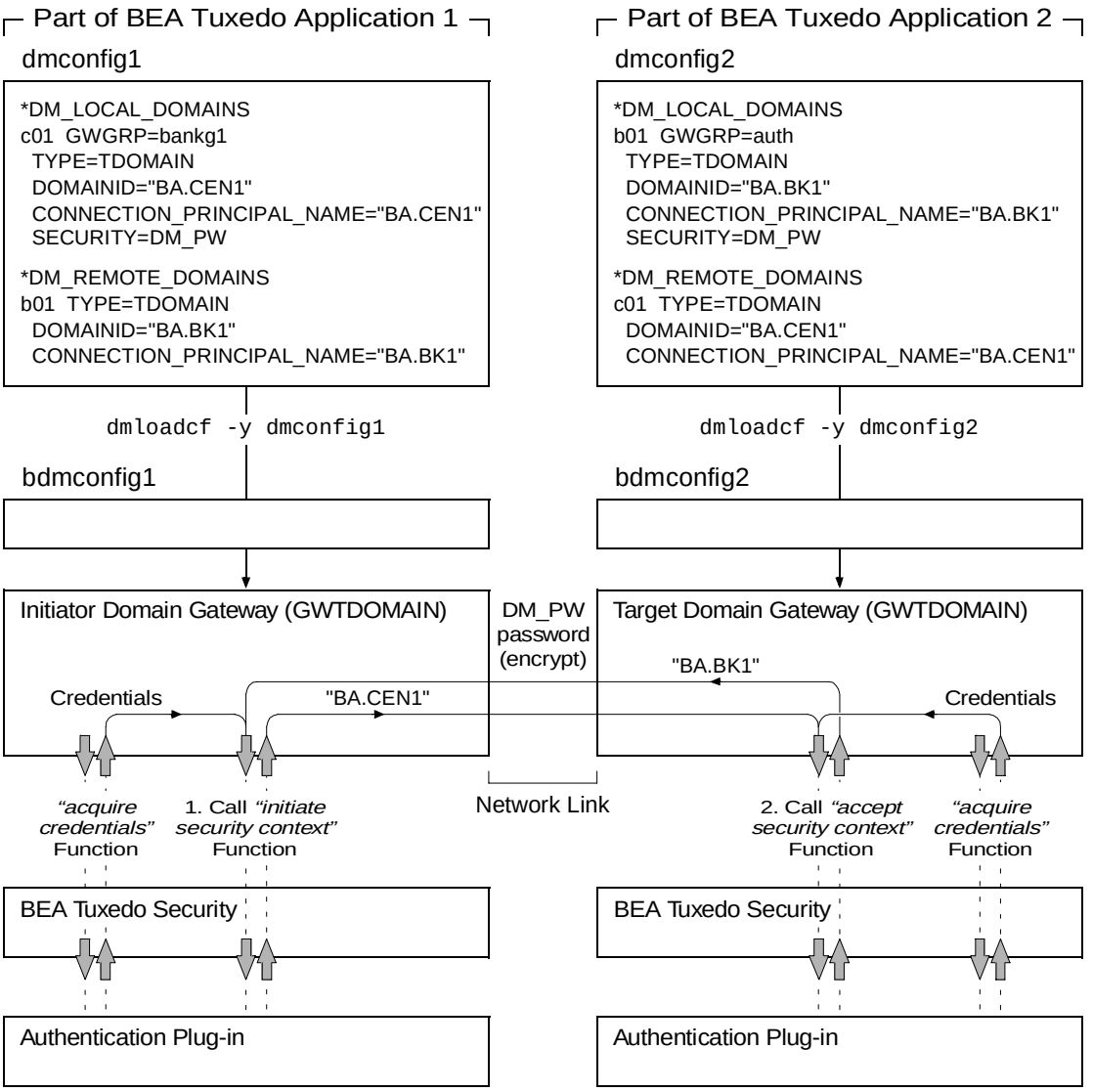
As the administrator, you use the following configuration parameter to establish a link between domain gateways running BEA Tuxedo Release 7.1 or later software.

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

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

The following diagram demonstrates how a link is established between domains using default authentication plug-ins.

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



Note: The "Credentials" shown in the preceding diagram were acquired by each domain gateway process at application booting using the CONNECTION_PRINCIPAL_NAME identity configured for the local domain access point.

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

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

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

Example DMCONFIG Entries for Establishing a Link

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

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

```
    .
    .
*DM_REMOTE_DOMAINS
# <RDOM name> <domain type> <domain id>
#       [<connection principal name>]...
b01     TYPE=TDOMAIN
        DOMAINID="BA.BANK01"
        CONNECTION_PRINCIPAL_NAME="BA.BANK01"
```

See Also

- “Specifying Principal Names” on page 2-11
- “Mandating Interoperability Policy” on page 2-15
- “Setting ACL Policy” on page 2-29
- “Security Administration Tasks” on page 2-3
- “How to Set Up Domains Authentication” on page 2-39 in *Using the BEA Tuxedo Domains Component*

Setting ACL Policy

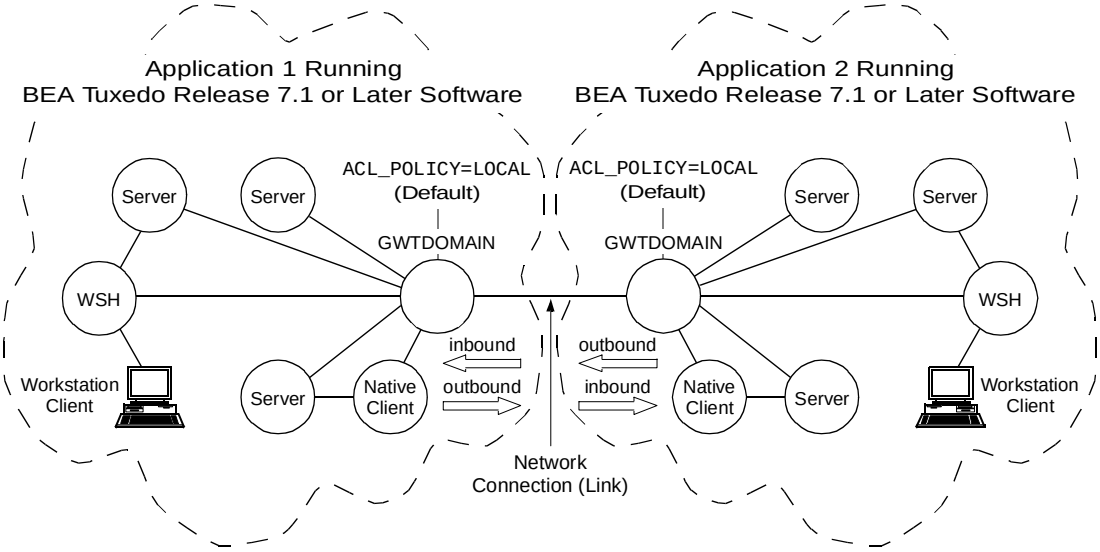
As the administrator, you use the following configuration parameters to set and control the access control list (ACL) policy between applications running BEA Tuxedo Release 7.1 or later software.

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

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

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

Figure 2-10 Establishing a Local ACL Policy

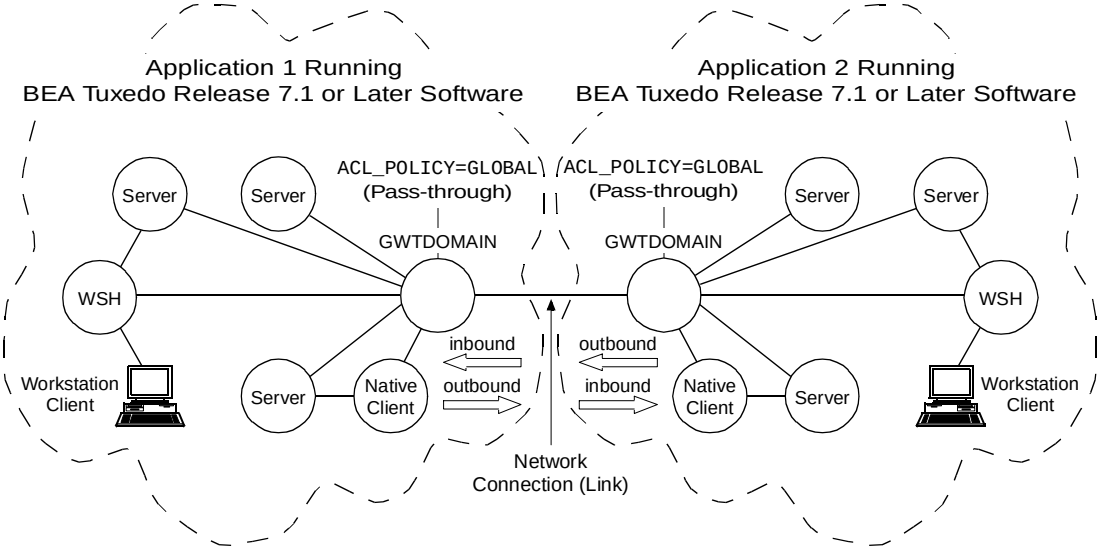


In the preceding figure, each domain gateway (`GWTDOMAIN`) modifies *inbound* client requests (requests originating from the remote application and received over the network connection) so that they take on the `LOCAL_PRINCIPAL_NAME` identity configured for the remote domain access point and thus have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

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

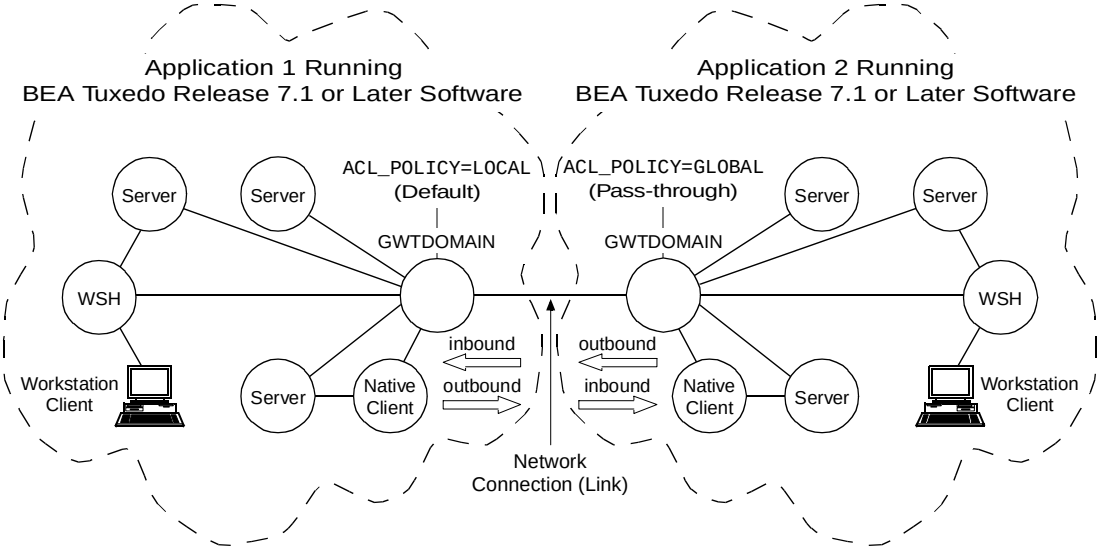
Note: The preceding description also applies to applications running BEA Tuxedo pre-Release 7.1 software except that the system uses the `DOMAINID` identity configured for the remote domain access point. Essentially, the local ACL policy is hardcoded in BEA Tuxedo Release 6.5 or earlier software.

Figure 2-11 Establishing a Global ACL Policy



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

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



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

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

Impersonating the Remote Domain Gateway

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

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

For more detail on the “impersonate user” function, see “Establishing an Identity for an Older Client” on page 2-20.

Example DMCONFIG Entries for ACL Policy

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

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

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

See Also

- “Specifying Principal Names” on page 2-11
- “Mandating Interoperability Policy” on page 2-15
- “Establishing a Link Between Domains” on page 2-24
- “Security Administration Tasks” on page 2-3

Administering Authorization

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

The procedures for administering authorization depend upon the underlying authorization system of the application. For procedures to administer a custom authorization system, see the documentation for that system. For procedures to administer the default authorization system, see “Administering Default Authentication and Authorization” on page 2-56.

See Also

- “Authorization” on page 1-12
- “Default Authentication and Authorization” on page 1-44
- “Administering Default Authentication and Authorization” on page 2-56
- “Security Administration Tasks” on page 2-3
- “Security Compatibility” on page 1-59

Administering Link-Level Encryption

Link-level encryption establishes data privacy for messages moving over the network links that connect the machines in a BEA Tuxedo application. There are three levels of link-level encryption (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.

LLE applies to the following types of BEA Tuxedo links:

- Workstation client to Workstation Handler (WSH)
- Bridge to Bridge
- Administrative utility (such as `tmboot`) to `tlisten`
- Domain gateway to domain gateway

Understanding min and max Values

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

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

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

For more information, see “How LLE Works” on page 1-24 and “Encryption Key Size Negotiation” on page 1-24.

Verifying the Installed LLE Version

You can verify the LLE version installed on a machine by running the `tmadmin` command in `verbose` mode.

```
tmadmin -v
```

Key lines from the local BEA Tuxedo `lic.txt` file will appear on your computer screen, similar to the sample display shown below. The sample entry `STRENGTH=128` indicates a United States and Canada LLE version.

```
[BEA Tuxedo] VERSION=7.1
[LINK ENCRYPTION] VERSION=7.1
STRENGTH=128
   .
   .
   .
```

All BEA Tuxedo licenses are in the `$TUXDIR/udataobj/lic.txt` file on a UNIX host machine, or in the `%TUXDIR%\udataobj\lic.txt` file on a Windows NT host machine.

How to Configure LLE on Workstation Client Links

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

As the administrator, you enable Workstation client access to the application by specifying a WSL server in the `SERVERS` section of the application's `UBBCONFIG` file. You need to specify the `-z` and `-Z` command-line options for the WSL server if you want to override the defaults for the LLE *`min`* and *`max`* parameters. (See "Understanding min and max Values" on page 2-35 for details.) Of course, link-level encryption is possible only if LLE is installed on both the local machine and the Workstation client.

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

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

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

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

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

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

See WSL(5), WS_MIB(5), and UBBCONFIG(5) in *BEA Tuxedo File Formats and Data Descriptions Reference* for additional information.

How to Configure LLE on Bridge Links

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

As the administrator, you place an entry in the NETWORK section of the UBBCONFIG file for each machine in a BEA Tuxedo application on which a Bridge server resides. You need to specify the MINENCRYPTBITS and MAXENCRYPTBITS optional run-time parameters for the Bridge server if you want to override the defaults for the LLE *min*

and *max* parameters. (See “Understanding min and max Values” on page 2-35 for details.) Of course, Bridge-to-Bridge link-level encryption is possible only if LLE is installed on the machines where the Bridge servers reside.

To configure LLE on Bridge links, follow these steps.

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

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

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

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

See TM_MIB(5) and UBBCONFIG(5) in *BEA Tuxedo File Formats and Data Descriptions Reference* for additional information.

How to Configure LLE on tlisten Links

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

To configure LLE on tlisten links, follow the steps given in the previous topic, "How to Configure LLE on Bridge Links" on page 2-37. If you so desire, you can start a separate instance of tlisten on the local machine by entering a command such as:

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

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

How to Configure LLE on Domain Gateway Links

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

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

As the administrator, you must place an entry in the DM_TDOMAIN section of the DMCONFIG file for each local domain access point that will accept requests for local services from remote domain access points. You must also create an entry for each remote domain access point accessible by a defined local domain access point. You need to specify the MINENCRYPTBITS and MAXENCRYPTBITS optional run-time parameters for each domain access point for which you want to override the defaults for the LLE *min* and *max* parameters. (See "Understanding min and max Values" on page 2-35 for details.) Of course, domain-to-domain link-level encryption is possible only if LLE is installed on the machines where the domains reside.

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

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

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

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

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

In the preceding example, when tmboot(1) starts the application, each domain gateway reads the BDMCONFIG file to access various parameters, including MINENCRYPTBITS and MAXENCRYPTBITS, and propagates those parameters to its local and remote domains. When the local domain is establishing a network link with a remote domain, the two domains negotiate the key size until they agree on the largest key size supported by both.

See DMCONFIG(5) in *BEA Tuxedo File Formats and Data Descriptions Reference* for additional information. Also, see “Setting Up Security in Domains” on page 2-35“ in *Using the BEA Tuxedo Domains Component*.

See Also

- “Link-Level Encryption” on page 1-23
- “Security Administration Tasks” on page 2-3
- “Security Interoperability” on page 1-53
- “Security Compatibility” on page 1-59

Administering Public Key Security

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

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

Recommended Practices for Public Key Security

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

Assigning Public-Private Key Pairs

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

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

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

Setting Digital Signature Policy

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

Parameter Name	Description	Setting
`SIGNATURE_AHEAD` in `UBBCONFIG` (`TA_SIGNATURE_AHEAD` in `TM_MIB`)	Maximum permissible time difference between (1) the timestamp value attached to a digitally signed message buffer and (2) the time at which the message buffer is received. If the signature timestamp is too far into the future, the receiving process rejects the message buffer.	1 - 2147483647 seconds. Default is 3600 seconds (one hour).

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

Setting a Postdated Limit for Signature Timestamps

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

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

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

Example UBBCONFIG Entries for Postdated Limit

```
*RESOURCES
SIGNATURE_AHEAD 2400
```

Setting a Predated Limit for Signature Timestamps

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

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

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

Example UBBCONFIG Entries for Predated Limit

```
*RESOURCES
SIGNATURE_BEHIND 300000
```

Enforcing the Signature Policy for Incoming Messages

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

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

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

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

You may specify both SIGNATURE_REQUIRED=Y and ENCRYPTION_REQUIRED=Y *together* at the domain-wide level, machine level, group level, or service level. See "Enforcing the Encryption Policy for Incoming Messages" on page 2-47 for a description of ENCRYPTION_REQUIRED.

Qualifier

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

Example

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

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

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

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

In the preceding example, when `tmboot(1)` starts the application, it passes the `SIGNATURE_REQUIRED=Y` parameter to the machine named `mach1`. At that point, all application services advertised by `mach1`, including those advertised by gateway processes, are allowed to accept only messages that include valid digital signatures. If a process controlled by `mach1` receives a message that does *not* include a valid digital signature, the system takes the following actions:

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

Note: A NULL (empty) buffer cannot be digitally signed, meaning that the system rejects any NULL buffer received by a process requiring digital signatures, in the manner stated in the preceding bullet list.

How the EventBroker Signature Policy Is Enforced

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

If the `TMUSREVT(5)` system server is running in a domain, machine, or server group that requires digital signatures, it rejects any incoming posting without a `TPSIGN_OK` composite signature status—see “Understanding the Composite Signature Status” on page 3-56.

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

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

How the /Q Signature Policy Is Enforced

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

If the TMQUEUE(5) system server is running in a domain, machine, or server group that requires digital signatures, it rejects any incoming enqueue request without a TPSIGN_OK composite signature status—see "Understanding the Composite Signature Status" on page 3-56. In addition, the TMQUEUE server requires a digital signature if such a policy is in effect for the service name associated with the queue space.

How the Remote Client Signature Policy Is Enforced

If the Workstation Handler (WSH) is running in a domain, machine, or server group that requires digital signatures, it rejects any incoming message buffer containing application data without a TPSIGN_OK composite signature status—see "Understanding the Composite Signature Status" on page 3-56.

Setting Encryption Policy

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

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

Enforcing the Encryption Policy for Incoming Messages

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

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

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

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

You may specify both ENCRYPTION_REQUIRED=Y and SIGNATURE_REQUIRED=Y *together* at the domain-wide level, machine level, group level, or service level. See "Enforcing the Signature Policy for Incoming Messages" on page 2-44 for a description of SIGNATURE_REQUIRED.

Qualifier

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

Example

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

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

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

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

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

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

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

How the EventBroker Encryption Policy Is Enforced

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

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

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

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

How the /Q Encryption Policy Is Enforced

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

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

How the Remote Client Encryption Policy Is Enforced

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

Initializing Decryption Keys Through the Plug-ins

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

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

Parameter Name	Description	Setting
SEC_PRINCIPAL_PASSVAR in UBBCONFIG (SEC_PRINCIPAL_PASSVAR in TM_MIB)	The variable in which the password for the target principal is stored.	1 - 511 characters. If not specified, defaults to a NULL (zero length) string.

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

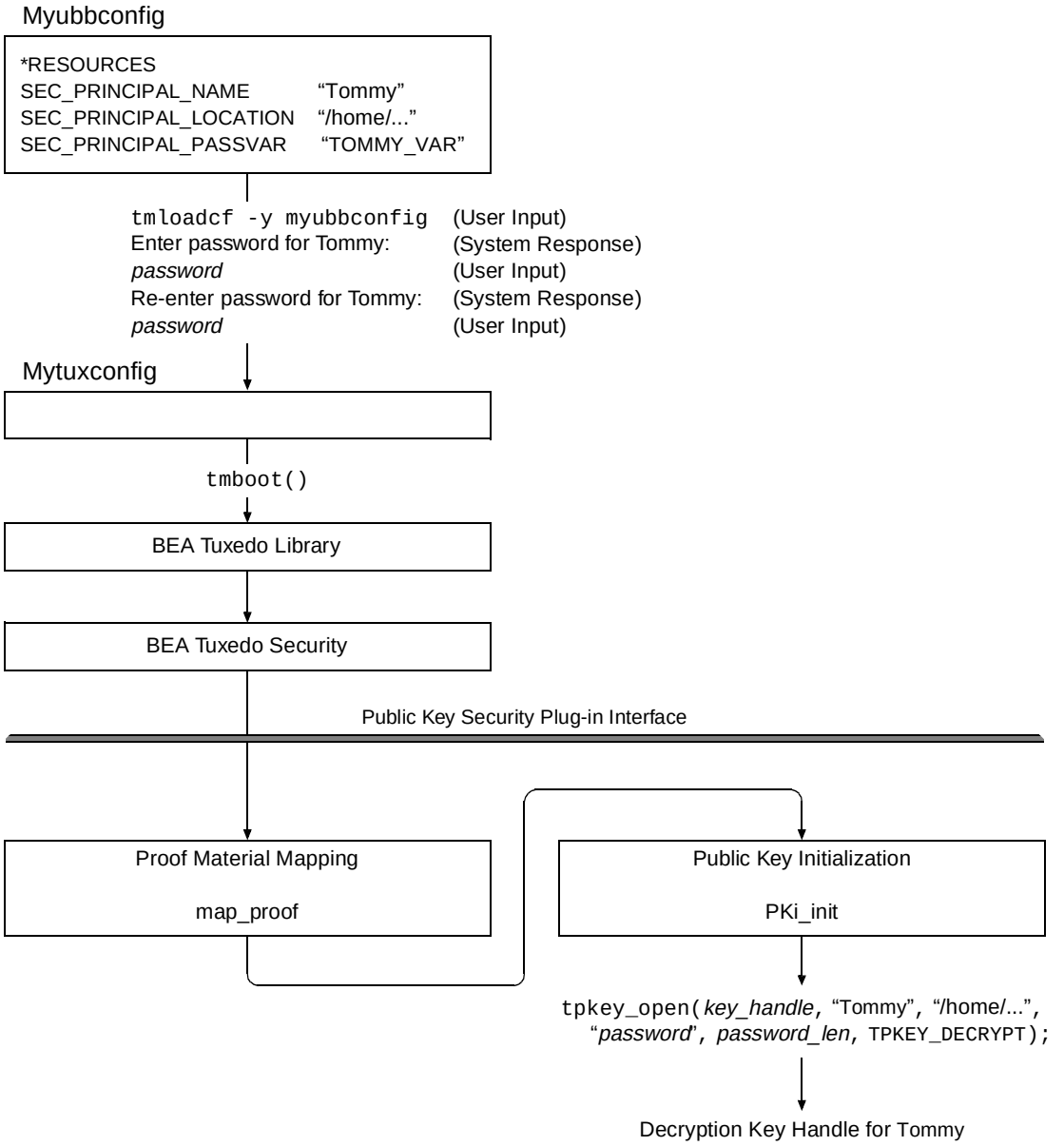
- RESOURCES section in UBBCONFIG or T_DOMAIN class in TM_MIB
- MACHINES section in UBBCONFIG or T_MACHINE class in TM_MIB
- GROUPS section in UBBCONFIG or T_GROUP class in TM_MIB
- SERVERS section in UBBCONFIG or T_SERVER class in TM_MIB

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

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

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

Figure 2-13 How a Decryption Key Is Initialized—Example



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

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

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

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

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

Example UBBCONFIG Entries for Principal Names and Decryption Keys

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

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

Failure Reporting and Auditing

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

Digital Signature Error Handling

If message tampering is detected (that is, if the composite signature status is either `TPSIGN_TAMPERED_MESSAGE` or `TPSIGN_TAMPERED_CERT`—see "Understanding the Composite Signature Status" on page 3-56), the system takes the following actions:

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

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

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

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

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

Encryption Error Handling

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

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

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

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

See Also

- "Public Key Security" on page 1-29
- "Public Key Implementation" on page 1-41
- "Security Administration Tasks" on page 2-3
- "Security Interoperability" on page 1-53
- "Security Compatibility" on page 1-59

Administering Default Authentication and Authorization

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

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

Designating a Security Level

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

Establishing Security by Editing the Configuration File

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

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

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

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

Establishing Security by Changing the TM_MIB

To designate a security level through the TM_MIB, you must assign a value to the TA_SECURITY attribute in the T_DOMAIN class. When an application is inactive, the administrator can SET the value of TA_SECURITY to any of the values that are valid in UBBCONFIG. To complete this task, run the administrative interface tpadmcall(3c).

Establishing Security by Using the BEA Administration Console

You can also designate a security level through the BEA Administration Console. The BEA Administration Console is a Web-based tool used to configure, monitor, and dynamically re-configure an application.

Configuring the Authentication Server

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

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

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

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

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

As another example:

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

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

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

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

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

```
*RESOURCES
SECURITY   USER_AUTH
AUTHSVC    KERBEROS
    .
    .
    .
*SERVERS
KERBEROSSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600
MAXGEN=2 CLOPT="-A"
```

See Also

- “How to Enable Application Password Security” on page 2-59
- “How to Enable User-Level Authentication Security” on page 2-60
- “Enabling Access Control Security” on page 2-64
- “Default Authentication and Authorization” on page 1-44
- “Security Administration Tasks” on page 2-3
- `AUTHSVR(5)` in the *BEA Tuxedo File Formats and Data Descriptions Reference*

How to Enable Application Password Security

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

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

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

See Also

- “Default Authentication and Authorization” on page 1-44
- “Administering Default Authentication and Authorization” on page 2-56
- “Security Administration Tasks” on page 2-3

How to Enable User-Level Authentication Security

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

To enable the `USER_AUTH` security level, follow these steps.

1. Set up the `UBBCONFIG` file.
2. Set up the user and group files.

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

Setting Up the UBBCONFIG File

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

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

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

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

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

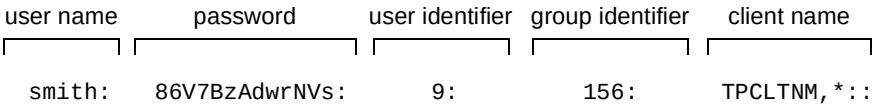
4. The system prompts you for a password. The password you enter may be up to 30 characters long. It becomes the password for the application and remains in effect until you change it by using the passwd command of tmadmin.

5. Distribute the application password to authorized users of the application through an off-line means such as telephone or letter.

Setting Up the User and Group Files

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

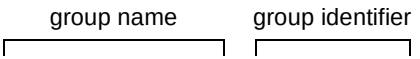
The following display is a sample entry in the tpusr file.



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

AUTHSVC assigns an authenticated client user an *application key*, which contains a user identifier and associated group identifier for the USER_AUTH, ACL, or MANDATORY_ACL security level. (See "Application Key" on page 1-48 for more information about application keys.)

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




```
Administrators::      156:
```

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

Converting System Security Data Files to BEA Tuxedo User and Group Files

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

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

2. To convert the `/etc/password` file into the format needed by the BEA Tuxedo system, enter the following command.

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

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

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

3. To convert the `/etc/group` file into the format needed by the BEA Tuxedo system, enter the following command.

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

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

Adding, Modifying, or Deleting Users and Groups

The BEA Tuxedo system requires that you maintain a list of your application users in a file named `tpusr`, and a list of groups, in a file named `tpgrp`. There are two methods of modifying the entries in these files: by issuing commands or by changing the values of the appropriate attributes in the `ACL_MIB`.

Changing Entries for Users and Groups through Commands

You can add, modify, or delete entries in the `tpusr` and `tpgrp` files at any time by running a BEA Tuxedo command provided for that purpose.

Run . . .	To . . .	An entry in this file
`tpusradd(1)`	Add	`tpusr`
`tpusrmod(1)`	Modify	
`tpusrdel(1)`	Delete	
`tpgrpadd(1)`	Add	`tpgrp`
`tpgrpmod(1)`	Modify	
`tpgrpdel(1)`	Delete	

To run any of these commands, follow these steps.

1. For an inactive application, make sure you are working from the application `MASTER` machine. For an active application, you may work from any machine in the configuration.
2. For specific instructions on running a command, see the entry for that command in the *BEA Tuxedo Command Reference*.

Changing Entries for Users and Groups through the ACL_MIB

If you prefer not to use the command-line interface, you can add, modify, or delete user entries in `tpusr` by changing the appropriate attribute values in the `T_ACLPRINCIPAL` class in the `ACL_MIB(5)`. This method is more efficient than the command-line interface if you want to add several user entries simultaneously, since `tpusradd(1)` allows you to add only one user at a time.

Similarly, you can add, modify, or delete group entries in `tpgrp` by changing the appropriate attribute values in the `T_ACLGROUP` class in the `ACL_MIB(5)`. This method is more efficient than the command-line interface if you want to add several group entries simultaneously, since `tpgrpadd(1)` allows you to add only one group at a time.

Of course, the easiest way to access the `MIB` is via the BEA Administration Console.

See Also

- "Default Authentication and Authorization" on page 1-44
- "Administering Default Authentication and Authorization" on page 2-56
- "Security Administration Tasks" on page 2-3

Enabling Access Control Security

Default authorization consists of an access control checking feature that determines which users can execute a service, post an event, or enqueue (or dequeue) a message on an application queue. There are two levels of access control security: optional access control list (`ACL`) and mandatory access control list (`MANDATORY_ACL`). Only when users are authenticated to join an application does the access control list become active.

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

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

The access control checking feature is based on three files that are created and maintained by the application administrator:

- `tpusr` contains a list of users
- `tpgrp` contains a list of groups
- `tpacl` contains a list of mappings of groups to application entities (such as services) known as the access control list (ACL)

By parsing the client's *application key*, which contains information identifying the client as a valid user and valid group member, an entity (such as a service, event, or application queue) can identify the group to which the user belongs; by checking the `tpacl` file, an entity can determine whether the client's group has access permission.

The application administrator, application operator, and processes or service requests running with the privileges of the application administrator/operator are *not* subject to ACL permission checking.

If user-level ACL entries are needed, they may be implemented by creating a group for each user, and then mapping the group to the appropriate application entities in the `tpacl` file.

How to Enable Optional ACL Security

Default authentication offers an *optional ACL* (`ACL`) security level that you invoke by specifying `SECURITY ACL` in your configuration file. This security level requires that each client provide an application password, a user name, and user-specific data, such as a password, to join the application. If there is no entry in the `tpacl` file associated with the target application entity, the user is permitted to access the entity.

This security level enables an administrator to configure access for only those resources that need more security. That is, there is no need to add entries to the `tpacl` file for services, events, or application queues that are open to everyone. Of course, if there *is* an entry in the `tpacl` file associated with the target application entity and a user attempts to access that entity, the user *must* be a member of a group that is allowed to access that entity; otherwise, permission is denied.

To enable the ACL security level, follow these steps.

1. Set up the UBBCONFIG file.
2. Set up the ACL file.

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

Setting Up the UBBCONFIG File

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

```
*RESOURCES
SECURITY    ACL
AUTHSVC     ..AUTHSVC
      .
      .
      .
*SERVERS
AUTHSVR SRVGRP="group_name" SRVID=1 RESTART=Y GRACE=600 MAXGEN=2
CLOPT="-A"
```

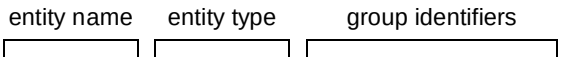
   CLOPT="-A" causes tmboot(1) to pass only the default command-line options (invoked by "-A") to AUTHSVR when tmboot starts the application. By default, AUTHSVR uses the client user information in a file named tpusr to authenticate clients that want to join the application. tpusr resides in the directory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

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

Setting Up the ACL File

The access control checking feature requires a user file named `tpusr`, a group file named `tpgrp`, and an ACL file named `tpacl`. The ACL file contains mappings of groups to application entities. An entity may be a service, event, or application queue.

The following display is a sample entry in the `tpacl` file.




```
TOLOWER:    SERVICE:    156,281,282,305:
```

As the administrator, you must define the entries in the `tpacl` file, which is located in the directory referenced by the first path name defined in the application's `APPDIR` variable. The file is a colon-delimited, flat text file, readable and writable only by the application's administrator.

There are two methods of modifying the ACL entries in the `tpacl` file: by issuing commands or by changing the values of the appropriate attributes in the `ACL_MIB`.

Changing ACL Entries through Commands

You can add, modify, or delete ACL entries in the `tpacl` file at any time by running a BEA Tuxedo command provided for that purpose.

Run . . .	To . . .
`tpacladd(1)`	Add an entry
`tpaclmod(1)`	Modify an entry
`tpacldel(1)`	Delete an entry

To run any of these commands, follow these steps.

1. For an inactive application, make sure you are working from the application `MASTER` machine. For an active application, you may work from any machine in the configuration.
2. For specific instructions on running a command, see the entry for that command in the *BEA Tuxedo Command Reference*.

Changing ACL Entries through the ACL_MIB

If you prefer not to use the command-line interface, you can add, modify, or delete ACL entries in `tpacl` by changing the appropriate attribute values in the `T_ACLPERM` class in the `ACL_MIB(5)`. This method is more efficient than the command-line interface if you want to add several ACL entries simultaneously, since `tpacladd(1)` allows you to add only one ACL entry at a time.

Of course, the easiest way to access the `MIB` is via the BEA Administration Console.

How to Enable Mandatory ACL Security

Default authentication offers a *mandatory ACL* security level that you invoke by specifying `SECURITY MANDATORY_ACL` in your configuration file. This security level requires that each client provide an application password, a user name, and user-specific data, such as a password, to join the application. If there is no entry in the `tpacl` file associated with the target application entity, the client is *not* permitted to access the entity. In other words, an entry *must* exist in the `tpacl` file for every application entity that a client needs to access. For this reason, this level is called *mandatory.*

Of course, if there *is* an entry in the `tpacl` file associated with the target application entity and a user attempts to access that entity, the user *must* be a member of a group that is allowed to access that entity; otherwise, permission is denied.

To enable the `MANDATORY_ACL` security level, follow these steps.

1. Set up the `UBBCONFIG` file.
2. Set up the ACL file.

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

Setting Up the UBBCONFIG File

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

```
*RESOURCES
SECURITY   MANDATORY_ACL
AUTHSVC    ..AUTHSVC
     .
     .
     .

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

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

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

Setting Up the ACL File

See "Setting Up the ACL File" on page 2-67.

See Also

- "Default Authentication and Authorization" on page 1-44
- "Administering Default Authentication and Authorization" on page 2-56
- "Security Administration Tasks" on page 2-3

CHAPTER

3 Programming Security

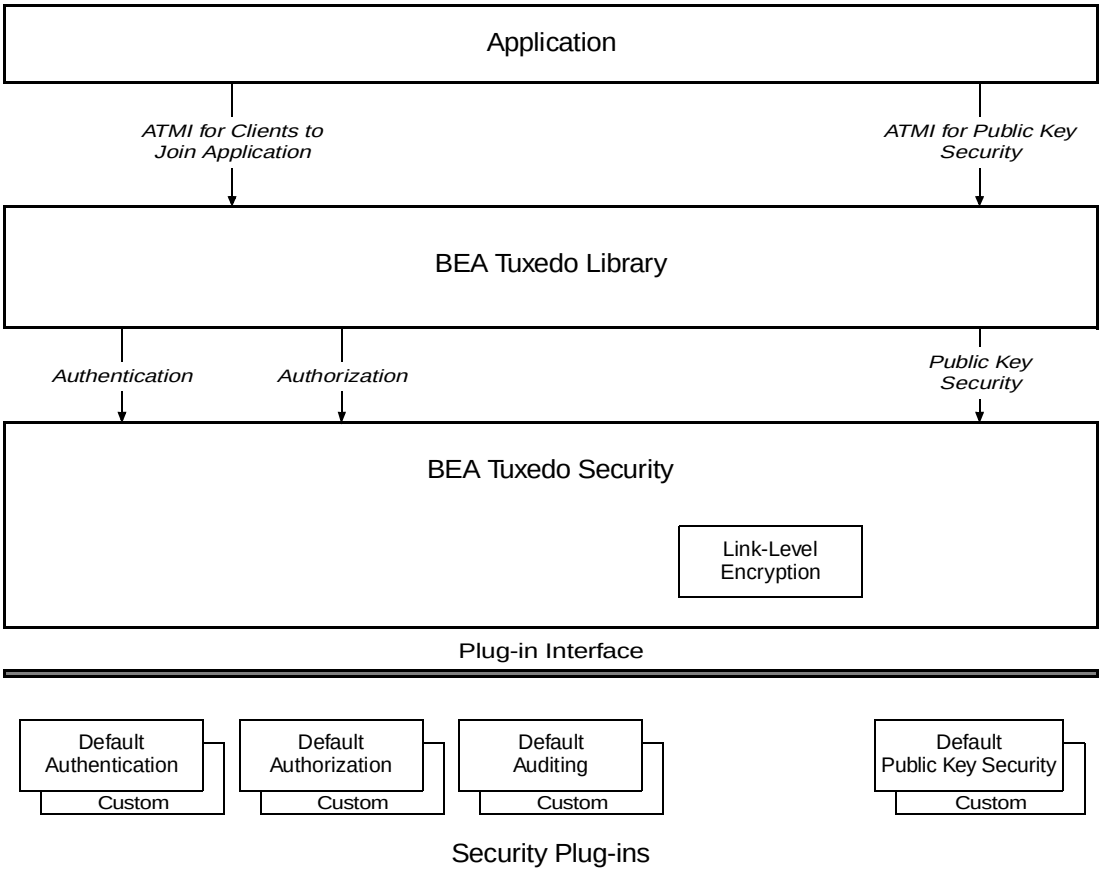
- What Programming Security Means
- Programming an Application with Security
- Writing Security Code So Client Programs Can Join the Application
- Writing Security Code to Protect Data Integrity and Privacy

What Programming Security Means

Programming security is the task of writing security code for BEA Tuxedo applications. In addition to the code that expresses the logic of the program, application programmers use the Application-to-Transaction Monitor Interface (ATMI) to link their application code with the BEA Tuxedo transaction monitor. The ATMI programming interfaces enable communication among application clients and servers running under the control of the BEA Tuxedo transaction monitor. C and COBOL implementations of the ATMI are available.

As shown in the following figure, application programmers have access to the ATMI functions for authenticating users and controlling user access, and for incorporating public key encryption techniques into their applications. Also shown is the absence, at the application level, of ATMI functions for auditing or link-level encryption. Auditing is accessed at the BEA Tuxedo system level, and link-level encryption is configured by the application administrator.

Figure 3-1 Programming BEA Tuxedo Security



See Also

- “Programming an Application with Security” on page 3-3
- “What Security Means” on page 1-1
- “What Administering Security Means” on page 2-1

Programming an Application with Security

The BEA Tuxedo system offers various ATMI functions for different security needs.

If you are writing security code for . . .	Then you use the ATMI functions available for . . .
Client programs so that clients can join a BEA Tuxedo application and access application services	Clients joining an application, which in turn invoke system-level calls to the authentication and authorization plug-ins
Both client and server programs to protect the integrity and privacy of the data they exchange	Public key security, which supports end-to-end digital signing and data encryption

See Also

- "Setting Up the Programming Environment" on page 3-3

Setting Up the Programming Environment

To be able to write security code, an application programmer needs:

- Access to BEA Tuxedo libraries and commands
- Read and execute permissions on the directories and files in the BEA Tuxedo system directory structure

To obtain access to the required libraries and commands, you must set the `TUXCONFIG`, `TUXDIR`, `APPDIR`, and other environment variables in your environment. For details, see "How to Set Your Environment" on page 1-2 in *Administering a BEA Tuxedo Application at Run Time*.

The application administrator is responsible for setting the permissions on directories and files. See your administrator to get the permissions you need.

See Also

- "Writing Security Code So Client Programs Can Join the Application" on page 3-4
- "Writing Security Code to Protect Data Integrity and Privacy" on page 3-15

Writing Security Code So Client Programs Can Join the Application

Client programs are responsible for gathering data from outside the application or computer, bundling the data into messages, and forwarding the messages to servers for processing. Client programs are made available to users through devices such as automatic teller machines (ATMs), data entry terminals, and graphics devices.

For default authentication and authorization, application security may be set to one of five levels. At the lowest level, no authentication is performed. At the highest level, an access control checking feature determines which users can execute a service, post an event, or enqueue (or dequeue) a message on an application queue. Setting the security level for an application is the responsibility of the application administrator.

An application programmer needs to perform two tasks so that a client program can join a BEA Tuxedo application:

- Get the security data for the specific client process
- Pass that data to the BEA Tuxedo system

The following pseudo-code summarizes the operation of a basic client program. The security-related statements are highlighted in bold.

Listing 3-1 Pseudo-code for a Client

```
main()
 {
      call tpchkauth() to check security level of application
      get usrname, cltname
      prompt for application password
      prompt for per-user password
      allocate a TPINIT buffer
      place initial client identification into TPINIT buffer
      call tpinit() to enroll as a client of the application
      allocate buffer
      do while true {
           place user input in buffer
           send service request
           receive reply
           pass reply to user }
      leave application
 }
```

Most of the statements in the preceding listing are implemented by ATMI functions in either C or COBOL. The preceding listing shows only the C language implementation.

A client program written in C uses `tpinit(3c)` to comply with the level of security set for the application and to join the application. The argument to `tpinit()` is a pointer to a `TPINIT` buffer. To perform the same tasks in a COBOL application, a client program calls `TPINITIALIZE(3cbl)`, specifying a pointer to a `TPINFDEF-REC` record as an argument.

See Also

- "Getting Security Data" on page 3-6
- "Joining the Application" on page 3-8
- "Writing Clients" on page 4-1 in *Programming a BEA Tuxedo Application Using C* and *Programming a BEA Tuxedo Application Using COBOL*
- `tpinit(3c)` in *BEA Tuxedo C Function Reference*
- `TPINITIALIZE(3cbl)` in *BEA Tuxedo COBOL Function Reference*

- "Administering Public Key Security" on page 2-41
- "Administering Authorization" on page 2-34
- "Default Authentication and Authorization" on page 1-44
- "Programming an Application with Security" on page 3-3

Getting Security Data

For general-purpose client programs that are written to work with a variety of applications, the BEA Tuxedo system provides an ATMI function that enables a client to determine the level of security required by the application that the client is trying to join. This ATMI function, implemented as `tpchkauth(3c)` for C and `TPCHKAUTH(3cbl)` for COBOL, is designed to work with applications using default authentication and authorization. The `tpchkauth()` and `TPCHKAUTH()` functions can also be used in applications in which custom authentication and/or authorization is used. How they are used, however, depends on how the custom security features are implemented. For the most part, this discussion focuses on default authentication and authorization.

An application programmer writing in C uses `tpchkauth()` to check the application's security level before calling `tpinit(3c)`, so that the client program can prompt for the application password and the user authentication data needed for the `tpinit()` call; `tpchkauth()` is called without arguments.

An application programmer writing in COBOL uses `TPCHKAUTH()` for the same purpose before calling `TPINITIALIZE(3cbl)`. The syntax and functionality of `TPCHKAUTH(3cbl)` and `TPINITIALIZE(3cbl)` are the same as those of `tpchkauth(3c)` and `tpinit(3c)`.

The `tpchkauth()` function (or `TPCHKAUTH()` routine) returns one of the following values.

`TPNOAUTH`
: Nothing is required beyond the normal operating system login and file permission security. `TPNOAUTH` is returned for security level `NONE`.

TPSYSAUTH

An application password is required. The client program should prompt the user to provide the password, and should put it in the password field of the TPINIT buffer for C, or TPINFDEF-REC record for COBOL. TPSYSAUTH is returned for security level APP_PW.

The application administrator informs users of the application password, and the application programmer writes client-program code to prompt users for the application password and to put the user-supplied password, as plaintext, in the password field of the TPINIT buffer or TPINFDEF-REC record. The password should not be displayed on the user's screen.

BEA Tuxedo system-supplied client programs, such as ud, wud(1), prompt for an application password. ud() allows fielded buffers to be read from standard input and sent to a service.

TPAPPAUTH

The application password is required. The client is expected to provide a value to be passed to the authentication service in the data field of the TPINIT buffer for C, or the TPINFDEF-REC record for COBOL. TPAPPAUTH is returned for security level USER_AUTH, ACL, or MANDATORY_ACL.

The application programmer writes client-program code to furnish additional information for the application authentication service, which is provided by the AUTHSVR server for default authentication and authorization. AUTHSVR is configured by the administrator to validate the per-user authentication information with client and user names, indicating whether the client program is allowed to join the application.

See Also

- "Joining the Application" on page 3-8
- "Writing Clients" on page 4-1 in *Programming a BEA Tuxedo Application Using C* and *Programming a BEA Tuxedo Application Using COBOL*
- tpinit(3c) and tpchkauth(3c) in *BEA Tuxedo C Function Reference*
- TPINITIALIZE(3cbl) and TPCHKAUTH(3cbl) in *BEA Tuxedo COBOL Function Reference*

- "Default Authentication and Authorization" on page 1-44
- "Programming an Application with Security" on page 3-3

Joining the Application

In a secure BEA Tuxedo application, it is necessary to pass security information to the BEA Tuxedo system via a `TPINIT` buffer for C, or a `TPINFDEF-REC` record for COBOL. The `TPINIT` buffer is a special typed buffer used by a client program to pass client identification and authentication information to the system as the client attempts to join the application. The `TPINFDEF-REC` record serves the same purpose in a COBOL application.

`TPINIT` is defined in the `atmi.h` header file, and `TPINFDEF-REC` is defined in the COBOL `COPY` file. They have the following structures.

TPINIT Structure

```
char   usrname[MAXTIDENT+2];
char   cltname[MAXTIDENT+2];
char   passwd[MAXTIDENT+2];
char   grpname[MAXTIDENT+2];
long   flags;
long   datalen;
long   data;
```

Note: `MAXTIDENT` may contain up to 30 characters.

TPINFDEF-REC Structure

```
05 USRNAME             PIC X(30).
05 CLTNAME             PIC X(30).
05 PASSWD              PIC X(30).
05 GRPNAME             PIC X(30).
05 NOTIFICATION-FLAG   PIC S9(9) COMP-5.
   88 TPU-SIG            VALUE 1.
   88 TPU-DIP            VALUE 2.
   88 TPU-IGN            VALUE 3.
05 ACCESS-FLAG         PIC S9(9) COMP-5.
   88 TPSA-FASTPATH      VALUE 1.
   88 TPSA-PROTECTED     VALUE 2.
05 DATLEN              PIC S9(9) COMP-5.
```

The fields in the TPINIT buffer/ TPINFDEF-REC record are described in the following table.

Table 3-1 Fields in TPINIT Buffer/ TPINFDEF-REC Record

TPINIT Fields	TPINFDEF-REC Fields	Description
usrname	USRNAME	User name.* A null-terminated string of up to 30 characters. The user name represents the caller; writers of client programs might use the same login names used to log in to the host operating system.
cltname	CLTNAME	Client name.* A null-terminated string of up to 30 characters. The client name represents the client program; writers of client programs might use this field to indicate the job function or role of the user when executing the client program.
passwd	PASSWD	Application password.* A null-terminated string of up to eight characters. tpinit() or TPINITIALIZE() validates this password by comparing it to the configured application password stored in the TUXCONFIG file.**
grpname	GRPNAME	Group name. A null-terminated string of up to 30 characters. This field is not related to security. The group name allows a client to be associated with a resource manager group that is defined in the UBBCONFIG file.

* This field is required for the USER_AUTH, ACL, and MANDATORY_ACL security levels provided by default authentication and authorization.

** The binary equivalent of the UBBCONFIG file; created using tmloadcf(1).

*** Usually a user password.

Table 3-1 Fields in TPINIT Buffer/ TPINFDEF-REC Record

TPINIT Fields	TPINFDEF-REC Fields	Description
flags	NOTIFICATION-FLAG TPU-SIG TPU-DIP TPU-IGN ACCESS-FLAG TPSA-FASTPATH TPSA-PROTECTED	Notification and access flags. This field is not related to security. The flag settings specify the notification mechanism and system access mode to be used for the client. Selections override (with some exceptions) the values set in the RESOURCES section of the UBBCONFIG file.
datalen	DATALEN	Length of the user-specific data*** that follows.* To get a size value for this field, writers of client programs written in C can call TPINITNEED with the number of bytes of user-specific data expected to be sent. TPINITNEED is a macro provided in the atmi.h header file.
data	N/A	User-specific data*** of no fixed length.* tpinit() or TPINITIALIZE() forwards the user-specific data to the authentication server for validation. For default authentication, the authentication server is AUTHSVR.

* This field is required for the USER_AUTH, ACL, and MANDATORY_ACL security levels provided by default authentication and authorization.

** The binary equivalent of the UBBCONFIG file; created using tmloadcf(1).

*** Usually a user password.

The client program calls tpalloc(3c) to allocate a TPINIT buffer. The following sample code prepares to pass eight bytes of application-specific data to tpinit() and enables the client to join a BEA Tuxedo application.

Listing 3-2 Allocating a TPINIT Buffer and Joining a BEA Tuxedo Application

```
    .
    .
    .
TPINIT *tpinfo;
    .
```

```
  .
  .
if ((tpinfo = (TPINIT *)tpalloc("TPINIT",(char *)NULL,
     TPINITNEED(8))) == (TPINIT *)NULL){
     Error Routine
}
  .
  .
  .
tpinit(tpinfo)  /* join a BEA Tuxedo application */
  .
  .
  .
```

When a Workstation client calls the `tpinit()` function or the `TPINITIALIZE()` routine to join an application, the following major events occur.

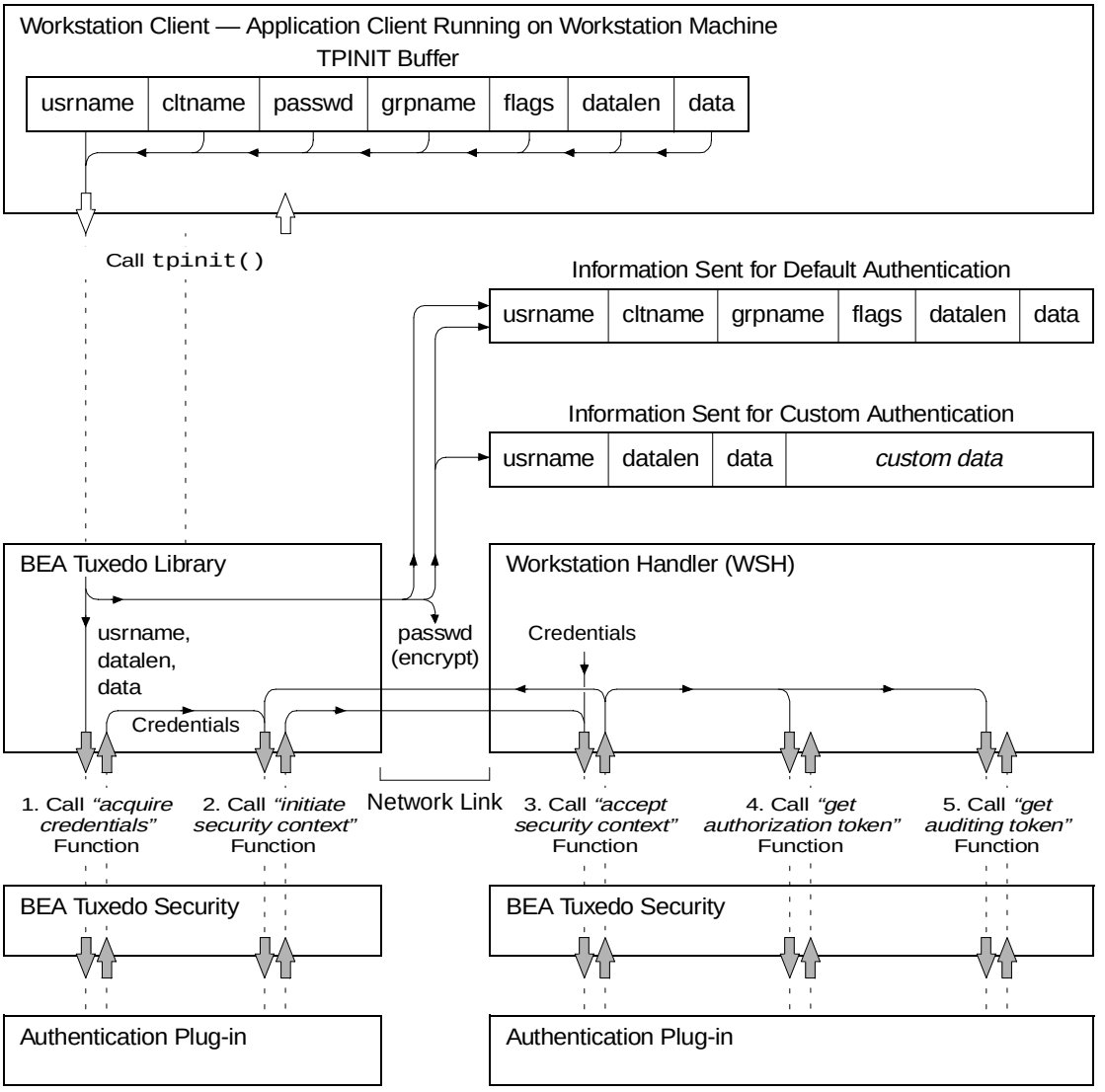
1. The *initiator* Workstation client and the *target* Workstation Listener (WSL) exchange link-level encryption (LLE) *`min-max`* values to be used to set up LLE on the link between the initiator Workstation client and the *target* WSH. LLE is described in "Link-Level Encryption" on page 1-23.
2. The initiator Workstation client and target WSH authenticate one another through the exchange of security tokens. For default authentication, a successful authentication ends with the transfer of client security data from the `TPINIT` buffer or `TPINFDEF-REC` record to the target WSH.
3. After a successful authentication, the initiator Workstation client sends another buffer to the target WSH containing the values of the `usrname`, `cltname`, and `flags` fields, to ensure that the target WSH receives this information for the authenticated Workstation client.

When a native client calls the `tpinit()` function or the `TPINITIALIZE()` routine to join an application, only authentication occurs. In essence, the native client authenticates with itself.

Transferring the Client Security Data

The following diagram demonstrate the transfer of data from the `TPINIT` buffer for a Workstation client. The transfer of data from the `TPINFDEF-REC` record is similar to what is shown in the diagram.

Figure 3-2 Transferring Data from the TPINIT Buffer for a Workstation Client



Note: The authorization procedure shown in the preceding diagram is essentially the same for a native client attempting to join an application except that no network link or WSH is involved. A native client authenticates with itself.

In the preceding diagram, notice that the information sent to the BEA Tuxedo system differs between default and custom authentication. For default authentication, the values of the `cltname`, `grpname`, and `flags` fields are delivered to the default authentication plug-in at the Workstation client by a means *other* than through the plug-in interface. However, for custom authentication, writers of client programs can include these values as well as any other values they so choose in the variable length `data` field.

For a Workstation client *and* assuming default authentication, the authentication plug-in at the Workstation client uses the `passwd`/`PASSWD` field to encrypt the information when transmitting the information over the network. The encryption algorithm used is 56-bit DES, where DES is an acronym for the Data Encryption Standard. The authentication plug-in at the target WSH uses the application password stored in the `TUXCONFIG` file to decrypt the information. For a native client, the system simply compares the `passwd`/`PASSWD` field with the application password stored in the `TUXCONFIG` file.

Note: At the Workstation client, the `passwd`/`PASSWD` field is delivered to the authentication plug-in by a means *other* than through the authentication plug-in interface. At the WSH, the application password in the `TUXCONFIG` file is delivered to the authentication plug-in through the authentication plug-in interface during application booting.

After a successful authentication of a Workstation client, the `tpinit()` function ends with the sending of another buffer to the WSH containing the values of the `usrname`, `cltname`, and `flags` fields, to ensure that the WSH receives this information for the authenticated Workstation client. Similarly, the `TPINITIALIZE()` routine ends with the sending of another buffer containing the same information. A custom authentication plug-in might not send this information to the WSH during the authentication procedure, and the WSH needs this information for reporting purposes, that is, during an invocation of the `tmadmin(1)` `printclient` (`pclt`) command.

When a Workstation or native client passes the security check, it may initiate service requests and receive replies.

Calling a Service Request Before Joining the Application

If a client calls a service request (or any ATMI function) before invoking `tpinit()` or `TPINITIALIZE()` *and* assuming the `SECURITY` configuration for the target application is *not* set or is set to `NONE`, the BEA Tuxedo system automatically invokes `tpinit()`/ `TPINITIALIZE()` with a `NULL` parameter. This behavior has the following consequences:

- The `TPINIT`/ `TPINFDEF-REC` feature cannot be used.
- Default values are used for client naming, unsolicited notification type, and system access mode.
- The client cannot be associated with a resource manager group.
- An application password cannot be specified.

If a client calls a service request (or any ATMI function) before invoking `tpinit()` or `TPINITIALIZE()` *and* assuming the `SECURITY` configuration for the target application is set to `APP_PW`, `USER_AUTH`, `ACL`, or `MANDATORY_ACL`, the BEA Tuxedo system rejects the service request.

See Also

- "Writing Clients" on page 4-1 in *Programming a BEA Tuxedo Application Using C* and *Programming a BEA Tuxedo Application Using COBOL*
- `tpinit(3c)` and `tpalloc(3c)` in *BEA Tuxedo C Function Reference*
- `TPINITIALIZE(3cbl)` in *BEA Tuxedo COBOL Function Reference*
- "Default Authentication and Authorization" on page 1-44
- "Programming an Application with Security" on page 3-3

Writing Security Code to Protect Data Integrity and Privacy

Public key security comprises end-to-end digital signing and data encryption. Both features are supported by BEA Tuxedo ATMI functions. Applications protected by public key security are much safer for use across the Internet than programs in which this type of security is not used.

The capabilities that make end-to-end digital signing and data encryption possible are message-based digital signature and message-based encryption. Both capabilities are built upon the *PKCS-7 standard*, which is one of a set of Public-Key Cryptography Standards (PKCS) developed by RSA Laboratories in cooperation with several other leading communications companies.

Message-based digital signature ensures data integrity and non-repudiation by having the sending party bind proof of its identity to a specific message buffer. Message-based encryption protects the confidentiality of messages; only parties for whom messages are intended can decrypt and read them.

Because the unit of digital signing and encryption is a BEA Tuxedo message buffer, both capabilities are compatible with existing BEA Tuxedo programming interfaces and communication paradigms. It is possible for a message buffer to be both signed and encrypted. There is no required relationship between the number of digital signatures and the number of *encryption envelopes* associated with a message buffer.

Note: Each encryption envelope identifies a recipient of the message, and contains information needed by the recipient to decrypt the message.

ATMI for Public Key Security

The ATMI for public key security is a compact set of functions used to:

- Open and close key resources
- View and change key optional parameters
- Sign and seal (encrypt) message buffers
- Access the digital signature and encryption information associated with a message buffer
- Convert a typed message buffer into an exportable, machine-independent string representation, which includes the generation of any digital signatures or encryption envelopes associated with the buffer

The ATMI for public key security is available in both C and COBOL implementations. The BEA Tuxedo COBOL language binding, however, does not support *message buffers*; thus, explicit signature, encryption, and query operations on individual buffers cannot be used in a COBOL application. However, key management interfaces do have a COBOL language binding, which enables signature generation in the `AUTOSIGN` mode and encryption-envelope generation in the `AUTOENCRYPT` mode. All operations related to automatic signature verification or automatic decryption apply to COBOL client and server processes.

Note: The COBOL `TPKEYDEF` record is used to manage public-private keys for performing message-based digital signature and encryption operations. See "COBOL Language ATMI Return Codes and Other Definitions" in the introduction part of *BEA Tuxedo COBOL Function Reference* for a description of the `TPKEYDEF` record.

The following tables summarize the ATMI for public key security. Each function is also documented in *BEA Tuxedo C Function Reference* / *BEA Tuxedo COBOL Function Reference*.

Table 3-2 C Functions in ATMI for Public Key Security

Use this function . . .	To . . .
`tpkey_open(3c)`	Open a key handle for digital signature generation, message encryption, or message decryption. Keys are represented and manipulated via handles. A handle has data associated with it that is used by the ATMI to locate or access the item named by the handle. A key may play one or more of the following roles: ■ *Signature Generation* The key identifies the calling process as being authorized to generate a digital signature under the *principal*'s identity. (A principal may be a person or a process.) Calling `tpkey_open()` with the principal's name and either the `TPKEY_SIGNATURE` or `TPKEY_AUTOSIGN` flag returns a handle to the principal's private key and digital certificate. ■ *Signature Verification* The key represents the principal associated with a digital signature. Signature verification does not require a call to `tpkey_open()`; the verifying process uses the public key specified in the digital certificate accompanying the digitally signed message to verify the signature. ■ *Encryption* The key represents the intended principal of an encrypted message. Calling `tpkey_open()` with the principal's name and either the `TPKEY_ENCRYPT` or `TPKEY_AUTOENCRYPT` flag returns a handle to the principal's public key via the principal's digital certificate. ■ *Decryption* The key identifies the calling process as being authorized to decrypt a private message for the intended principal. Calling `tpkey_open()` with the principal's name and the `TPKEY_DECRYPT` flag returns a handle to the principal's private key and digital certificate.

Table 3-2 C Functions in ATMI for Public Key Security

Use this function . . .	To . . .
tpkey_getinfo(3c)	Get information associated with a key handle. Some information is specific to a cryptographic service provider, but the following set of attributes is supported by all providers: ■ PRINCIPAL The name of the *principal* associated with the specified key (key handle). A principal may be a person or a process, depending on how an application developer sets up public key security. Any principal specified in an application's UBBCONFIG file using the SEC_PRINCIPAL_NAME parameter become the identity of one or more system processes. (See "Specifying Principal Names" on page 2-11 and "Initializing Decryption Keys Through the Plug-ins" on page 2-50 for more detail.) ■ PKENCRYPT_ALG An ASN.1 Distinguished Encoding Rules (DER) *object identifier* of the public key algorithm used by the key for public key encryption. See the tpkey_getinfo(3c) reference page for details. ■ PKENCRYPT_BITS The key length of the public key algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. ■ SIGNATURE_ALG An ASN.1 DER *object identifier* of the digital signature algorithm used by the key for digital signature. See the tpkey_getinfo(3c) reference page for details. ■ SIGNATURE_BITS The key length of the digital signature algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. ■ ENCRYPT_ALG An ASN.1 DER *object identifier* of the symmetric key algorithm used by the key for bulk data encryption. See the tpkey_getinfo(3c) reference page for details. ■ ENCRYPT_BITS The key length of the symmetric key algorithm. The value must be within the range of 40 to 128 bits, inclusive. ■ DIGEST_ALG An ASN.1 DER *object identifier* of the message digest algorithm used by the key for digital signature. See the tpkey_getinfo(3c) reference page for details. ■ PROVIDER The name of the cryptographic service provider. ■ VERSION The version number of the cryptographic service provider's software.

Table 3-2 C Functions in ATMI for Public Key Security

Use this function . . .	To . . .
`tpkey_setinfo(3c)`	Set optional attribute parameters associated with a key handle. A core set of key handle attributes is identified in the preceding description of `tpkey_getinfo()`. Other attributes, specific to a certain cryptographic service provider, may also be available.
`tpkey_close(3c)`	Close a previously opened key handle. A key handle may be opened explicitly using `tpkey_open()`, or implicitly (automatically) using `tpenvelope()`.
`tpsign(3c)`	Mark a typed message buffer for digital signature. The public key software generates the digital signature just before the message is sent.
`tpseal(3c)`	Mark a typed message buffer for encryption. The public key software encrypts the message just before the message is sent.
`tpenvelope(3c)`	Access the digital signature and encryption information associated with a typed message buffer. `tpenvelope()` returns status information about the digital signatures and encryption envelopes attached to a particular message buffer. It also returns the key handle associated with each digital signature or encryption envelope. The key handle for a digital signature identifies the signer, and the key handle for an encryption envelope identifies the recipient of the message.
`tpexport(3c)`	Convert a typed message buffer into an exportable, machine-independent (externalized) string representation. `tpexport()` generates any digital signatures or encryption envelopes associated with a typed message buffer just before it converts that buffer into an externalized string representation. An externalized string representation can be transmitted between processes, machines, or domains through any communication mechanism. It can be archived on permanent storage.
`tpimport(3c)`	Convert an externalized string representation back into a typed message buffer. During the conversion, `tpimport()` decrypts the message, if necessary, and verifies any associated digital signatures.

Table 3-3 COBOL Routines in ATMI for Public Key Security

Use this routine . . .	To . . .
TPKEYOPEN(3cbl)	Open a key handle for digital signature generation, message encryption, or message decryption. Keys are represented and manipulated via handles. A handle has data associated with it that is used by the ATMI to locate or access the item named by the handle. A key may play one or more of the following roles: ■ *Signature Generation* The key identifies the calling process as being authorized to generate a digital signature under the *principal*'s identity. (A principal can be a person or a process.) Calling TPKEYOPEN() with the principal's name and the TPKEY-SIGNATURE and TPKEY-AUTOSIGN settings returns a handle to the principal's public key and enables signature generation in AUTOSIGN mode. The public key software generates and attaches the digital signature to the message just before the message is sent. ■ *Signature Verification* The key represents the principal associated with a digital signature. Signature verification does not require a call to TPKEYOPEN(); the verifying process uses the public key specified in the digital certificate accompanying the digitally signed message to verify the signature. ■ *Encryption* The key represents the intended principal of an encrypted message. Calling TPKEYOPEN() with the principal's name and the TPKEY-ENCRYPT and TPKEY-AUTOENCRYPT settings returns a handle to the principal's public key (via the principal's digital certificate) and enables encryption in AUTOENCRYPT mode. The public key software encrypts the message and attaches an encryption envelope to the message; the encryption envelope enables the receiving process to decrypt the message. ■ *Decryption* The key identifies the calling process as being authorized to decrypt a private message for the intended principal. Calling TPKEYOPEN() with the principal's name and the TPKEY-DECRYPT setting returns a handle to the principal's private key and digital certificate.

Table 3-3 COBOL Routines in ATMI for Public Key Security

Use this routine . . .	To . . .
TPKEYGETINFO(3cbl)	Get information associated with a key handle. Some information is specific to a cryptographic service provider, but the following set of attributes is supported by all providers: ■ PRINCIPAL The name of the *principal* associated with the specified key (key handle). A principal may be a person or a process, depending on how an application developer sets up public key security. Any principal specified in an application's UBBCONFIG file using the SEC_PRINCIPAL_NAME parameter become the identity of one or more system processes. (See "Specifying Principal Names" on page 2-11 and "Initializing Decryption Keys Through the Plug-ins" on page 2-50 for more detail.) ■ PKENCRYPT_ALG An ASN.1 Distinguished Encoding Rules (DER) *object identifier* of the public key algorithm used by the key for public key encryption. See the TPKEYGETINFO(3cbl) reference page for details. ■ PKENCRYPT_BITS The key length of the public key algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. ■ SIGNATURE_ALG An ASN.1 DER *object identifier* of the digital signature algorithm used by the key for digital signature. See the TPKEYGETINFO(3cbl) reference page for details. ■ SIGNATURE_BITS The key length of the digital signature algorithm (RSA modulus size). The value must be within the range of 512 to 2048 bits, inclusive. ■ ENCRYPT_ALG An ASN.1 DER *object identifier* of the symmetric key algorithm used by the key for bulk data encryption. See the TPKEYGETINFO(3cbl) reference page for details. ■ ENCRYPT_BITS The key length of the symmetric key algorithm. The value must be within the range of 40 to 128 bits, inclusive. ■ DIGEST_ALG An ASN.1 DER *object identifier* of the message digest algorithm used by the key for digital signature. See the TPKEYGETINFO(3cbl) reference page for details. ■ PROVIDER The name of the cryptographic service provider. ■ VERSION The version number of the cryptographic service provider's software.

Table 3-3 COBOL Routines in ATMI for Public Key Security

Use this routine . . .	To . . .
TPKEYSETINFO(3cbl)	Set optional attribute parameters associated with a key handle. A core set of key handle attributes is identified in the preceding description of TPKEYGETINFO(). Other attributes, specific to a certain cryptographic service provider, may also be available.
TPKEYCLOSE(3cbl)	Close a key handle previously opened using TPKEYOPEN().

Recommended Uses of Public Key Security

- Use tpkey_close() to release key handles used for digital signature generation or for data decryption as soon as they are no longer needed.
- To inhibit replay attacks, generate digital signatures only on message buffers that contain details identifying a specific operation. For example, a buffer that contains the message "Your deposit is confirmed" is dangerously vague. An attacker who intercepts such a message can easily re-use it. On the other hand, a message that contains many operation-specific details is much safer. An attacker who intercepts a message such as the one that follows will not be able to re-use it easily: "John Smith's deposit of $100.00, account 987654321, confirmation code 123456789, 7/31/2001, is confirmed."

See Also

- "Sending and Receiving Signed Messages" on page 3-23
- "Sending and Receiving Encrypted Messages" on page 3-34
- "Examining Digital Signature and Encryption Information" on page 3-52
- "Externalizing Typed Message Buffers" on page 3-59
- "Public Key Security" on page 1-29
- "Administering Public Key Security" on page 2-41
- "Programming an Application with Security" on page 3-3

Sending and Receiving Signed Messages

Message-based digital signature provides end-to-end authentication and message integrity protection. For a diagram that illustrates how it works, see the figure "BEA Tuxedo PKCS-7 End-to-End Digital Signing" on page 1-35.

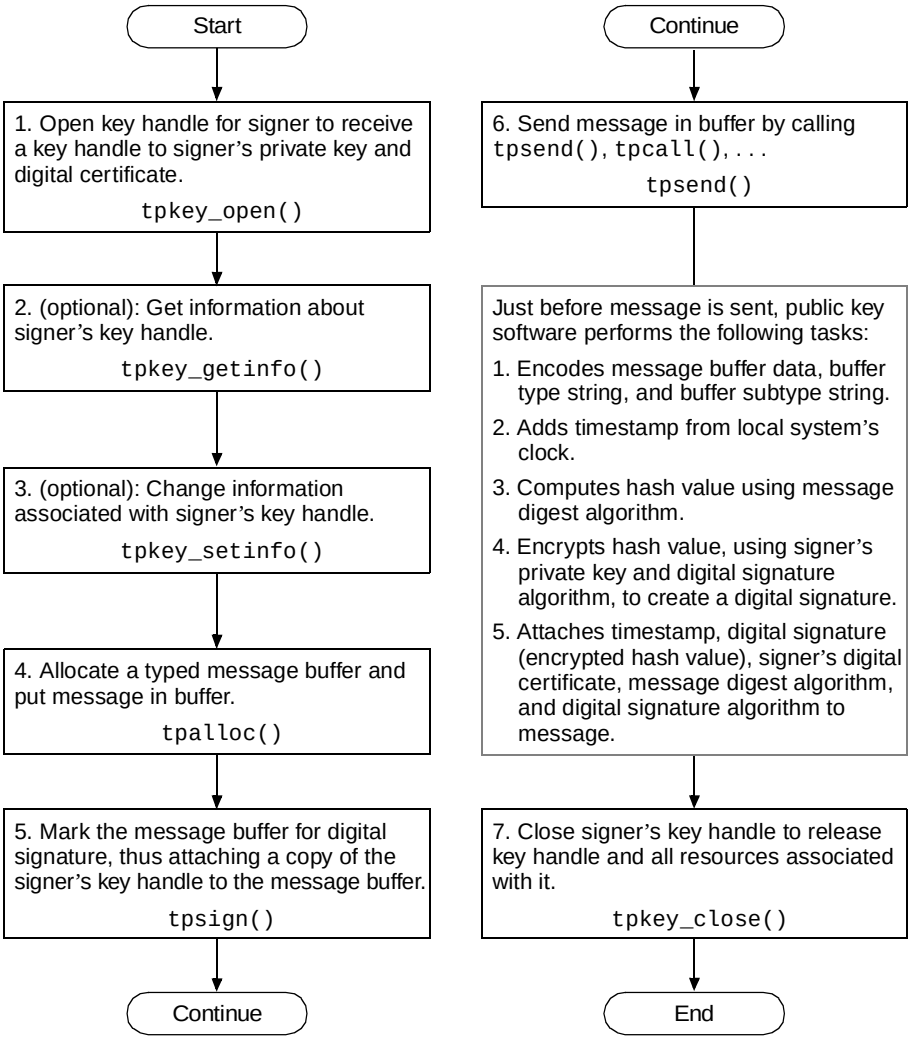
To add a digital signature to a BEA Tuxedo message buffer, the originating process or user signs the message buffer. This signature contains a cryptographically secure checksum of the message buffer's content and a timestamp based on the signer's local clock.

Any party with access to the message buffer can verify that the signing party's signature is authentic, that the message buffer content is unchanged, and that the timestamp is within a configured tolerance of the verifier's local clock. In addition, time-independent verification by a third party guarantees non-repudiation: the originating process or user cannot later deny authorship or claim the message was altered.

Writing Code to Send Signed Messages

The following flowchart provides the procedure for writing code to send signed messages.

Figure 3-3 Procedure for Sending Signed Messages



For details about these steps and insight into how the system signs a message buffer, see the following topics.

Step 1: Opening a Key Handle for Digital Signature

Call the `tpkey_open(3c)` function or `TPKEYOPEN(3cbl)` routine to make the private key and the associated digital certificate of the signer available to the originating process. The private key is highly protected, and possession of it is equivalent to possessing the signer's identity.

In order to access the signer's private key, the originating process must prove its right to act as the signer. Proof requirements depend on the implementation of the public key plug-in interface. The default public key implementation requires a secret password from the calling process.

When the originating process calls `tpkey_open()` to open the key handle, it specifies either the `TPKEY_SIGNATURE` or `TPKEY_AUTOSIGN` flag to indicate that the handle will be used to digitally sign a message buffer. Typically, a client makes this call after calling `tpinit()`, and a server makes this call as part of initializing through `tpsvrinit()`.

Opening a key handle with the `TPKEY_AUTOSIGN` flag enables automatic signature generation: subsequently, the originating process signs message buffers automatically whenever they are sent. Using the `TPKEY_AUTOSIGN` flag is beneficial for three reasons:

- Less work is required from application programmers because fewer ATMI calls are required when operating in a secure application.
- Existing applications can leverage digital signature technology with minimal coding changes.
- The possibility of programming errors that might result in an unsigned buffer being sent over an insecure network is reduced.

The following example code shows how to open a signer's key handle. `TPKEY` is a special data type defined in the `atmi.h` header file.

Listing 3-3 Opening a Signer's Key Handle—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY sdo_key;
   char *sdo_location;
  .
  .
  .
   if (tpkey_open(&sdo_key, "sdo", sdo_location,
      NULL, 0, TPKEY_SIGNATURE) == -1) {
      (void) fprintf(stderr, "tpkey_open sdo failed
         tperrno=%d(%s)\n", tperrno, tpstrerror(tperrno));
      exit(1);
   }
  .
  .
  .
}
```

Step 2 (Optional): Getting Key Handle Information

You may want to get information about a signer's key handle to establish the validity of the key. To do so, call the `tpkey_getinfo(3c)` function or `TPKEYGETINFO(3cbl)` routine. While some of the information returned may be specific to a cryptographic service provider, a core set of attributes is common to all providers.

The default public key implementation supports the following signature modes for computing signatures on a message buffer:

- MD5 message digest algorithm with RSA public key signature
- SHA-1 message digest algorithm with RSA public key signature

The message digest algorithm is controlled by the `DIGEST_ALG` key attribute, and the public key signature is controlled by the `SIGNATURE_ALG` key attribute. Public key sizes from 512 to 2048 bits are supported, to allow a wide range of safety and performance options. The public key size is controlled by the `SIGNATURE_BITS` key attribute.

The default public key implementation recognizes only those digital certificate signatures that are created with these algorithm and key size choices.

The following example code shows how to get information about a signer's key handle.

Listing 3-4 Getting Information About a Signer's Key Handle—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY sdo_key;
   char principal_name[PNAME_LEN];
   long pname_len = PNAME_LEN;
  .
  .
  .
   if (tpkey_getinfo(sdo_key, "PRINCIPAL",
      principal_name, &pname_len, 0) == -1) {
      (void) fprintf(stdout, "Unable to get information
         about principal: %d(%s)\n",
            tperrno, tpstrerror(tperrno));
  .
  .
  .
      exit(1);
   }
  .
  .
  .
}
```

Step 3 (Optional): Changing Key Handle Information

To set optional attributes associated with a signer's key handle, call the `tpkey_setinfo(3c)` function or `TPKEYSETINFO(3cbl)` routine. Key handle attributes vary, depending on the cryptographic service provider.

The following example code shows how to change information associated with a signer's key handle.

Listing 3-5 Changing Information Associated with a Signer's Key Handle—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY sdo_key;
   static const unsigned char sha1_objid[] = {
      0x06, 0x05, 0x2b, 0x0e, 0x03, 0x02, 0x1a
   };
  .
  .
  .
   if (tpkey_setinfo(sdo_key, "DIGEST_ALG", (void *) sha1_objid,
      sizeof(sha1_objid), 0) == -1) {
      (void) fprintf(stderr, "tpkey_setinfo failed
         tperrno=%d(%s)\n",
            tperrno, tpstrerror(tperrno));
      return(1);
   }
  .
  .
  .
}
```

Step 4: Allocating a Buffer and Putting a Message in the Buffer

To allocate a typed message buffer, call the `tpalloc(3c)` function. Then put a message in the buffer.

Step 5: Marking the Buffer for Digital Signature

To mark, or register, the message buffer for digital signature, call the `tpsign(3c)` function. By calling this function, you attach a copy of the signer's key handle to the message buffer. If you open the key with the `TPKEY_AUTOSIGN` flag, each message that you send is automatically marked for digital signature without an explicit call to `tpsign()`; signature parameters are stored and associated with the buffer for later use.

Note: In COBOL applications, use the `AUTOSIGN` settings member to create a digital signature. See `TPKEYOPEN(3cbl)`.

The following example code shows how to mark a message buffer for digital signature.

Listing 3-6 Marking a Message Buffer For Digital Signature—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY sdo_key;
   char *sendbuf, *rcvbuf;
  .
  .
  .
   if (tpsign(sendbuf, sdo_key, 0) == -1) {
      (void) fprintf(stderr, "tpsign failed tperrno=%d(%s)\n",
         tperrno, tpstrerror(tperrno));
      tpfree(rcvbuf);
      tpfree(sendbuf);
      tpterm();
      (void) tpkey_close(sdo_key, 0);
      exit(1);
   }
  .
  .
  .
}
```

Step 6: Sending the Message

After the message buffer has been marked for digital signature, transmit the message buffer using one of the following C functions or COBOL routines:

- `tpcall()` or `TPCALL`
- `tpbroadcast()` or `TPBROADCAST`
- `tpconnect()` or `TPCONNECT`
- `tpenqueue()` or `TPENQUEUE`
- `tpforward()`
- `tpnotify()` or `TPNOTIFY`

- `tppost()` or `TPPOST`
- `tpreturn()` or `TPRETURN`
- `tpsend()` or `TPSEND`

Step 7: Closing the Signer's Key Handle

Call the `tpkey_close(3c)` function or `TPKEYCLOSE(3cbl)` routine to release the signer's key handle and all resources associated with it.

How the System Generates a Digital Signature

Just before a message buffer is sent, the public key software digitally signs the message. If a signed buffer is transmitted more than once, the software generates a new signature for each communication. This process makes it possible to modify a message buffer after marking the buffer to be digitally signed.

The public key software generates a digital signature by performing the following three-step procedure.

1. digest[message_buffer_data + buffer_type_string + buffer_subtype_string] = hash1
2. digest[hash1 + local_timestamp + PKCS-7_message_type] = hash2
3. {hash2}signer's_private_key = encrypted_hash2 = digital_signature

The notation *digest*[*something*] means that a hash value has been computed for *something* using a message digest algorithm—in this case, MD5 or SHA-1. The notation {*something*}*key* means that *something* has been encrypted or decrypted using *key*. In this case, the computed hash value is encrypted using the signer's private key.

Signature Timestamp

A digital signature includes a timestamp from the local system's clock. Inclusion of such a timestamp ensures that any tampering with the timestamp value will be detected when the recipient verifies the signature. In addition, a copy of the timestamp accompanies the digitally signed message when the message is routed to its destination.

Time resolution is to the second. Timestamps are stored in PKCS-9 `SigningTime` format.

Multiple Signatures

More than one signature can be associated with a message buffer, which means that any number of signers can sign a message buffer in parallel. A signer can be a person or a process. Each signer signs the message buffer using his, her, or its private key.

Different signatures may be based on different message digest or digital signature algorithms. If two signers use the same message digest and digital signature algorithm, the hash value is computed for only one of them.

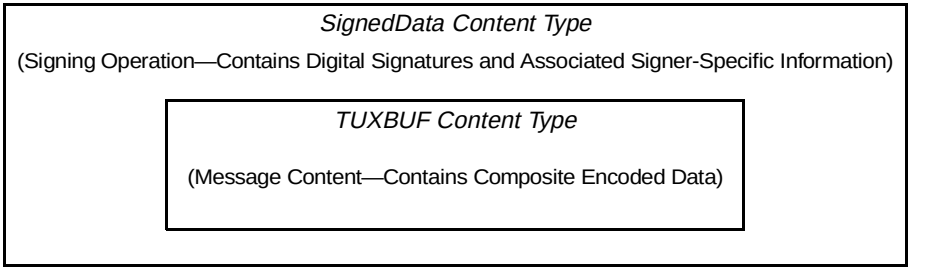
Signed Message Content

A digitally signed message buffer is represented in the PKCS-7 format as a version 1 `SignedData` content type. The `SignedData` content type, as used by the BEA Tuxedo system, consists of the following items:

- One or more digital signatures, each with its own set of signer-specific information, such as:
  - Signer's X.509v3 certificate
  - Message digest and digital signature algorithm identifiers
  - Timestamp based on the local clock
- Message content, which is a composite of message buffer data, buffer type string, and buffer subtype string represented in the BEA Tuxedo encoded format. The encoded format allows a message buffer's signature to be verified on any machine architecture.

As shown in the following figure, the message content is enveloped by `SignedData` content type.

Figure 3-4 SignedData Content Type



How a Signed Message Is Received

No application code is needed to receive a signed message buffer. The public key software automatically verifies the attached digital signatures and passes the message to the receiving process.

Upon receiving a signed message buffer, the public key software, operating on behalf of the receiving process, performs the following tasks.

1. Reads the digital signature information attached to the received message, including the signer's digital certificate, message digest algorithm, digital signature algorithm, and signature timestamp.
2. Decrypts the attached digital signature (encrypted hash value) using the signer's public key (found in the signer's digital certificate) and the digital signature algorithm.
3. Re-computes the hash value for the received message, as shown in the following two-step procedure.

   a. digest[message_buffer_data + buffer_type_string + buffer_subtype_string] = hash1

   b. digest[hash1 + received_timestamp + PKCS-7_message_type] = hash2

   The notation *digest*[*something*] means that a hash value has been computed for *something* using a message digest algorithm—in this case, MD5 or SHA-1.
4. Compares the re-computed hash value with the received hash value; if the two are not identical, discards the message buffer.
5. Compares the received timestamp with the local system's clock; if the timestamp is not within a configured tolerance, discards the message buffer.
6. If the message buffer successfully passes the checks performed in Steps 4 and 5, the public key software decodes the message buffer data, buffer type string, and buffer subtype string, and then passes the message to the receiving process. This step reverses the encoding performed by the originating process. (The BEA Tuxedo encoded format allows a message buffer's signature to be verified on any machine architecture.)

Note: If none of the attached digital signatures can be verified, the receiving process does *not* receive the message buffer. Moreover, the receiving process has no knowledge of the message buffer.

Verifying Digital Signatures

The public key software automatically verifies digital signatures whenever a signed message buffer enters a client process, server process, or any system process that needs to access the content of the message buffer. If a system process is acting as a *conduit* (that is, if it is not reading the content of the message), then the attached digital signatures need not be verified. Bridges and Workstation Handlers (WSHs) are examples of system processes acting as conduits.

The signature timestamp is based on an unsynchronized clock, and therefore cannot be fully trusted, especially if the signature is performed on a PC or personal workstation. However, a server may reject requests with timestamps that are too old or dated too far into the future. The capability to reject a request based on the timestamp provides a measure of protection against replay attacks.

Verifying and Transmitting an Input Buffer's Signatures

If a message buffer is passed to an ATMI function (such as `tpacall()`) as an input parameter, the public key software verifies any signatures previously attached to the message and then forwards the message. This behavior enables a secure, verified transfer of information with signatures from multiple processes.

If a server modifies a received message buffer and then forwards the buffer, the original signature is no longer valid. In this case, the public key software detects the invalid signature and silently discards it. For an example of the process, see "Discarding an Input Buffer's Encryption Envelopes" on page 3-49.

Replacing an Output Buffer's Signatures

If a message buffer is passed to an ATMI function (such as `tpgetreply()`) as an output parameter, the public key software deletes any signature information associated with the buffer. This information includes any *pending* signatures and signatures from previous uses of the buffer. (A pending signature is a signature that is registered with a message buffer.)

New signature information might be associated with the new buffer content after successful completion of this operation.

See Also

- "Sending and Receiving Encrypted Messages" on page 3-34
- "Examining Digital Signature and Encryption Information" on page 3-52
- "Externalizing Typed Message Buffers" on page 3-59
- "Public Key Security" on page 1-29
- "Administering Public Key Security" on page 2-41
- "Programming an Application with Security" on page 3-3

Sending and Receiving Encrypted Messages

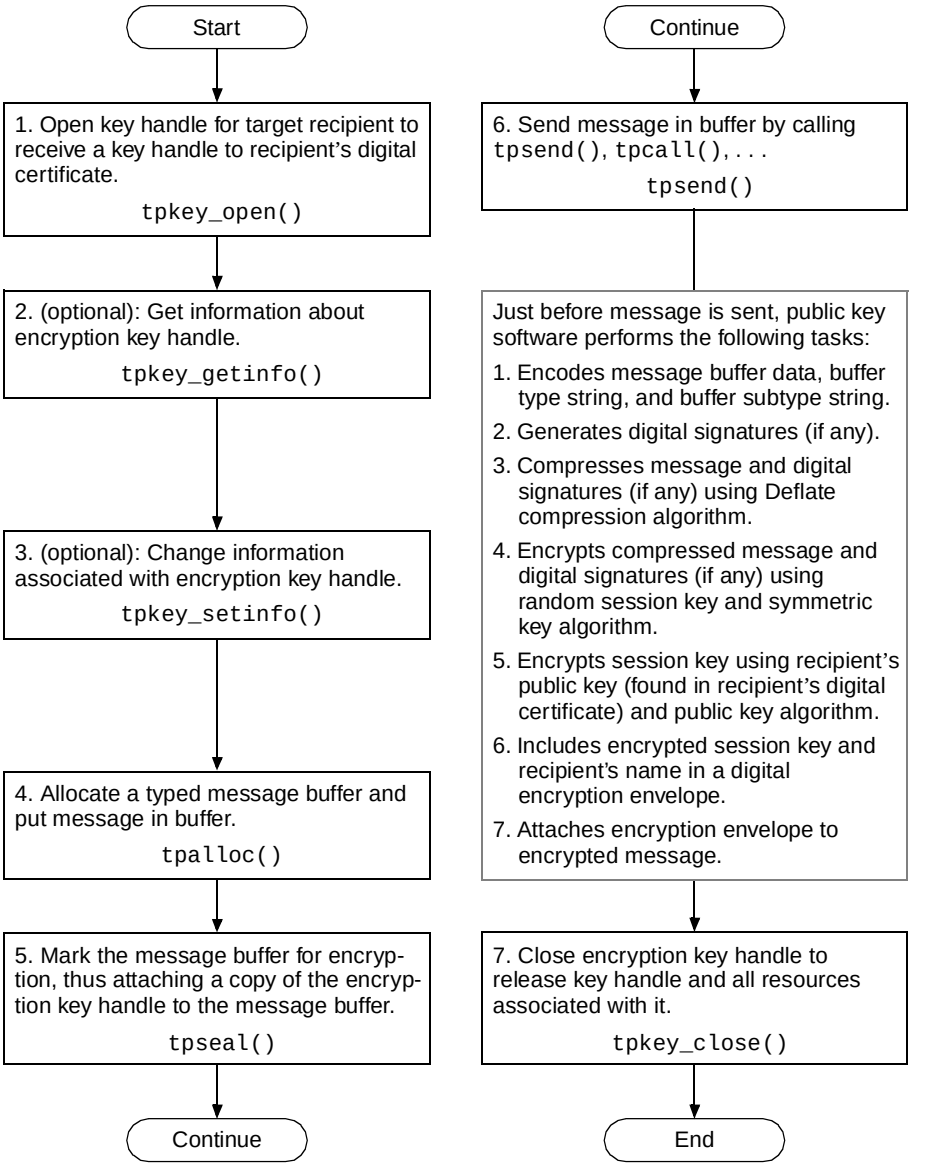
Message-based encryption provides end-to-end data privacy. For a diagram that illustrates how it works, see the figure "BEA Tuxedo PKCS-7 End-to-End Encryption" on page 1-40.

A message is encrypted just before it leaves the originating process, and remains encrypted until it is received by the final destination process. It is opaque at all intermediate transit points (including operating system message queues, system processes, and disk-based queues) and during network transmission over inter-server network links.

Writing Code to Send Encrypted Messages

The following flowchart provides the procedure for writing code to send encrypted messages.

Figure 3-5 Procedure for Sending Encrypted Messages



For details about these steps and insight into how the system encrypts a message buffer, see the following topics.

Step 1: Opening a Key Handle for Encryption

Call the `tpkey_open(3c)` function or `TPKEYOPEN(3cbl)` routine to make the digital certificate of the target recipient available to the originating process. The target recipient might be a client, a service, a server group, a gateway group, a server machine, or an entire domain of servers.

When the originating process calls `tpkey_open()` to open the key handle, it specifies either the `TPKEY_ENCRYPT` or `TPKEY_AUTOENCRYPT` flag to indicate that the handle will be used to encrypt a message buffer. Typically, a client makes this call after calling `tpinit()`, and a server makes this call as part of initializing through `tpsvrinit()`.

Opening a key handle with the `TPKEY_AUTOENCRYPT` flag enables automatic encryption: subsequently, the originating process encrypts message buffers automatically whenever they are sent. Using the `TPKEY_AUTOENCRYPT` flag is beneficial for three reasons:

- Less work is required from application programmers because fewer ATMI calls are required when operating in a secure application.
- Existing applications can leverage encryption technology with minimal coding changes.
- The possibility of programming errors that might result in an unencrypted (plaintext) buffer being sent over an insecure network is reduced.

The following example code shows how to open an encryption key handle. `TPKEY` is a special data type defined in the `atmi.h` header file.

Listing 3-7 Opening an Encryption Key Handle—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY tu_key;
  .
  .
```

```
   .
    if (tpkey_open(&tu_key, "TOUPPER", NULL,
       NULL, 0, TPKEY_ENCRYPT) == -1) {
       (void) fprintf(stderr, "tpkey_open tu failed
          tperrno=%d(%s)\n", tperrno, tpstrerror(tperrno));
       exit(1);
    }
   .
   .
   .
}
```

Step 2 (Optional): Getting Key Handle Information

You may want to get information about an encryption key handle to establish the validity of the key. To do so, call the `tpkey_getinfo(3c)` function or `TPKEYGETINFO(3cbl)` routine. While some of the information returned may be specific to a cryptographic service provider, a core set of attributes is common to all providers.

The default public key implementation supports three algorithms for bulk data encryption of message content:

- DES (DES-CBC)—A 64-bit block cipher run in Cipher Block Chaining (CBC) mode. It provides 56-bit keys (8 parity bits are stripped from the full 64-bit key) and is exportable outside the United States. (DES stands for the Data Encryption Standard.)
- 3DES (two-key triple-DES)—A 128-bit block cipher run in Encrypt-Decrypt-Encrypt (EDE) mode. 3DES provides two 56-bit keys (in effect, a 112-bit key) and is *not* exportable outside the United States.
- RC2—A variable key-size block cipher with a key size range of 40 to 128 bits. It is faster than DES and is exportable with a key size of 40 bits. A 56-bit key size is allowed for foreign subsidiaries and overseas offices of United States companies. In the United States, RC2 can be used with keys of virtually unlimited length, but the public key software restricts the key length to 128 bits. (RC2 stands for Rivest's Cipher 2.)

Encryption strength is controlled by the ENCRYPT_BITS key attribute, and the algorithm is controlled by the ENCRYPT_ALG key attribute. When an algorithm with fixed key length is set in ENCRYPT_ALG, the value of ENCRYPT_BITS is automatically adjusted to match.

The following example code shows how to get information about an encryption key handle.

Listing 3-8 Getting Information About an Encryption Key Handle—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY tu_key;
   char principal_name[PNAME_LEN];
   long pname_len = PNAME_LEN;
  .
  .
  .
   if (tpkey_getinfo(tu_key, “PRINCIPAL”,
      principal_name, &pname_len, 0) == -1) {
      (void) fprintf(stdout, “Unable to get information
         about principal: %d(%s)\n”,
            tperrno, tpstrerror(tperrno));
  .
  .
  .
      exit(1);
   }
  .
  .
  .
}
```

Step 3 (Optional): Changing Key Handle Information

To set optional attributes associated with an encryption key handle, call the `tpkey_setinfo(3c)` function or `TPKEYSETINFO(3cbl)` routine. Key handle attributes vary, depending on the cryptographic service provider.

The following example code shows how to change information associated with an encryption key handle.

Listing 3-9 Changing Information Associated with an Encryption Key Handle—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY tu_key;
   static const unsigned char rc2_objid[] = {
      0x06, 0x08, 0x2a, 0x86, 0x48, 0x86, 0xf7, 0x0d, 0x03, 0x02
   };
  .
  .
  .
   if (tpkey_setinfo(tu_key, “ENCRYPT_ALG”, (void *) rc2_objid,
      sizeof(rc2_objid), 0) == -1) {
      (void) fprintf(stderr, “tpkey_setinfo failed
         tperrno=%d(%s)\n”,
            tperrno, tpstrerror(tperrno));
      return(1);
   }
  .
  .
  .
}
```

Step 4: Allocating a Buffer and Putting a Message in the Buffer

To allocate a typed message buffer, call the `tpalloc(3c)` function. Then put a message in the buffer.

Step 5: Marking the Buffer for Encryption

To mark, or register, the message buffer for encryption, call the `tpseal(3c)` function. By calling this function, you attach a copy of the encryption key handle to the message buffer. If you open the key with the `TPKEY_AUTOENCRYPT` flag, each message that you send is automatically marked for encryption without an explicit call to `tpseal()`.

Note: In COBOL applications, use the `AUTOENCRYPT` settings member to encrypt a message buffer. See `TPKEYOPEN(3cbl)`.

The following example code shows how to mark a message buffer for encryption.

Listing 3-10 Marking a Message Buffer For Encryption—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY tu_key;
   char *sendbuf, *rcvbuf;
  .
  .
  .
   if (tpseal(sendbuf, tu_key, 0) == -1) {
      (void) fprintf(stderr, "tpseal failed tperrno=%d(%s)\n",
         tperrno, tpstrerror(tperrno));
      tpfree(rcvbuf);
      tpfree(sendbuf);
      tpterm();
      (void) tpkey_close(tu_key, 0);
      exit(1);
   }
  .
  .
  .
}
```

Step 6: Sending the Message

After the message buffer has been marked for encryption, transmit the message buffer using one of the following C functions or COBOL routines:

- `tpcall()` or `TPCALL`
- `tpbroadcast()` or `TPBROADCAST`
- `tpconnect()` or `TPCONNECT`
- `tpenqueue()` or `TPENQUEUE`
- `tpforward()`
- `tpnotify()` or `TPNOTIFY`
- `tppost()` or `TPPOST`
- `tpreturn()` or `TPRETURN`
- `tpsend()` or `TPSEND`

Step 7: Closing the Encryption Key Handle

Call the `tpkey_close(3c)` function or `TPKEYCLOSE(3cbl)` routine to release the encryption key handle and all resources associated with it.

How the System Encrypts a Message Buffer

Just before a message buffer is sent, the public key software encrypts the message and attaches an encryption envelope; the encryption envelope enables the target recipient to decrypt the message. If a sealed buffer is transmitted more than once, encryption is performed for each transmission. This process makes it possible to modify a message buffer after marking the buffer to be encrypted.

The public key software encrypts the content of the message buffer and generates an encryption envelope for the recipient of the encrypted message by performing the following two-step procedure.

1. {message_buffer_data + buffer_type_string + buffer_subtype_string}session_key = encrypted_message
2. {session_key}recipient's_public_key = encrypted_session_key = encryption_envelope_for_recipient

The notation {*something*}*key* means that *something* has been encrypted or decrypted using *key*. In Step 1, a message buffer is encrypted using the session key, and in Step 2, the session key is encrypted using the recipient's public key.

Multiple Message Recipients

More than one encryption envelope can be associated with a message buffer, which means that multiple recipients, with different private keys, can receive and decrypt an encrypted message. A recipient can be a person or a process. When a message is encrypted for multiple recipients, it is encrypted only once, but the session key is encrypted with the public key of each recipient. All encryption envelopes are attached to the encrypted message.

If several encryption envelopes are associated with one message buffer, all of them must use the same symmetric key algorithm and the same key size for that algorithm.

Encrypted Message Content

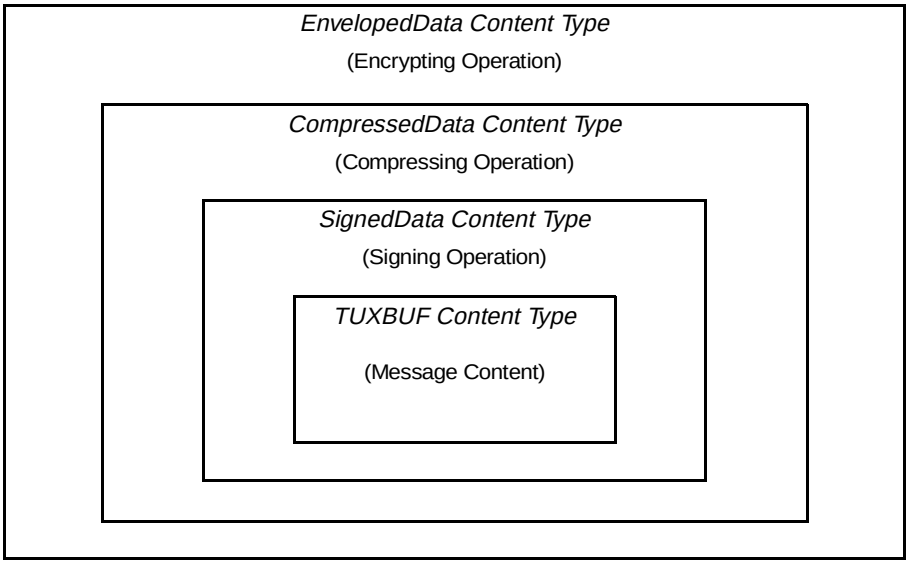
An encrypted message buffer is represented in the PKCS-7 format as a version 0 `EnvelopedData` content type. The `EnvelopedData` content type, as used by the BEA Tuxedo system, consists of the following items:

- A list of recipients (in plaintext) that can be read by any BEA Tuxedo process
- Encryption envelopes for one or more recipients
- Public key algorithm (and any associated parameters) under which the session key was encrypted
- Symmetric key algorithm (and any associated parameters) under which the bulk data was encrypted
- Encrypted bulk data, which is a composite of message buffer data, buffer type string, buffer subtype string, and digital signatures (if any) that have undergone the following transformations:
  - Conversion of the message buffer data, buffer type string, and buffer subtype string into the BEA Tuxedo encoded format to form the composite encoded data. (The BEA Tuxedo encoded format allows a message buffer to be decrypted on any machine architecture.)

- Compression of the composite encoded data and digital signatures (if any) using the Deflate compression algorithm to form the composite compressed data.
- Encryption of the composite compressed data under a randomly generated session key and symmetric key algorithm (identified earlier in this list) to form the encrypted bulk data.

The following figure shows the envelope hierarchy for the `EnvelopedData` content type. The `SignedData` content type is part of the hierarchy *only* if the message to which it belongs has one or more associated digital signatures.

Figure 3-6 EnvelopedData Content Type



As shown in the preceding figure, a message buffer may be both signed and encrypted. No relationship is required between the number of digital signatures and the number of encryption envelopes associated with a message buffer.

When both processes are performed on a message buffer, signatures are generated first, on unencrypted data. The number of attached signatures and the identity of signing parties are then obscured by the bulk data encryption.

Note: A suitable decryption key must be available to access message data before signatures can be verified.

Writing Code to Receive Encrypted Messages

The procedure for writing code to receive encrypted messages consists of the following steps.

1. Call `tpkey_open()` to open a key handle for the target recipient. `tpkey_open` returns a key handle to the recipient's private key and digital certificate.
2. (Optional): Call `tpkey_getinfo()` to get information about the decryption key handle.
3. (Optional): Call `tpkey_setinfo()` to change information associated with the decryption key handle.
4. Call `tpkey_close()` to close the decryption key handle. `tpkey_close()` releases the key handle and all resources associated with it.

For details about these steps and insight into how the system decrypts a message buffer, see the following topics.

Step 1: Opening a Key Handle for Decryption

Call the `tpkey_open(3c)` function or `TPKEYOPEN(3cbl)` routine to make the private key and the associated digital certificate of the target recipient available to the receiving process. The receiving process might be a client, a service, a server group, a gateway group, a server machine, or an entire domain of servers.

An application administrator can configure the application's `UBBCONFIG` file such that decryption key handles are opened automatically when the application is booted. No more than one decryption key handle per server may be used with this method. See "Initializing Decryption Keys Through the Plug-ins" on page 2-50 for details.

If an application is not configured to open a decryption key handle for the receiving process during booting, the receiving process initiates its own `tpkey_open()` call. Or, if the receiving process wants to open another decryption key handle, the receiving process makes an additional `tpkey_open()` call.

In order to access the target recipient's private key, the receiving process must prove its right to act as the target recipient. Proof requirements depend on the implementation of the public key plug-in interface. The default public key implementation requires a secret password from the calling process.

When the receiving process calls `tpkey_open()` to open the key handle, it specifies the `TPKEY_DECRYPT` flag to indicate that the handle will be used to decrypt a message buffer. Typically, a client makes this call after calling `tpinit()`, and a server makes this call as part of initializing through `tpsvrinit()`.

The following example code shows how to open a decryption key handle. `TPKEY` is a special data type defined in the `atmi.h` header file.

Listing 3-11 Opening a Decryption Key Handle—Example

```
TPKEY tu_key;

tpsvrinit(argc, argv)
int argc;
char **argv;
#endif
{
   char  *tu_location;
  .
  .
  .
   if (tpkey_open(&tu_key, "TOUPPER", tu_location,
      NULL, 0, TPKEY_DECRYPT) == -1) {
      userlog("Unable to open private key: %d(%s)",
         tperrno, tpstrerror(tperrno));
   return(-1)
   }
  .
  .
  .
}
```

Step 2 (Optional): Getting Key Handle Information

You may want to get information about a decryption key handle to establish the validity of the key. To do so, call the `tpkey_getinfo(3c)` function or `TPKEYGETINFO(3cbl)` routine. While some of the information returned may be specific to a cryptographic service provider, a core set of attributes is common to all providers.

The following example code shows how to get information about a decryption key handle.

Listing 3-12 Getting Information About a Decryption Key Handle—Example

```
TPKEY tu_key;

tpsvrinit(argc, argv)
int argc;
char **argv;
#endif
{
   char principal_name[PNAME_LEN];
   long pname_len = PNAME_LEN;
  .
  .
  .
   if (tpkey_getinfo(tu_key, “PRINCIPAL”,
      principal_name, &pname_len, 0) == -1) {
      (void) fprintf(stdout, “Unable to get information
         about principal: %d(%s)\n”,
            tperrno, tpstrerror(tperrno));
  .
  .
  .
      exit(1);
   }
  .
  .
  .
}
```

Step 3 (Optional): Changing Key Handle Information

To set optional attributes associated with a decryption key handle, call the `tpkey_setinfo(3c)` function or `TPKEYSETINFO(3cbl)` routine. Key handle attributes vary, depending on the cryptographic service provider.

The following example code shows how to change information associated with a decryption key handle.

Listing 3-13 Changing Information Associated with a Decryption Key Handle—Example

```
TPKEY tu_key;

tpsvrinit(argc, argv)
int argc;
char **argv;
#endif
{
   TM32U mybits = 128;
  .
  .
  .
   if (tpkey_setinfo(tu_key, “ENCRYPT_BITS”, &mybits,
      sizeof(mybits), 0) == -1) {
      (void) fprintf(stderr, “tpkey_setinfo failed
         tperrno=%d(%s)\n”,
            tperrno, tpstrerror(tperrno));
      return(1);
   }
  .
  .
  .
}
```

Step 4: Closing the Decryption Key Handle

Call the `tpkey_close(3c)` function or `TPKEYCLOSE(3cbl)` routine to release the decryption key handle and all resources associated with it.

How the System Decrypts a Message Buffer

The public key software automatically decrypts an encrypted message buffer whenever it enters a BEA Tuxedo client process, server process, or any system process that needs to access the content of the message buffer. For automatic decryption to succeed, the receiving process must have opened a decryption key (type `TPKEY_DECRYPT`) corresponding to a recipient identified in one of the attached encryption envelopes.

Upon receiving an encrypted message, the public key software, operating on behalf of the receiving process, performs the following tasks.

1. Reads the target recipient's name on the attached encryption envelope.
2. To recover the session key, decrypts the recipient's encryption envelope using the recipient's private key and the public key algorithm.
3. Decrypts the message using the recovered session key and the symmetric key algorithm.
4. Uncompresses the message.
5. Verifies digital signatures if any. (See "How a Signed Message Is Received" on page 3-32.)
6. If the message buffer successfully passes the check performed in Step 5, the public key software decodes the message buffer data, buffer type string, and buffer subtype string, and then passes the plaintext message to the receiving process. This step reverses the encoding performed by the originating process. (The BEA Tuxedo encoded format allows a message buffer to be decrypted on any machine architecture.)

Note: If none of the attached digital signatures can be verified or the message buffer cannot be decrypted, the receiving process does *not* receive the message buffer. Moreover, the receiving process has no knowledge of the message buffer.

If a system process is acting as a *conduit* (that is, if it is not reading the content of the message), then the message need not be decrypted. Bridges and Workstation Handlers (WSHs) are examples of system processes acting as conduits.

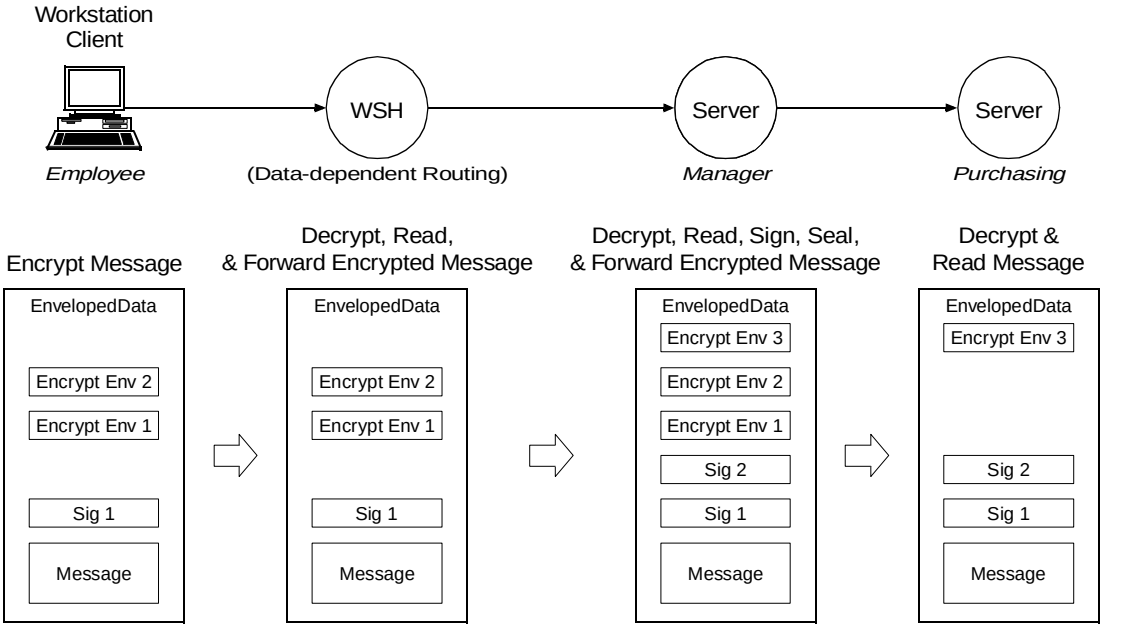
The WSH is a special example of a conduit. If a WSH is configured for data-dependent routing, it needs to read the received message buffer to determine how to route the buffer. The public key software makes a copy of the received message buffer, decrypts the copy, and then passes the decrypted copy to the WSH. The WSH analyzes the decrypted copy to determine how to route the buffer, and then routes the original message buffer *unchanged* to the appropriate server. (For more detail about the interaction between data-dependent routing and public key security, see "Compatibility/Interaction with Data-dependent Routing" on page 1-60.)

Discarding an Input Buffer's Encryption Envelopes

If a message buffer is passed to an ATMI function (such as `tpacall()`) as an input parameter, the public key software discards any encryption envelopes previously attached to the message. This behavior prevents the target recipients for the original message from receiving any modifications made by an intermediate process.

As an example of this process, consider the scenario shown in the following figure.

Figure 3-7 Forwarding a Signed and Encrypted Message—Example



A server process named Manager receives a signed and encrypted message buffer from a client process named Employee, decrypts and reads the received message buffer, signs and seals it for a service named Purchasing, and then forwards the message to Purchasing.

The following is a detailed description of how this operation is performed.

1. The Workstation Handler (WSH) receives the signed and encrypted message buffer from the employee and forwards it *as is*.

   The WSH process is configured for data-dependent routing, which is briefly described in "How the System Decrypts a Message Buffer" on page 3-48. The public key software uses a decryption key previously opened for the WSH process to decrypt a copy of the received message buffer, and then passes the decrypted copy to the WSH. After analyzing the decrypted copy, the WSH routes the received message buffer to the Manager process *as is*.

   If the WSH process is *not* configured for data-dependent routing, the Employee process does not need to tpseal() the message buffer for the WSH process, and the WSH process does not need to open a decryption key.

   Regardless of how it is configured, the WSH does not verify digital signatures.

2. When the message buffer arrives at the Manager process, the public key software:

   a. Decrypts the message buffer using a decryption key previously opened for the Manager process

   b. Verifies the employee's signature

   c. Passes the message *without* digital signature or encryption information to the Manager

   When a process receives a message buffer, it receives *only* the message content. Any digital signatures or encryption envelopes associated with the message buffer are not included.

3. The Manager calls tpenvelope() *repeatedly* to find out about the digital signature and encryption information associated with the message buffer. tpenvelope() returns:

- Digital signature information, including the signer's public key and a digital-signature status of `TPSIGN_OK`
- Encryption information, including the public keys of the WSH process and the `Manager` process itself

4. The `Manager` calls `tpkey_getinfo()` with the signer's public key as an argument, to obtain more information about the signer, including the signer's principal name.
5. If the `Manager` determines that the signer is a known employee and that the employee's request (as stated in the message content) is valid, the `Manager` proceeds as follows.

   a. Calls `tpsign()` to mark the message buffer for digital signature by the `Manager`.

   a. Calls `tpseal()` to mark the message buffer to be encrypted for `Purchasing`.

   b. Calls `tpforward()` (or some other function used to transmit data) to send the message to `Purchasing`.

Just before the message is transmitted, the public key software performs the following tasks.

1. Generates a digital signature for the `Manager`
2. Verifies the employee's digital signature
3. Encrypts the message content and associated digital signatures
4. Creates an encryption envelope for `Purchasing`

Replacing an Output Buffer's Encryption Envelopes

If a message buffer is passed to an ATMI function (such as `tpgetrply()`) as an output parameter, the public key software deletes any encryption information associated with the buffer. This information includes any *pending* seals, or seals from previous uses of the buffer. (A pending seal is a recipient's seal that is registered with a message buffer.)

New encryption information might be associated with the new buffer content after successful completion of the operation.

See Also

- "Examining Digital Signature and Encryption Information" on page 3-52
- "Externalizing Typed Message Buffers" on page 3-59
- "Public Key Security" on page 1-29
- "Administering Public Key Security" on page 2-41
- "Programming an Application with Security" on page 3-3

Examining Digital Signature and Encryption Information

The public key software maintains the order in which:

- Digital-signature registration requests and digital signatures are attached to a message buffer
- Encryption registration requests and encryption envelopes are attached to a message buffer

A process obtains this information by calling the `tpenvelope()` function with the target message buffer as an argument. `tpenvelope()` is described on the `tpenvelope(3c)` reference page in *BEA Tuxedo C Function Reference*.

There may be multiple occurrences of digital-signature registration requests, digital signatures, encryption registration requests, and encryption envelopes associated with a message buffer. The occurrences are stored in sequence, with the first item at the zero position and subsequent items in consecutive positions. The `occurrence` input parameter for `tpenvelope()` indicates which item is being requested. When the value of `occurrence` is beyond the position of the last item, `tpenvelope()` fails with the `TPENOENT` error condition. A process can examine all items by calling `tpenvelope()` repeatedly until `TPENOENT` is returned.

In an originating process, digital signature and encryption information is generally in a pending state, waiting until the message is sent. In a receiving process, digital signatures have already been verified, and encryption and decryption have already been performed.

What Happens When an Originating Process Calls tpenvelope

When an originating process calls `tpenvelope()` with the originating message buffer as an argument, `tpenvelope()` reports:

- Any digital signature request *explicitly* registered with the message buffer as being in the `TPSIGN_PENDING` state. The originating process explicitly registers a digital signature request by calling the `tpsign(3c)` function.
- Any digital signature request *implicitly* registered with the message buffer as also being in the `TPSIGN_PENDING` state. The originating process implicitly registers a digital signature request by calling `tpkey_open(3c)` with the `TPKEY_AUTOSIGN` flag specified.
- Any encryption (seal) request *explicitly* registered with the message buffer as being in the `TPSEAL_PENDING` state. The originating process explicitly registers an encryption request by calling the `tpseal(3c)` function.
- Any encryption (seal) request *implicitly* registered with the message buffer as also being in the `TPSEAL_PENDING` state. The originating process implicitly registers an encryption request by calling `tpkey_open()` with the `TPKEY_AUTOENCRYPT` flag specified.

In addition to the status, `tpenvelope()` returns the key handle associated with a digital signature or encryption registration request. A process can call the `tpkey_getinfo(3c)` function with the key handle as an argument, to get more information about the key handle.

What Happens When a Receiving Process Calls tpenvelope

When a process receives a message buffer, it receives *only* the message content. Any digital signatures or encryption envelopes associated with the message buffer are not included. The receiving process must call `tpenvelope()` to obtain information about any attached digital signatures or encryption envelopes.

When a receiving process calls `tpenvelope()` with the received message buffer as an argument, `tpenvelope()` reports:

- Any digital signature attached to the message buffer. A digital signature has one of the following states:
  - `TPSIGN_OK`

    Digital signature has been verified.
  - `TPSIGN_TAMPERED_MESSAGE`

    Digital signature is not valid because the content of the message buffer has been altered.
  - `TPSIGN_TAMPERED_CERT`

    Digital signature is not valid because the signer's digital certificate has been altered.
  - `TPSIGN_REVOKED_CERT`

    Digital signature is not valid because the signer's digital certificate has been revoked.
  - `TPSIGN_POSTDATED`

    Digital signature is not valid because its timestamp is too far into the future.
  - `TPSIGN_EXPIRED_CERT`

    Digital signature is not valid because the signer's digital certificate has expired.
  - `TPSIGN_EXPIRED`

    Digital signature is not valid because its timestamp is too old.

- TPSIGN_UNKNOWN

  Digital signature is not valid because the signer's digital certificate was issued by an unknown Certification Authority (CA).

- Any encryption envelope attached to the message buffer. An encryption envelope has one of the following states:

  - TPSEAL_OK

    Encryption envelope is valid.

  - TPSEAL_TAMPERED_CERT

    Encryption envelope is not valid because the target recipient's digital certificate has been altered. (Target recipient will *not* receive the message buffer.)

  - TPSEAL_REVOKED_CERT

    Encryption envelope is not valid because the target recipient's digital certificate has been revoked. (Target recipient will *not* receive the message buffer.)

  - TPSEAL_EXPIRED_CERT

    Encryption envelope is not valid because the target recipient's digital certificate has expired. (Target recipient will *not* receive the message buffer.)

  - TPSEAL_UNKNOWN

    Encryption envelope is not valid because the target recipient's digital certificate was issued by an unknown CA. (Target recipient will *not* receive the message buffer.)

In addition to the status, `tpenvelope()` returns the key handle associated with a digital signature or encryption envelope. A process can call the `tpkey_getinfo(3c)` function with the key handle as an argument, to get more information about the key handle.

If a receiving process calls `tpsign()` to register a digital signature request after receiving the message buffer, `tpenvelope()` reports the status of the registration as `TPSIGN_PENDING`. Similarly, if a receiving process calls `tpseal()` to register an encryption (seal) request after receiving the message buffer, `tpenvelope()` reports the status of the registration as `TPSEAL_PENDING`.

If a receiving process modifies the content of a *signed* message buffer after receiving it, the attached signatures are no longer valid. As a result, `tpenvelope()` cannot verify the signatures, and reports a signature status of `TPSIGN_TAMPERED_MESSAGE`.

Understanding the Composite Signature Status

For a message buffer with multiple digital signatures, the public key software calls an internal function equivalent to `tpenvelope()` to examine the state of each digital signature. Then, by observing certain rules, the public key software forms a *composite signature status*. The rules for forming a composite signature status are shown in the following table.

Table 3-4 Composite Signature Status

If any status is . . .	And there is no status of . . .	Then the composite status is . . .
TPSIGN_TAMPERED_MESSAGE	. . .	TPSIGN_TAMPERED_MESSAGE
TPSIGN_TAMPERED_CERT	TPSIGN_TAMPERED_MESSAGE	TPSIGN_TAMPERED_CERT
TPSIGN_REVOKED_CERT	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT	TPSIGN_REVOKED_CERT
TPSIGN_POSTDATED	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT	TPSIGN_POSTDATED
TPSIGN_EXPIRED_CERT	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED	TPSIGN_EXPIRED_CERT
TPSIGN_OK	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED TPSIGN_EXPIRED_CERT	TPSIGN_OK

Table 3-4 Composite Signature Status

If any status is . . .	And there is no status of . . .	Then the composite status is . . .
TPSIGN_EXPIRED	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED TPSIGN_EXPIRED_CERT TPSIGN_OK	TPSIGN_EXPIRED
TPSIGN_UNKNOWN	TPSIGN_TAMPERED_MESSAGE TPSIGN_TAMPERED_CERT TPSIGN_REVOKED_CERT TPSIGN_POSTDATED TPSIGN_EXPIRED_CERT TPSIGN_OK TPSIGN_EXPIRED	TPSIGN_UNKNOWN

Any incoming message buffer *without* a composite signature status of TPSIGN_OK or TPSIGN_UNKNOWN is discarded as if it were never received. If the SIGNATURE_REQUIRED parameter is set to Y (yes) in the application's UBBCONFIG file, then any incoming message buffer *without* a composite signature status of TPSIGN_OK is discarded as if it were never received. See "Enforcing the Signature Policy for Incoming Messages" on page 2-44 for more detail.

An exception to the handling of signed message buffers described in the previous paragraph is the tpimport(3c) function. The tpimport(3c) function delivers an incoming message buffer regardless of the composite signature status.

Example Code for tpenvelope

The following example code shows how to use tpenvelope() to examine the digital signature and encryption information associated with a message buffer.

Listing 3-14 Using tpenvelope—Example

```
main(argc, argv)
int argc;
char *argv[];
#endif

{
   TPKEY tu_key;
   TPKEY sdo_key;
   TPKEY output_key;
   char *sendbuf, *rcvbuf;
   int ret;
   int occurrence = 0;
   long status;
   char principal_name[PNAME_LEN];
   long pname_len = PNAME_LEN;
   int found = 0;
  .
  .
  .
   output_key = NULL;
   ret = tpenvelope(rcvbuf, 0, occurrence, &output_key,
      &status, NULL, 0);

   while (ret != -1) {
      if (status == TPSIGN_OK) {
         if (tpkey_getinfo(output_key, "PRINCIPAL",
            principal_name, &pname_len, 0) == -1) {
            (void) fprintf(stdout, "Unable to get information
               about principal: %d(%s)\n",
                  tperrno, tpstrerror(tperrno));
            tpfree(sendbuf);
            tpfree(rcvbuf);
            tpterm();
            (void) tpkey_close(tu_key, 0);
            (void) tpkey_close(sdo_key, 0);
            (void) tpkey_close(output_key, 0);
            exit(1);
         }

         /* Do not forget to free resources */
         (void) tpkey_close(output_key, 0);
         output_key = NULL;
         found = 1;
         break;
      }
```

```
        /* Do not forget to free resources */
        (void) tpkey_close(output_key, 0);
        output_key = NULL;

        occurrence++;
        ret = tpenvelope(rcvbuf, 0, occurrence, &output_key,
           &status, NULL, 0);
    }
   .
   .
   .
}
```

See Also

- "Externalizing Typed Message Buffers" on page 3-59
- "Public Key Security" on page 1-29
- "Administering Public Key Security" on page 2-41
- "Programming an Application with Security" on page 3-3

Externalizing Typed Message Buffers

An externalized representation is a message buffer that does *not* include any BEA Tuxedo header information that is normally added to a message buffer just before the buffer is transmitted. An externalized representation of a signed message buffer enables "pass through" transmission of signed data and long-term storage of the signed buffer for non-repudiation. It also enables an encrypted message buffer to be transported through intermediate processes without access to a decryption key.

How to Create an Externalized Representation

A process converts a typed message buffer into an externalized representation by calling the `tpexport(3c)` function. Pending signatures associated with a message buffer are generated at the time `tpexport()` is called, just as if the buffer were being transmitted to another process by an ATMI function. Similarly, pending seals associated with a message buffer are generated at the time `tpexport()` is called, just as if the buffer were being transmitted to another process by an ATMI communication function.

The externalized representation of a message buffer is stored in the PKCS-7 format, which is a binary format. If a string format is required, the calling process must call `tpexport()` with the `TPEX_STRING` flag specified.

Note: The ability to create an externalized representation of a typed message buffer is not unique to public key security. A process may call `tpexport()` to externalize a typed message buffer regardless of whether a message buffer is marked for digital signature or encryption.

How to Convert an Externalized Representation

A receiving process calls the `tpimport(3c)` function to convert the externalized representation of a message buffer into a typed message buffer. The `tpimport()` function also performs decryption, if necessary, and verifies any associated digital signatures.

Example Code for tpexport and tpimport

The following example code shows how to use `tpexport()` to convert a typed message buffer into an externalized representation, and how to use `tpimport()` to convert the externalized representation back into a typed message buffer.

Listing 3-15 Using tpexport and tpimport—Example

```
static void hexdump _((unsigned char *, long));

#define MAX_BUFFER 80000

main(argc, argv)
int argc;
char *argv[];
#endif

{
   char  *databuf;
   char  exportbuf[MAX_BUFFER];
   long  exportbuf_size = 0;
   char  *importbuf = NULL;
   long  importbuf_size = 0;
   int go_on = 1;
  .
  .
  .
   exportbuf_size = 0;
   while (go_on == 1) {
      if (tpexport(databuf, 0, exportbuf, &exportbuf_size, 0)
         == -1) {
         if (tperrno == TPELIMIT) {
            printf("%d tperrno is TPELIMIT, exportbuf_size=%ld\n",
               __LINE__, exportbuf_size);
            if (exportbuf_size > MAX_BUFFER) {
               return(1);
            }
         }
         else {
            printf("tpexport(%d) failed: tperrno=%d(%s)\n",
               __LINE__, tperrno, tpstrerror(tperrno));
            return(1);
         }
      }
      else {
         go_on = 0;
      }
   }
  .
  .
  .

   hexdump((unsigned char *) exportbuf, (long) exportbuf_size);
```

```
   if (tpimport(exportbuf, exportbuf_size, &importbuf,
      &importbuf_size, 0) == -1) {
      printf("tpimport(%d) failed: tperrno=%d(%s)\n",
         __LINE__, tperrno, tpstrerror(tperrno));
      return(1);
   }
  .
  .
  .
}
```

See Also

- "Public Key Security" on page 1-29
- "Administering Public Key Security" on page 2-41
- "Programming an Application with Security" on page 3-3